Tiling and Asynchronous Communication Optimizations for Stencil Computations

Thesis by

Tareq Majed Yasin Malas

In Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy

King Abdullah University of Science and Technology, Thuwal,

Kingdom of Saudi Arabia

October, 2015

The thesis of Tareq Majed Yasin Malas is approved by the examination committee

Committee Chairperson: Professor David Keyes

Committee Member: Professor Mootaz Elnozahy

Committee Member: Professor Basem Shihada

Committee Member: Professor David Ketcheson

Committee Member: Professor Satoshi Matsuoka

Committee Member: Dr. Hatem Ltaief

ABSTRACT

Tiling and Asynchronous Communication Optimizations for Stencil Computations

Tareq Majed Yasin Malas

The importance of stencil-based algorithms in computational science has focused attention on optimized parallel implementations for multilevel cache-based processors. Temporal blocking schemes leverage the large bandwidth and low latency of caches to accelerate stencil updates and approach theoretical peak performance. A key ingredient is the reduction of data traffic across slow data paths, especially the main memory interface. Most of the established work concentrates on updating separate cache blocks per thread, which works on all types of shared memory systems, regardless of whether there is a shared cache among the cores. This approach is memory-bandwidth limited in several situations, where the cache space for each thread can be too small to provide sufficient in-cache data reuse.

We introduce a generalized multi-dimensional intra-tile parallelization scheme for shared-cache multicore processors that results in a significant reduction of cache size requirements and shows a large saving in memory bandwidth usage compared to existing approaches. It also provides data access patterns that allow efficient hardware prefetching. Our parameterized thread groups concept provides a controllable trade-off between concurrency and memory usage, shifting the pressure between the memory interface and the Central Processing Unit (CPU).

We also introduce efficient diamond tiling structure for both shared memory cache blocking and distributed memory relaxed-synchronization communication, demonstrated using one-dimensional domain decomposition. We describe the approach and our open-source testbed implementation details (called *Girih*), present performance results on contemporary Intel processors, and apply advanced performance modeling techniques to reconcile the observed performance with hardware capabilities. Furthermore, we conduct a comparison with the state-of-the-art stencil frameworks PLUTO and Pochoir in shared memory, using corner-case stencil operators. We study the impact of the diamond tile size on computational intensity, cache block size, and energy consumption. The impact of computational intensity on power dissipation on the CPU and in the DRAM is investigated and shows that DRAM power is a decisive factor for energy consumption in the Intel Ivy Bridge processor, which is strongly influenced by the computational intensity. Moreover, we show that highest performance does not necessarily lead to lowest energy even if the clock speed is fixed. We apply our approach to an electromagnetic simulation application for solar cell development, demonstrating several-fold speedup compared to an efficient spatially blocked variant. Finally, we discuss the integration of our approach with other techniques for future High Performance Computing (HPC) systems, which are expected to be more memory bandwidth-starved with a deeper memory hierarchy.

ACKNOWLEDGEMENTS

I would like to thank sincerely my advisor Professor David Keyes for supporting my Ph.D. under the Extreme Computing Research Center (ECRC) at KAUST. I thank my parents and my wife for their continuous encouragement and for bearing with me for my preoccupation during this journey and their deep moral support at all times.

I am in debt to Georg Hager and Gerhard Wellein from Erlangen University. Their long experience in High Performance Computing and performance modeling was critical in improving our research, through our collaboration and regular meetings.

Many thanks to my fellow students and the research scientists of the ECRC for the long fruitful discussions we had and for their support.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

1WD	Single-threaded Wavefront Diamond blocking
CFL	Courant-Friedrichs-Lewy
CL	Cache Line
CPU	Central Processing Unit
DP	Double Precision
DSL	Domain Specific Language
ECM	Execution-Cache-Memory
FDFD	Finite-Difference Frequency Domain
FED	Fixed-Execution to Data
FIFO	First In First Out
FIT	Finite Integration Technique
FMA	Fused Multiply-Add
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
HPC	High Performance Computing
KNC	Knight's Corner
LIFO	Last In First Out
LLC	Last-Level Cache
LUP	Lattice-site Update
MPI	Message Passing Interface

MWD	Multi-threaded Wavefront Diamond blocking
NUMA	Non-Uniform Memory Access
PDE	Partial Differential Equation
PTX	Parallel Thread Execution
PV	Photovoltaic
RAPL	Running Average Power Level
SIMD	Single Instruction Multiple Data
SMX	Streaming Multiprocessor
TG	Thread Group
THIIM	Time Harmonic Inverse Iteration Method
TLB	Translation Lookaside Buffer
UMA	Uniform Memory Access

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

In this chapter we introduce the problem, motivation, and the contribution of our work. We also provide an overview of this thesis.

1.1 Problem statement

In this thesis, we introduce data traversal ordering schemes that offer a substantial reduction in memory bandwidth usage and cache size requirements in the memory bandwidth demanding stencil computations on multi- and many-core processors. Performance modeling of the baseline and the proposed techniques is utilized to provide tight bounds on the achievable performance and provide sufficient understanding of the obtained performance. We investigate the benefit of our techniques on corner-case stencil kernels and a solar cell simulation application to improve their performance on both contemporary and future processors. Our proposed approach performs efficient multi-dimensional intra-tile parallelization of wavefront and diamond temporal blocking/tiling schemes. Diamond tiling provides a convenient data structure for performing domain decomposition in a distributed memory configuration. Combining wavefront blocking with diamond tiling maximizes the in-cache data reuse in three-dimensional solution domains.

1.2 Motivation

The evaluation of stencil operators on Cartesian lattices is a classic kernel in computational science and engineering, arising from systems that are "born" discrete and from discretizations of PDEs, both explicit and implicit. In the implicit case the iteration index is analogous to explicit time and stencil evaluation becomes a case of sparse matrix-vector multiplication with special structure. Lattice values are updated from neighbors at a previous time step or iteration sweep with concurrency that scales linearly with the number of degrees of freedom. However, low flop-per-byte ratios put a premium on locality: regular access patterns allow high spatial locality, in the sense of packing cache blocks. The modest temporal locality within a single iteration from reuse of a value within several adjacent stencils can be enhanced across iterations, with the explicit goal of bringing the *code balance*, i.e., the ratio of memory data traffic to arithmetic work, closer to the *machine balance*, i.e., the ratio of memory bandwidth to peak arithmetic performance [1]. On modern hardware, and for most non-optimized stencil algorithm implementations, the machine balance is usually much smaller than the code balance, which leads to performance being governed by the memory bandwidth. Any reduction of code balance will thus lead to a proportional performance increase, up to a point where execution decouples from memory and other bottlenecks apply.

In future computer architectures each node may have up to a thousand shared-memory cores with: small memory bandwidth per core, small cache size per core, complex cache sharing among cores, expensive synchronization among all the cores, interaction between heterogeneous processors, and expensive intra-node lockstep synchronization after each iteration of stencil computations. The development of algorithms that can run these stencil computations efficiently on the emerging HPC systems is essential.

1.3 Contribution

We propose a multi-dimensional intra-tile parallelization scheme using multi-core wavefront diamond blocking for optimizing practically relevant stencil algorithms. The results demonstrate a substantial reduction in cache block size and memory bandwidth requirements using four corner-case stencil types that represent a full range of practically important stencil computations. In contrast to many temporal blocking approaches in the literature, our approach efficiently utilizes the shared cache between threads of modern processors. It also provides a controllable tradeoff between memory bandwidth per thread and frequency of synchronization to alleviate the bottleneck at the CPU or memory interface, as needed when applying a stencil to a particular architecture. Relaxed synchronization of Message Passing Interface (MPI) messages and overlapping of computations with communication in distributed implementations is achieved through the structure of diamond tiling.

Our scheme provides cache block sharing along the leading dimension (the dimension of most rapid index advance in a Cartesian ordering) that results in better utilization of hardware prefetching to the shared cache level. We introduce a novel Fixed-Execution to Data (FED) wavefront parallelization technique that reduces the data movement in the cache hierarchy of the processor by using tiling hyperplanes that are parallel to the time dimension. Our approach achieves hierarchical cache blocking by using large tiles in the shared cache level and fitting subsets of the tiles in the private caches of the threads, providing cache blocks that span multiple cache domains. Our implementation uses an efficient runtime system to dynamically schedule tiles to thread groups. We also develop an efficient fine-grained synchronization scheme to coordinate the work of the thread groups and avoid race conditions. Finally, coupled with auto-tuning, our cache block sharing algorithm provides a rich set of run-time configurable options that allow architecture-friendly data access patterns for various setups.

1.4 Thesis outline

Chapter 2 provides a background of stencil computations and temporal blocking techniques relevant to our contribution. We motivate our approach through comprehensive analysis of efficient spatial and temporal blocking techniques over stencil types representing corner cases in Chapter 3. Chapter 4 presents our main contribution, a multi-dimensional intra-tile parallelization scheme that is implemented using MWD tiling, extending previous state-of-the-art to the many-core frontier. Experimental results of our proposed approach including an extension to distributed memory are presented in Chapter 5. We also show the effectiveness of our work in a real application in Chapter 6. A review of the related work is described in Chapter 7. Finally, we conclude in Chapter 8 and present the potential extensions of our work to different applications and expected future HPC systems in Chapter 9.

Chapter 2

Background

We provide an introduction to stencil computations in this chapter. We also describe important memory and energy properties of contemporary processors. Cache blocking techniques are essential to improve the performance of stencil computations in contemporary processors, so we describe some very efficient cache blocking techniques in detail. To understand the efficiency of the achieved performance, given the hardware resources limits, we review two important performance modeling techniques, the Roofline and the Execution-Cache-Memory (ECM) models.

2.1 Stencil computations

Regular stencil computations arise as kernels in structured grid finite-difference, finite-volume, and finite-element discretizations of partial differential equation conservation laws and constitute the principal innermost kernel in many temporally explicit schemes for such problems. They also arise as a co-principal innermost kernel of Krylov solvers for temporally implicit schemes on regular grids. In [2], they constitute the fifth of the "seven dwarfs," the class of floating point kernels that receive the greatest attention in high performance computing.

In iterative stencil computations of explicit or implicit type, each point in a multi-dimensional spatial grid is updated using weighted contributions from neighboring points, whose locations and weights define the stencil operator. Depending upon the application, the weights can be constant or variable in space and/or time with some or no symmetry to be exploited around the updated point.

A major demarcation exists between stencils whose coefficients are constant in space and time and those that vary, since variable coefficients can shift the dominant working set from the lattice values being updated to the coefficients themselves. In the PDE context, coefficient variability can arise from constitutive parameters (conductivities, elastic moduli, etc.) that depend upon space or time intrinsically or through dependencies on the evolving field values, themselves, the typical nonlinear case. Some models can be scaled so that the variability affects only the diagonal term of the stencil, which is then an important case to which to specialize. Whether to compute coefficients on-the-fly is a decision that affects the code balance, since it both releases all of the memory bandwidth to the lattice values and increases the flop intensity of the typical lattice update. For this reason, we incorporate examples of both constant and variable coefficient operators into our models and experiments.

Another decisive property of stencil operators is their local spatial extent, which derives from the truncation error order of the finite discretization scheme. In contemporary applications that drive fast stencil evaluations, such as seismic imaging, eighth order is used in industrial applications of which we are aware [3]. This contrasts with the second-order schemes for which the greatest amount of performance-oriented research has been done to date.

Another property of stencil operators with fundamental impact on achievable efficiency is the spatial dimension. Our contribution concentrates on the most common case of three dimensions, though our illustrations of concepts often retreat into one or two dimensions, so that space and time fronts can be visualized on planar figures. In principle, each spatial dimension may be treated in the same or in a different way with respect to partitioning and participation in wavefronts, as we show in our work and the literature.

Our work includes stencil computation kernels that use a “Jacobi-like” update scheme where the stencil array (i.e., the data structure with read access to neighboring grid points) is not written to during the same sweep. Another possible variant is “Gauss-Seidel” update scheme, where the stencil array is adapted during the same sweep. The latter is also relevant in practice but saved for an expanded scope of work. All stencils considered here are “star stencils” of various spatial differential or truncation orders, where the stencil operator extends along one dimension at a time, without diagonal offsets. “Box stencils”, “diamond stencils”, and multicomponent stencils, in which multiple discrete fields interact on the same (or interlaced, staggered) lattices, are also important in practice, and can be handled with similar techniques.

The aforementioned issues put once-humble stencil evaluation in the cross-hairs of co-design and motivate our examination of several shared-memory (multi-core) and distributed-memory (message-passing) optimizations of a variety of stencils (see Code Listings 3.1, 3.2, 3.3, and 3.4, in Chapter 3) on state-of-the-art hardware. We are especially concerned with the degradation of memory bandwidth per core forecast at tomorrow’s extreme scale. We test our techniques on a range of star-like stencils accommodating up to eighth-order, constant and variable coefficient (without on-the-fly recomputation), noting their salutary effects on memory pressure, power consumption, and obtainable performance, and noting the transition of hardware bottlenecks.

For the fundamental kernel of Cartesian lattice updates, this thesis merely scratches the surface of co-design. Pipelined or s-step Krylov solvers, time parallelism, and high-order temporal discretizations that are obtained by using the governing PDE to estimate high time derivatives from high space derivatives are all potentially stencil-expanding (and halo-expanding) decisions made at an upstream algorithmic stage. Their downstream consequences on stencil update performance and code balance can be examined using the analyses and software tools introduced herein, but it remains to close the loop with analyses and tools that allow how to design the best discrete schemes in the first place.

2.2 Contemporary computer processors

Computer processors usually have slow and large memory (main memory), where most or all of the application data lives, together with fast and small memory (cache memory) near the CPU to bridge the performance gap between the CPU and main memory. In the past decade, the frequency power wall forced processor designers to increase processor concurrency. More compute power is obtained by adding more CPUs instead of increasing the clock frequency of the processors. Modern processors usually have multiple levels of cache memory, with a trend of having private cache levels close to the CPUs and shared cache levels among groups of them.

One of the major challenges facing the applications is the increasing performance gap between the CPU and the memory bandwidth, as shown in Table 2.1. Stencil computations are among the important kernels that suffer from this issue. Naïve implementations of stencil computations leave the CPU idle a significant amount of time while waiting for data transfer with the main memory to complete.

The cache memory levels utilize temporal and spatial locality principles to bridge the performance gap between the memory and the CPU. Temporal locality relies on the fact that when a memory location is accessed, it is likely to be accessed again in the near future. That is, keeping the data in the cache memory makes consecutive accesses to it faster. Spatial locality relies on the fact that when a memory location is accessed, its neighbor is likely to be accessed in the near future. When the CPU request data from the main memory, its neighbor data is brought with it, to reduce the data transfer latency penalty. The cache memory size plays a significant role

Processor [cores]	Cache			Mem. GB/s [/core]		Gflops/	Flops/
	L3	L2	L1	Theory	STREAM	Sec	Byte
E5-2680 [8]	20M	256k	32k	51.2	38 [4.8]	173	4.6
E5-2695v2 [12]	30M	256k	32k	59.7	51 [4.3]	230	4.5
E5-2699v3 [18]	45M	256k	32k	68	47 [2.6]	662	14.1
IBM BG/Q [16]	-	32M	16k	42.6	29 [1.8]	205	7.1
AMD 6380 [16]	16M	1M	16k	51.2	31.6 [2]	160	5.1
KNC SE10P [61]	-	512k*61	32k	352	162 [2.7]	1060	6.5
KNL (DRAM)	-	2M/2-core	-	100	-	3000+	30+
KNL (MCDRAM)	-	2M/2-core	-	500	-	3000+	6+

Table 2.1: Contemporary processor characteristics, showing the trend of memory bandwidth decreasing per core.

in maintaining high data reuse near the processor to bridge the increasing gap with main memory bandwidth. Unfortunately, contemporary computer processors exhibit a trend of having fixed or smaller cache size per CPU, as shown in Table 2.1.

The latency of bringing the data from main memory to the CPU is another challenging factor to obtaining high performance. Hardware and software prefetching techniques alleviate this latency issue. Regular or contiguous memory access patterns are important to utilize the hardware prefetching units. Moreover, the hardware prefetching units of modern Intel processors stream the data within the boundaries of 4KiB Translation Lookaside Buffer (TLB) pages, making long contiguous memory accesses more important to utilize them.

In summary, careful optimization techniques are required to overcome the memory bandwidth and cache size limitations. This can be achieved through algorithms with higher data reuse in the cache memory, while maximizing the regular data access patterns for proper utilization of the prefetching units.

2.3 Cache blocking

Typical scientific applications use larger grid size, on the order of GiBs, than a processor's cache memory, which is on the order of MiBs. In the "naïve" approach, the grid cells are updated in lexicographic order. Each grid cell update involves loading the neighbor grid cells to perform the stencil computation. This neighbor access results in loading the same data multiple times from main memory in each iteration, with no data reuse across iterations. As a result, "naïve" stencil computations are typically memory-bound due to their low flops/byte ratio [4].

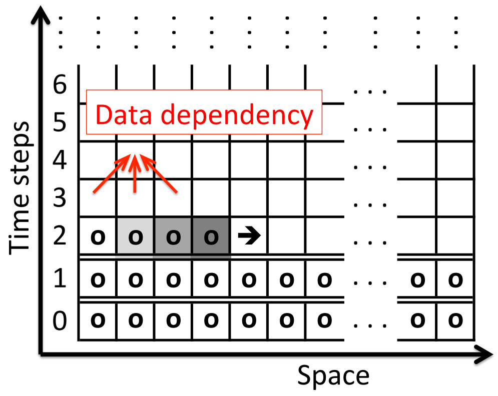
Spatial blocking aims to maximize the in-cache data reuse within the same iteration of the grid sweep by changing the grid point update order. We describe an efficient spatial blocking strategy in details in Section 3.2.1. Temporal blocking alleviates the memory pressure by allowing even more in-cache data reuse, where several time step updates are performed to a grid point before evicting the data to main memory. We describe very efficient temporal blocking techniques from the literature, namely, wavefront and diamond blocking.

2.3.1 Wavefront temporal blocking

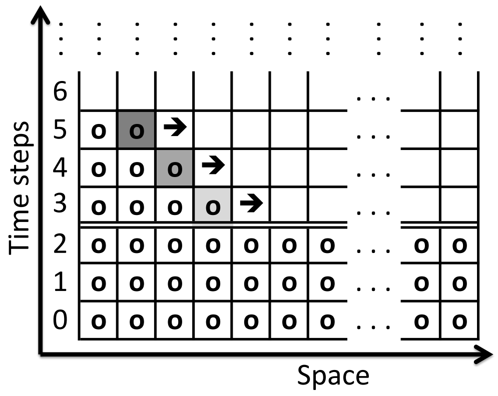
We use the naïve update order of a 3-point stencil in one dimension to illustrate the algorithms described here, using the C language syntax:

```
for(t=0; t<T; t++){
  for(x=1; x<Nx-1; x++){
    A[(t+1)%2][x] = W1 * A[t%2][x] + W2 * (A[t%2][x-1] + A[t%2][x+1])
}}
```

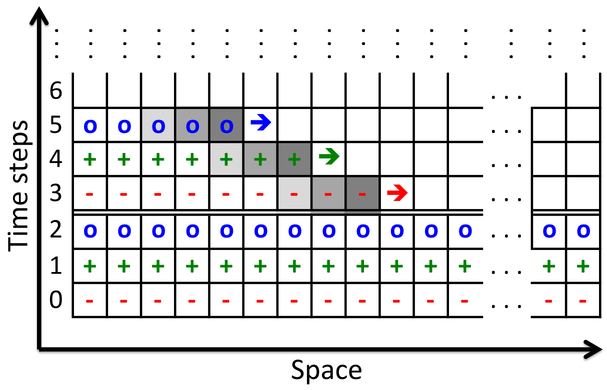
Figure 2.1a shows the naïve update order of this stencil. The fading gray color represents recently updated grid points, with the darkest assigned to the most recent update. The three "upward pointing" red arrows show the data dependency of each



(a) naïve: Lattice sites updated in chronological order, one time step at a time. Data dependencies example represented by the three "upward pointing" red arrows.



(b) Single thread wavefront traversal in space-time blocks. Wavefront tiles are updated sequentially.



(c) Multi-thread wavefront traversal in space-time blocks. The symbols/colors in the cells represent the update of different threads. Each thread updates one time step of the wavefront tile in this example. Extra spacing between the threads (i.e., steeper tile slope) allows concurrent update of the wavefront tile.

Figure 2.1: Different stencil update approaches for the 3-point stencil in one dimension. Fading gray boxes represent the last three updates.

grid point, which is important to consider at temporal blocking optimizations.

Wavefront temporal blocking is a well-known technique in the literature [5, 6, 7]. Compared to the naïve approach, the grid point update order maximizes the reuse of the most recently visited grid points while respecting the data dependencies. Figure 2.1b shows the basic idea of wavefront temporal blocking for the 3-point stencil in one dimension. Wavefront tiles traverse the space-time block in the direction of the arrows. The slope S of the wavefront tile is determined by the radius of the stencil operator R , where high-order (i.e., long-range) stencils require a smaller tile slope to respect the data dependency ($S = -1/R$). For example, the 3-point stencil, with stencil bandwidth unity, has $S = -1$. The data in the wavefront tile has to fit in the cache memory (along with the surrounding grid points accessed by the stencil operator) to achieve the desired cache memory data reuse in the wavefront approach. The time dimension block size controls the wavefront tile size to fit in the desired cache level. For example, three time steps are blocked in Figure 2.1b to illustrate the concept.

On multi-core systems one can perform single-thread wavefronts on separate spatial locations in the grid. There is no need (nor use) for any cache sharing among the threads, even if the hardware provides it. These concepts are described and implemented in [7, 8]. An explicitly multi-core aware wavefront scheme leveraging the shared cache among the threads was introduced in [9]. The wavefront tile update is pipelined over a group of threads sharing a cache memory level. This has the advantage of reducing the cache memory size requirements and allowing the use of larger cache blocks. As a result, the memory bandwidth pressure can be reduced compared to the single-thread wavefront. Fig. 2.1c shows the multi-thread wavefront variant proposed in [9], where each thread's update is assigned different symbol/color. Each thread is assigned to one or more consecutive time steps of the wavefront tile. To enable concurrent update in the wavefront tile, the slope of the wavefront is increased to add spacing between the threads. For example, the additional cell spacing in Fig. 2.1c allows the three threads to update the wavefront tile concurrently. All threads have the same amount of work for load balancing, and they must be synchronized after each time step update to ensure correctness. A global barrier is the simplest solution for this, but a relaxed synchronization scheme may result in better performance if the workload per thread is small [10].

Combining the wavefront blocking scheme with other tiling techniques is essential in three-dimensional grids. Otherwise, using only the wavefront scheme in multiple dimensions would require much larger cache block size than typical cache sizes, when a reasonable grid size is used. Wavefront tiling is commonly combined with tiling approaches such as trapezoidal, parallelogram, or diamond tiling. These tiling approaches limit the tile size in other dimensions, such that the wavefront tile size fits in the desired cache memory. Since diamond tiling is proven to provide maximum the data reuse of a loaded spatial cache block [11], we describe it in greater detail in the next section.

Wavefront blocking is ideal when the block size in time is fixed by a different factor, for example, by the tiling approach in a different spatial dimension. By pipelining the data to the cache memory, the wavefront method loads each element once to the cache memory and performs the maximum time updates before evicting the results from the cache memory. On the other hand, loading data blocks separately and updating their elements by other tiling techniques would not allow maximum data reuse at the edges of the loaded block due to the data dependencies.

2.3.2 Diamond tiling

Diamond tiling has received much attention in recent years. Figure 2.2 shows the basic idea for the one-dimensional 3-point stencil. Arrows represent the data dependency across the diamond tiles. Diamond tiles that start at the same time step compose a

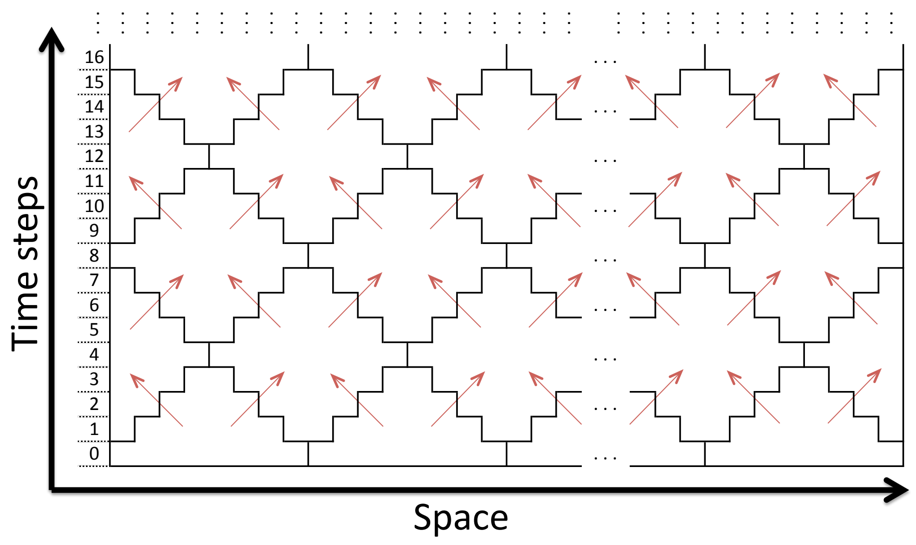


Figure 2.2: Diamond tiling on a one-dimensional space grid, with arrows representing inter-tile data dependencies. The number of diamond tiles per row represents the maximum attainable concurrency (i.e., concurrency limit), as the tiles in the row can be executed independently of each other.

"row of diamonds", in which the diamond tiles are independent of each other, hence can be updated concurrently. Each interior diamond tile has a data dependency on the two diamond tiles that share edges with it in the lower row of diamonds ("parents"). Sub-diamond tiles at the boundaries of the spatial domain have only one parent. The slope of the tile edges depends on the stencil radius, where $S = \pm 1/R$. In this work we consider diamond tile updates that do not perform synchronization with other tiles until the update is completed. Several advantages of diamond tiling make it favorable in shared memory systems: it maximizes the data reuse of the loaded data block [11], has low synchronization requirements, allows concurrent start-up in updating the diamond tiles, maintains high concurrency in transient state, and uses a unified tile shape, which simplifies the implementation.

2.4 Analytic and phenomenological performance modeling

Analytic performance models can answer the question what “good enough” performance means, in the sense that they predict the optimal performance of an algorithm and/or an implementation in view of the available resources. The Roofline model is a well-known example, whose principles date back into the 1980s [12] and which has received revived interest in the context of cache-based multi-core processor architectures in recent years [13]. It predicts the performance of “steady-state” loops or loop nests on a CPU, assuming that one of two possible bottlenecks apply: either the runtime is limited by the execution of instructions (execution bottleneck) or by the required data transfers through the memory hierarchy (data bottleneck), whichever takes longer. “Steady state” in this context means that start-up and wind-down effects are ignored, and that the CPU executes the same mix of instructions with the same data requirements for a long time. The assumptions behind the Roofline model are fulfilled for many algorithms in computational science that are either very data-bound or very compute-bound, i.e., whenever the data transfer and execution times differ strongly. Unfortunately, stencil algorithms with temporal blocking optimizations do not fall in this category, as we show in our results.

The ECM Model [14, 15] extends the Roofline model. In contrast to the latter, the ECM model does not assume perfect overlap between in-core and data transfer times. Instead, it assumes that the in-core execution time, i.e., the time required to execute a number of loop iterations with data coming from the L1 cache, is composed of an overlapping and a non-overlapping part. The CPU serializes the non-overlapping part with all transfer times between adjacent memory hierarchy levels down to where the data originally resided. We briefly review the model here. See [15] for more details.

The execution unit (pipeline) sets the in-core execution time T_{core} by taking the largest number of cycles to execute the instructions for a given number of loop iterations. The non-overlapping part of T_{core}, called T_{nOL}, consists of all cycles in which a LOAD instruction retires. As all data transfers are in units of Cache Lines (CLs), we usually consider one "cache line's worth of work." With a double-precision stencil code, one unit of work is 8 iterations. The time for all data transfers required to execute the work unit is the "transfer time." The ECM model neglects the latency altogether, so the maximum bandwidth (cache lines per cycle) between adjacent memory hierarchy levels determines the cost for one CL transfer. For example, on the Intel Haswell architecture, one CL transfer takes one cycle between the L1 and L2 caches and two cycles between L2 and L3. Moving a 64-byte CL between memory and L3 takes $64\,\text{bytes} \cdot f/b_{\text{S}}$ cycles. Here f is the clock frequency of the CPU and b_{S} is the fully saturated memory bandwidth.

If T_{data} is the transfer time, T_{OL} is the overlapping part of the core execution, and T_{nOL} is the non-overlapping part, then

$$T_{\text{core}} = \max\left(T_{\text{nOL}}, T_{\text{OL}}\right) \quad \text{and} \quad T_{\text{ECM}} = \max(T_{\text{nOL}} + T_{\text{data}}, T_{\text{OL}}) \ . \tag{2.1}$$

T_{data} is the sum of the data transfer times through the memory hierarchy. If, for example, the data is in the L3 cache, $T_{\text{data}} = T_{\text{L1L2}} + T_{\text{L2L3}}$.

We use a shorthand notation for the cycle times in the model for executing a unit of work: $\{T_{\text{OL}} \,\|\, T_{\text{nOL}} \,|\, T_{\text{L1L2}} \,|\, T_{\text{L2L3}} \,|\, T_{\text{L3Mem}}\}$. Adding up the contributions from T_{data} and T_{nOL} and applying (2.1) one can predict the cycles for executing the loop with data from any given memory level. For example, if the model is $\{4 \,\|\, 4 \,|\, 2 \,|\, 4 \,|\, 9\}$ cy, the prediction for L3 cache will be $\max\left(4, 4+2+4\right)$ cy $= 10\,$cy. For predictions, we use "$\rceil$" as the separator to present the information in a similar way as in the model. In the example above this would be $T_{\text{ECM}} = \{4 \rceil 6 \rceil 10 \rceil 19\}$ cy. It is easy to get from time to performance (work divided by time) by calculating the fraction $P = W/T_{\text{ECM}}$, where W is the work. If T_{ECM} is given in clock cycles but the unit of work is a LUP and the performance metric is LUP/s then we have to multiply by the clock speed.

For multi-core scalability, we assume that the performance is linear in the number of cores until the maximum bandwidth of a bottleneck data path is exhausted. On Intel processors the memory bandwidth is the only bottleneck. Hence, the absolute maximum performance is the Roofline prediction for memory-bound execution: $P_{\mathrm{BW}} = I \cdot b_{\mathrm{S}}$, with I being the computational intensity. Thus, the model allows us to predict the number of cores required to reach bandwidth saturation.

It is possible in many situations to analyze the data transfer properties of a stencil code and construct the ECM model from first principles. In practice, one faces difficulties with this approach when looking at temporally blocked codes. The reason is that the model requires an accurate analysis of the data transfer volume in different cache levels, which becomes very difficult with small, non-rectangular block shapes (for example, diamonds). In such situations, the model can only give very rough upper limits. However, it is still possible to apply the principles of the model in a phenomenological way by *measuring* the data transfers by hardware performance monitoring. If the constructed model agrees with the performance measurements, it indicates that the code utilizes the hardware in an optimal way, and especially that the low-level machine code produced by the compiler does not incur any significant overhead. This phenomenological modeling approach is applied later when analyzing the performance of our temporal blocking approach.

Chapter 3

Motivation: On spatial and temporal blocking performance limits

In this chapter we motivate our cache block sharing work by showing the performance and resource requirements limits of the state-of-the-art spatial and temporal blocking schemes. We use the roofline model for the spatially blocked codes and we derive models for highly efficient temporal blocking schemes. These models are validated using the corner-case stencils shown in Code Listings 3.1, 3.2, 3.3, and 3.4.

3.1 Test systems: Intel Ivy Bridge and Haswell

Our experiments are performed using Intel 10-core Ivy Bridge (Xeon E5-2660v2) and Intel 18-core Haswell (E5-2699v3) processors. We use a cluster of dual-socket Intel Ivy Bridge (Xeon E5-2660v2) nodes with a nominal clock speed of 2.2 GHz. The "Turbo Mode" feature was disabled. Each CPU has a 25 MiB L3 cache which is shared among all cores, and core-private L2 and L1 caches of 256 KiB and 32 KiB, respectively. All data paths between the cache levels are half-duplex, 256-bit wide buses, so the transfer of one 64-byte cache line between adjacent caches takes two CPU cycles. The core architecture supports all Intel Single Instruction Multiple Data (SIMD) instruction sets up to AVX (Advanced Vector Extensions). With AVX, one core is able to sustain one full-width (32 byte) load and one half-width (16 byte) store per cycle. In addition, one AVX multiply and one AVX add instruction can be executed per cycle. Since one AVX register can hold either four double precision (DP) or eight single precision (SP) operands, the peak performance of one core is eight flops per cycle in DP or sixteen flops per cycle in SP.

Each Ivy Bridge node is equipped with 64 GB of DDR3-1600 RAM per socket (using four memory modules) and has a maximum attainable memory bandwidth of $b_S \approx 40\,\mathrm{GB/s}$ per socket (as measured with the STREAM COPY [16, 17] benchmark). The nodes are connected by a full non-blocking, fat-tree QDR InfiniBand network.

We use a node of dual-socket Intel Haswell (Xeon E5-2660v2) processors. The Haswell processor has several improvements over the Ivy Bridge (Sandy Bridge) microarchitecture. Mainly, the Haswell's CPU uses AVX2 instructions that include Fused Multiply-Add (FMA) operations, double the data transfer bandwidth of the CPU load/store and L1-L2 caches. We use a Haswell processor with 45 MiB L3 cache, 128 GB DDR4-2133 RAM per socket, a maximum attainable memory bandwidth of $b_S \approx 47.5\,\mathrm{GB/s}$, and a nominal clock speed of 2.3 GHz.

For compiling and linking in this chapter, we use the Intel C compiler 13 for the spatial blocking experiments and Intel C compiler 15 for the temporal blocking experiments. Hardware performance counter measurements were done with `likwid-perfctr` from the LIKWID multicore tools collection [18].

Apart from standard metrics, `likwid-perfctr` can also read the power dissipation and energy consumption estimates based on the Running Average Power Level (RAPL) mechanism. RAPL is an energy model implemented in hardware with high degree of accuracy [19]. Its technology allows to estimate energy consumption by using hardware counter technology available on Intel Sandy/Ivy Bridge lines of

Listing 3.1: 1st-order-in-time 7-point constant-coefficient isotropic stencil in three dimensions, with symmetry. The code shows single time iteration, where this code is repeated many times in the time loop, with arrays pointer swapping after each iteration.

```
for(int k=1; k < N-1; k++) {
 for(int j=1; j < N-1; j++) {
  for(int i=1; i < N-1; i++) {
   U[k][j][i] = c0 *  V[ k ][ j ][ i ]
              + c1 * (V[ k ][ j ][i+1] + V[ k ][ j ][i-1])
              + c1 * (V[ k ][j+1][ i ] + V[ k ][j-1][ i ])
              + c1 * (V[k+1][ j ][ i ] + V[k-1][ j ][ i ]);
}}}
```

multicore processors. On the system used for the tests, RAPL is able to report CPU energy separately from DRAM energy. To reduce the temperature impact on the energy estimates, we run long enough experiments to have a steady state temperature and energy consumption. We assume that RAPL reports consistent estimates of the energy values, where we compare the relative energy consumption of the examined cases, rather than measuring absolute energy consumption values.

3.2 Performance prediction and evaluation for pure spatial blocking

We use Lattice-site Update (LUP) as a basic performance metric, since it does not contain any uncertainty as to how many flops are actually done during one stencil update. Specific implementations have a fixed ratio of LUPs to flops and other relevant hardware events (such as bytes transferred, instructions executed, etc.), which are discussed as required. Unless otherwise noted, the working set does not fit into any CPU cache.

In the following we describe in detail two "corner cases" of stencil update schemes: a three-dimensional seven-point stencil with constant coefficients (Jacobi-type smoother, see listing 3.1) and a three-dimensional 25-point stencil with constant coefficients

Listing 3.2: 1^{st}-order-in-time 7-point variable-coefficient stencil in three dimensions, with no coefficient symmetry. The code shows single time iteration, where this code is repeated many times in the time loop, with arrays pointer swapping after each iteration.

```
for(int k=1; k < N-1; k++) {
 for(int j=1; j < N-1; j++) {
  for(int i=1; i < N-1; i++) {
   U[k][j][i] = C0[k][j][i] * V[ k ][ j ][ i ]
              + C1[k][j][i] * V[ k ][ j ][i+1]
              + C2[k][j][i] * V[ k ][ j ][i-1]
              + C3[k][j][i] * V[ k ][j+1][ i ]
              + C4[k][j][i] * V[ k ][j-1][ i ]
              + C5[k][j][i] * V[k+1][ j ][ i ]
              + C6[k][j][i] * V[k-1][ j ][ i ];
}}}
```

(see listing 3.3). These examples were picked because they are simple to model for memory-bound situations. Later we will show the effectiveness of temporal blocking for the Jacobi smoother, the variable-coefficient 7-point stencil shown in listing 3.2, and the variable-coefficient 25-point stencil shown in listing 3.4.

We present spatial blocking results for the Ivy Bridge processor. It has better machine balance (higher) than the Haswell processor, so the memory bandwidth saturation here would be less severe compared to the Haswell processor.

3.2.1 3D 7-point stencil with constant coefficients

The standard two-grid three-dimensional “Jacobi” update scheme in listing 3.1 is probably the best analyzed stencil algorithm to date. From a data flow perspective the spatial loop nest reads one array and updates another. In double precision, the minimum code balance is thus $B_\mathrm{C} = 24\,\mathrm{bytes/LUP}$: eight bytes for loading one new element of the previous time step data, eight bytes for the write-allocate transfer on the new time step, and eight bytes for evicting the updated data back to memory. The write-allocate transfer may be avoided by the use of “non-temporal stores,” which bypass the memory hierarchy, thereby reducing the code balance to 16 bytes/LUP.

Listing 3.3: 2^{nd} order in time 25-point constant-coefficient isotropic stencil in three dimensions, with symmetry across each axis. The code shows single time iteration, where this code is repeated many times in the time loop, with arrays pointer swapping after each iteration.

```
for(int k=4; k < N-4; k++) {
 for(int j=4; j < N-4; j++) {
  for(int i=4; i < N-4; i++) {
   U[k][j][i] = 2*V[k][j][i] - U[k][j][i] + C[k][j][i] * [
               +c0 *  V[ k ][ j ][i  ]
               +c1 * (V[ k ][ j ][i+1]+V[ k ][ j ][i-1]
                     +V[ k ][j+1][ i ]+V[ k ][j-1][ i ]
                     +V[k+1][ j ][ i ]+V[k-1][ j ][ i ])
               +c2 * (V[ k ][ j ][i+2]+V[ k ][ j ][i-2]
                     +V[ k ][j+2][ i ]+V[ k ][j-2][ i ]
                     +V[k+2][ j ][ i ]+V[k-2][ j ][ i ])
               +c3 * (V[ k ][ j ][i+3]+V[ k ][ j ][i-3]
                     +V[ k ][j+3][ i ]+V[ k ][j-3][ i ]
                     +V[k+3][ j ][ i ]+V[k-3][ j ][ i ])
               +c4 * (V[ k ][ j ][i+4]+V[ k ][ j ][i-4]
                     +V[ k ][j+4][ i ]+V[ k ][j-4][ i ]
                     +V[k+4][ j ][ i ]+V[k-4][ j ][ i ])];
}}}
```

Our experiments show that non-temporal stores can improve the performance of the 7-point constant-coefficient stencil by 33%. On the other hand, the non-temporal stores decrease the performance of the 25-point constant-coefficient stencil, as it reuses the updated grid points in the same iteration. The variable-coefficient stencils may have minor benefit from the non-temporal stores, as the contribution of the write-allocate transfers is minor compared to the loaded coefficient data.

Depending on the grid size and the cache size, spatial blocking may be required to achieve the minimum code balance of 24 bytes/LUP. If three successive "layers" of size $N_x \times N_y$ grid points fit into a cache, the only load operation within a LUP that causes a cache miss goes to `V[k+1][j][i]`, and all other loads can be satisfied from the cache. If C is the cache size, we assume (as a rule of thumb) that only about $C/2$ is available for the previous time step data, and the layer condition for double precision is

$$3 \times N_x \times N_y \times 8\,\text{bytes} < \frac{C}{2n_{\text{threads}}} \; . \tag{3.1}$$

Listing 3.4: 1st-order-in-time 25-point variable-coefficient anisotropic stencil in three dimensions, with symmetry across each axis. The code shows single time iteration, where this code is repeated many times in the time loop, with arrays pointer swapping after each iteration.

```
for(int k=4; k < N-4; k++) {
 for(int j=4; j < N-4; j++) {
  for(int i=4; i < N-4; i++) {
   U[k][j][i] = C00[k][j][i]*  V[ k ][ j ][ i ]
                +C01[k][j][i]*( V[ k ][ j ][i+1]+V[ k ][ j ][i-1])
                +C02[k][j][i]*( V[ k ][j+1][ i ]+V[ k ][j-1][ i ])
                +C03[k][j][i]*( V[k+1][ j ][ i ]+V[k-1][ j ][ i ])
                +C04[k][j][i]*( V[ k ][ j ][i+2]+V[ k ][ j ][i-2])
                +C05[k][j][i]*( V[ k ][j+2][ i ]+V[ k ][j-2][ i ])
                +C06[k][j][i]*( V[k+2][ j ][ i ]+V[k-2][ j ][ i ])
                +C07[k][j][i]*( V[ k ][ j ][i+3]+V[ k ][ j ][i-3])
                +C08[k][j][i]*( V[ k ][j+3][ i ]+V[ k ][j-3][ i ])
                +C09[k][j][i]*( V[k+3][ j ][ i ]+V[k-3][ j ][ i ])
                +C10[k][j][i]*( V[ k ][ j ][i+4]+V[ k ][ j ][i-4])
                +C11[k][j][i]*( V[ k ][j+4][ i ]+V[ k ][j-4][ i ])
                +C12[k][j][i]*( V[k+4][ j ][ i ]+V[k-4][ j ][ i ]);
}}}
```

This assumes that OpenMP parallelization is done along the z axis with static scheduling. If this condition is violated, at least the loads to `V[k+1][j][i]`, `V[k-1][j][i]`, and `V[k][j+1][i]` will cause cache misses, which leads to a code balance of 40 bytes/LUP. If the cache is too small to even hold three successive rows of the grid, the only loads that come from the cache will be to `V[k][j][i-1]` and to `V[k][j][i]`. The code balance for this case is 56 bytes/LUP.

The layer condition (3.1) is independent of N_z. Hence, it is sufficient to introduce spatial blocking in the x and/or y dimensions in order to arrive at the minimum code balance. In practice one should try to keep the inner (x) block size larger than about one OS page in order to avoid frequent TLB misses and excess data traffic due to hardware prefetching [20]. Additionally, we use "static,1" OpenMP scheduling, which relaxes the layer condition to

$$(n_{\text{threads}} + 2) \times N_x \times N_y \times 8\,\text{bytes} < \frac{C}{2}\ , \tag{3.2}$$

since each thread shares both neighboring layers of its current z layer with its neighboring threads (except the first and the last thread, which only share one layer with their respective neighbor).

Note that the layer condition can be satisfied for any cache in the hierarchy if the block sizes are chosen appropriately; for memory-bound implementations one usually tries to establish it for the Last-Level Cache (LLC) to ameliorate the impact of the memory bandwidth bottleneck. In case of temporal blocking, however, the bottleneck may not be main memory and the smaller caches need to be taken into account. Since the overhead at block boundaries becomes significant at small block sizes, the optimum code balance is a goal that is all but impossible to achieve in this case.

Figure 3.1a shows the performance of the seven-point stencil algorithm in double precision on one Ivy Bridge chip with up to ten cores for a grid of 960^3 points (circles) and the memory bandwidth usage as measured by `likwid-perfctr` (triangles), together with the estimated saturated performance (solid line) and ideal scaling (dashed line). The selected grid size is much larger than the L3 cache memory size, which is more practical in scientific applications. Any similar grid size would have similar performance since the same number of bytes would be transfered per LUP. We choose 960 because it is divisible by 4, 8, 12, 16, 20, and 24, which correspond to full diamond tile width in this work's implementation. This results in having less diamond tile fractions at the boundaries, which would cause interference in our performance modeling work.

With appropriate spatial blocking the expected saturated performance as given by the roofline model [1, 13] is

$$P_{\mathrm{roof}} = \frac{b_{\mathrm{S}}}{B_{\mathrm{C}}} = \frac{40\,\mathrm{GB/s}}{24\,\mathrm{bytes/LUP}} = 1.67\,\mathrm{GLUP/s}\ . \tag{3.3}$$

The performance saturates at 6–7 cores, and the available memory bandwidth is utilized by up to 95%. Since there is strong saturation, we expect a strong benefit from temporal blocking.

3.2.2 3D 25-point stencil with constant coefficients

The considerations about layer conditions as shown above for the seven-point stencil apply in a similar way for long-range stencils. In the particular case of the algorithm shown in listing 3.3, one sweep of the grid updates one array (read/modify/write) and reads two more arrays, one of which is accessed in a radius-four (semi-bandwidth of four) stencil pattern. The minimum code balance for double precision is thus $B_\mathrm{C} = 32\,\mathrm{bytes/LUP}$.

Due to the long-range stencil the layer condition is changed as compared to the previous case. With “static,1” scheduling, each thread can share the eight neighboring layers `V[k-4][][]...V[k-1][][]` and `V[k+1][][]...V[k+4][][]` with its eight neighboring threads (four in either z direction), but the top and bottom threads have less sharing. Consequently, the layer condition is

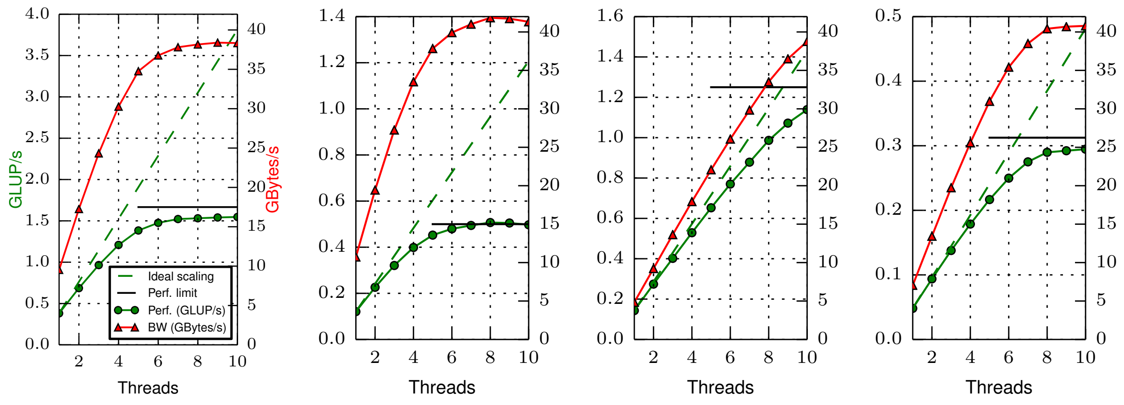
$$(n_\mathrm{threads} + 8) \times N_x \times N_y \times 8\,\mathrm{bytes} < \frac{C}{2}\ . \tag{3.4}$$

If this condition is fulfilled, `V[k+4][j][i]` is the only element from the stencil array that has to come from main memory.

Figure 3.1c shows the performance and memory bandwidth usage for ideal spatial blocking on a ten-core Ivy Bridge chip. The roofline model predicts an upper performance limit of

$$P_\mathrm{roof} = \frac{b_\mathrm{S}}{B_\mathrm{C}} = \frac{40\,\mathrm{GB/s}}{32\,\mathrm{bytes/LUP}} = 1.25\,\mathrm{GLUP/s}\ . \tag{3.5}$$

In contrast to the Jacobi-type stencil there is no clear saturation. The data transfers



(a) 7-pt const. coeff. (b) 7-pt var. coeff. (c) 25-pt const. coeff. (d) 25-pt var. coeff.

Figure 3.1: Performance scaling across the cores of a chip with purely spatial blocking and data sets larger than L3 cache for the stencil algorithms shown in listing 3.1 3.2 3.3 3.4. Problem sizes: 960^3, 680^3, 960^3, and 480^3 for subfigures a, b, c, and d, respectively. STREAM COPY memory bandwidth $b_S \approx 40\,\mathrm{GB/s}$

within the cache hierarchy and the execution of the loop code with data from the L1 cache take so much time that there is insufficient pressure on the memory interface to saturate the bandwidth even with ten cores, which makes this stencil a bad candidate for temporal blocking in the Intel Ivy Bridge processor unless there is an opportunity to save significant time with more efficient in-core execution [15]. On the other hand, since the Intel Haswell processor has a larger compute to memory bandwidth performance gap, temporal blocking techniques would be beneficial for it.

3.2.3 Other stencils

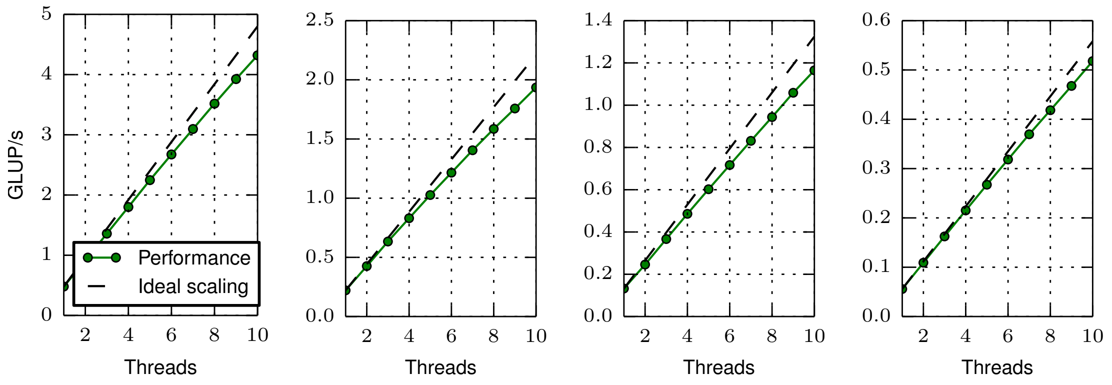
Figures 3.1b and 3.1d show the saturation characteristics and maximum performance levels for the seven-point stencil with variable coefficients (ideal code balance of 80 bytes/LUP) and the 25-point stencil with axis-symmetric variable coefficients (ideal code balance of 128 bytes/LUP) listed in Lsts. 3.2 and 3.4, respectively. Both show strong saturation close to the performance levels predicted by the roofline model, and are thus important targets for temporal blocking optimizations.

3.3 Upper performance bounds for in-cache execution

To find the expected performance of ideal temporal blocking (i.e., when performance has completely decoupled from the memory bottleneck), we have measured the performance at problems fitting completely in the last-level cache without temporal blocking. The results for the stencils discussed in the previous section are shown in Figs. 3.2a–3.2d. Problem sizes have been chosen so that work decomposition across threads is easy (no "artificial" load imbalance) and the inner loop length is not too short.

We see that all stencil algorithms scale very well across the cores, which is expected since the Ivy Bridge architecture does not have a hardware bottleneck except the main memory interface. It also shows that our implementation has no serious issues with OpenMP overhead or load balancing even with in-cache data sets.

Assuming 8 flops/LUP, the full-socket performance of the 7-point stencil with constant coefficients is only about 35 GF/s, whereas the arithmetic peak performance of the CPU is 176 GF/s. The question arises why the observed performance is just 20% of peak with an in-cache data set (this is similar for the other stencils). An in-depth analysis requires employing the Execution-Cache-Memory (ECM) model [14]. The model reveals that there is no definite bottleneck of in-cache execution; the time is spent executing instructions (the LOAD pipeline being the limiting factor in the CPU core) and transferring data through the cache hierarchy in roughly equal shares, with no overlap between them. The true benefit of an in-cache data set (equivalent to decoupling from main memory bandwidth bottleneck through temporal blocking) is not a large single-core performance but the lifting of the multi-core memory bottleneck, eliminating saturation. A thorough analysis of this effect using the ECM performance model is carried out in [15].



(a) 7-pt const. coeff. (b) 7-pt var. coeff. (c) 25-pt const. coeff. (d) 25-pt var. coeff.

Figure 3.2: Performance scaling across the cores of a chip with data sets fitting in the L3 cache for the stencil algorithms shown in listing 3.1 3.2 3.3 3.4. Problem sizes: $96 \times 96 \times 96$, $64 \times 64 \times 48$, $128 \times 64 \times 64$, and $64 \times 32 \times 32$ for subfigures a, b, c, and d, respectively. Variable-coefficient stencils require smaller grids to fit in cache.

3.4 On temporal blocking practical performance limits

In this section we study the performance of highly efficient temporal blocking schemes in the literature, while keeping the implementation practical and efficient for contemporary processors. We contribute accurate models to estimate the cache block size and the memory traffic of these schemes to show their limits. This work is particularly important to show the shortcomings of using single-thread per tile in contemporary processors, even when best tiling techniques are used.

3.4.1 Single-thread wavefront diamond blocking

In Sections 2.3.1 and 2.3.2 we showed that wavefront blocking and diamond tiling techniques maximize the data reuse in the cache memory. We find it reasonable to use these techniques over the outer dimensions (y and z) in three-dimensional grids. We prefer to leave the x dimension intact, at reasonable grid sizes, for efficient hardware data prefetching, minimum TLB misses, and longer strides for vectorization. In fact,

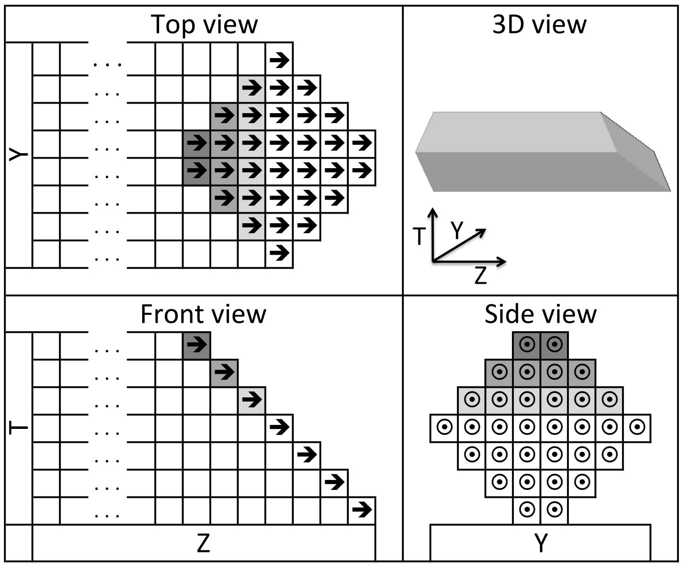


Figure 3.3: Diamond tiling (along the y-axis) with single-thread wavefront temporal blocking (along the z-axis) in a three-dimensional space grid using wavefront width of one cell.]

recent works are adopting these techniques in their implementations. Strzodka *et al.* [7] perform the diamond tiling along the y-axis and the wavefront blocking along the z-axis. Bandisti *et al.* [21] perform the diamond tiling along the z-axis and the wavefront blocking along the y-axis in PLUTO framework. As will be shown in Chapter 5, the auto-tuning of PLUTO tiling parameters shows that longer strides along the x-axis achieves the best performance, which validates our argument.

We implement a 1WD scheme in this work, as shown in Figure 3.3. The wavefront traverses along the z-axis. The diamond tiling is performed along the y-axis. Hence, each grid point in Fig. 3.3 extends along the full x range.

The 1WD scheme is an important ingredient of this work. We construct and validate cache block size and memory traffic models in the following subsections for the 1WD implementation to show its requirements and limits on contemporary processors.

3.4.2 Cache block size model

We construct a cache block size model of 1WD, validate its correctness, and study its impact on the code balance at different diamond sizes. The model calculations require four parameters: the diamond width D_w in the y axis, the wavefront tile width N_F, the bytes number in the leading dimension N_{xb}, the stencil radius R, and the number of domain-sized streams in the stencil operator, N_D. Examples of stencil radius are $R = 1$ and $R = 4$ at the 7- and 25-point stencils, respectively. The 7-point constant-coefficient stencil has $N_D = 2$ (Jacobi-like update). The 7-point variable-coefficient stencil uses seven additional domain-sized streams to hold the coefficients. For a stencil with $R = 1$, the wavefront width W_w has the size: $W_w = D_w + N_F - 2$ and the total required bytes in the wavefront cache block C_S, with some approximations, is:

$$C_S = N_{xb} \cdot \left[N_D \cdot \left(\frac{D_w^2}{2} + D_w \cdot (N_F - 1) \right) + 2 \cdot (D_w + W_w) \right] \ . \tag{3.6}$$

The equation is composed of three parts: "N_{xb}" factor is the size of the leading dimension tile size, "$D_w^2/2 + D_w \cdot (N_F - 1)$" term is the diamond area in the y-z plane as shown in the top view of Fig. 3.3, and the halo region of the wavefront is the "$2 \cdot (D_w + W_w)$" term.

For example, $D_w = 8$ and $N_F = 1$ in Fig. 3.3, so $W_w = 8 + 1 - 2 = 7$ and the total block size at 7-point constant-coefficient stencil is $N_{xb} \cdot (2 \cdot (8^2/2 + 8 \cdot 0) + 2 \cdot (8 + 7)) = 94 \cdot N_{xb}$ bytes.

The steeper wavefront in higher-order stencils results in different wavefront width ($W_w = D_w - 2 \cdot R + N_F$) and different C_S as follows:

$$C_S = N_{xb} \cdot \left[N_D \cdot D_w \cdot \left(\frac{D_w}{2} - R + N_F \right) + 2R(D_w + W_w) \right] \ . \tag{3.7}$$

It is worth mentioning that each thread requires a dedicated C_S in the blocked cache level. For example, using a 16-core Intel Haswell socket requires fitting $16 \cdot C_S$ bytes in the L3 cache memory.

3.4.3 Memory traffic model

In order to validate the effectiveness of the bandwidth pressure reduction on the memory interface, we set up a model to estimate the code balance for the temporally blocked case. If the wavefront fits completely in the L3 cache, each grid point is loaded once from main memory and is stored once after updating it during the extruded diamond update. In this case, the amount of data transfers during the extruded diamond update consists of $(2D_w - 2)$ data writes plus $(N_D \cdot D_w + 2)$ data reads, all multiplied by N_z. The number of total LUPs performed through the diamond volume is: $N_z \cdot D_w^2/2$. The code balance at double precision of a stencil with $R = 1$ is thus:

$$B_{\mathrm{C}} = \frac{16 \cdot [(2D_w - 2) + (N_D \cdot D_w + 2)]}{D_w^2} \frac{\text{bytes}}{\text{LUP}} . \tag{3.8}$$

When $R > 1$ the amount of data transfers becomes $N_z \cdot [(2D_w - 2R) + (N_D \cdot D_w + 2R)]$ and the extruded diamond volume becomes $N_z \cdot D_w^2/(2 \cdot R)$. In total, the equation becomes:

$$B_{\mathrm{C}} = \frac{16R \cdot [(2D_w - 2R) + (N_D \cdot D_w + 2R)]}{D_w^2} \frac{\text{bytes}}{\text{LUP}} . \tag{3.9}$$

3.4.4 Model verification

In this section, we verify the correctness of our memory traffic and cache block size models of the 1WD scheme. Our models and measurements prove the limitation of using separate cache block per thread, which is common in the literature. The desired code balance to decouple from the main memory bandwidth requires larger cache block size than the available cache memory in contemporary processors when separate cache block is used per thread.

We use the four stencils described in Lsts. 3.1, 3.2, 3.3, and 3.4. The grid sizes are larger than the cache memory size and fit in the main memory of the processor, which is typical in real applications. We minimize the cache block size in our experiments by using a unity wavefront tile width ($N_F = 1$), where larger N_F would increase the cache block size without decreasing the code balance. We perform our experiments using a single thread in the 18-core Haswell processor to dedicate the 45 MiB cache memory for its use. The single thread experiment allows us to test larger cache block sizes without having cache capacity misses.

Figure 3.4 shows the cache block size vs. the code balance at various diamond tile sizes (top x-axis). We compute the "Model" data using our cache block size model in Eq. 3.7 (bottom x-axis) and the code balance model in Eq. 3.9. The "Measured" data is the measured code balance in our experiments, which is computed by dividing the total measured memory traffic by the total updated grid points. We set the diamond tiles' widths to multiples of 4 and 16 for the 7-point and 25-point stencils, respectively. Data points at zero diamond width correspond to the spatial blocking scheme described in Sec. 3.2.

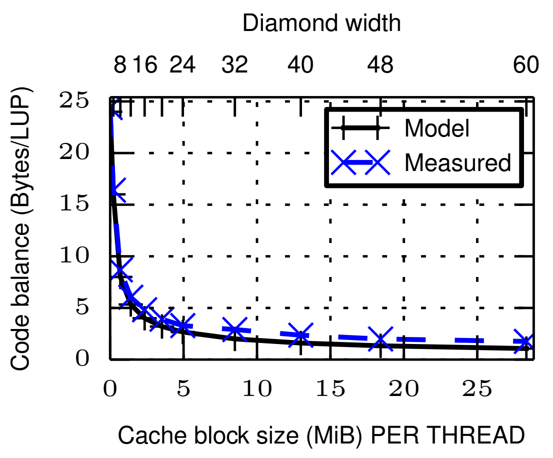
Our models are very accurate in predicting the code balance of corner-case stencil operators. There is a strong agreement between the model and the empirical results when the cache block fits in the L3 cache (below 22.5 MiB). These findings show that our implementation of the 1WD blocking scheme can achieve the theoretical memory traffic reductions. The measured code balance in Fig. 3.4 starts to deviate from the model at cache blocks larger than about half the Intel Haswell's L3 cache size (i.e., 22.5 MiB). The deviation at this point can be predicted from our cache block size model, considering the rule-of-thumb that half the cache size is usually usable for blocking [15].

The results in Fig. 3.4 prove how using separate temporally blocked tile per thread leads to starvation in the cache size requirement. The minimum diamond width ($D_W = 16$) of the 25-point constant-coefficient stencil, in Fig. 3.4c, requires a tile size of ~ 3 MiB/thread. To run all the 18 cores of the processor with efficient temporal blocking, the processor has to provide a minimum of $3 * 18 = 54$ MiB of cache memory, which is far from the available one. We observe more starvation in the cache memory in the 25-point variable-coefficient stencil in Fig 3.4d. The 7-point stencils in Figs. 3.4b and 3.4a can fit 18 tiles in the cache memory, but the diamond width has to be limited (i.e., provide limited data reuse) and does not reduce the code balance sufficiently to prevent main memory bandwidth saturation. We investigate these observations in more detail in the results in Chapter 5.

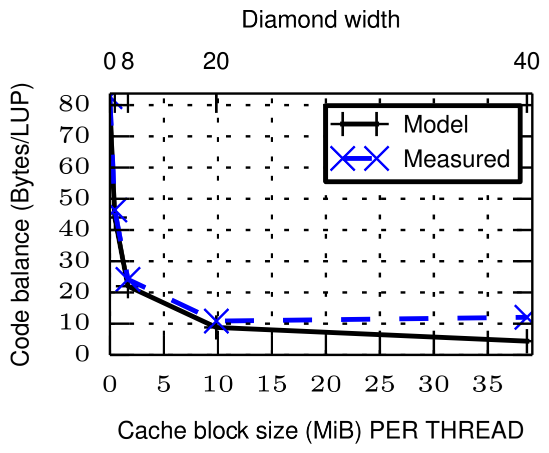
Our accurate cache block size model can be used to tune the tiling parameters to achieve high performance. We can use our model to select the largest tile size that fits in the usable cache memory size to replace auto-tuning, as performed in [7]. On the other hand, even an accurate cache block size model is not sufficient to select the tuning parameters for the best performance. For example, Fig. 3.4c shows a case when a tile size larger than the usable cache memory (i.e., $D_W = 48$ requires ~ 27 MiB cache block) can still achieve better code balance than the maximum tile size that fits entirely in the cache memory (i.e., $D_W = 32$ requires ~ 12.5 MiB cache block). This observation shows the importance of auto-tuning even when an accurate model for the cache block size is available. In fact, we use auto-tuning in our work, assisted with our modeling techniques to narrow down the parameter search space.

3.5 Summary

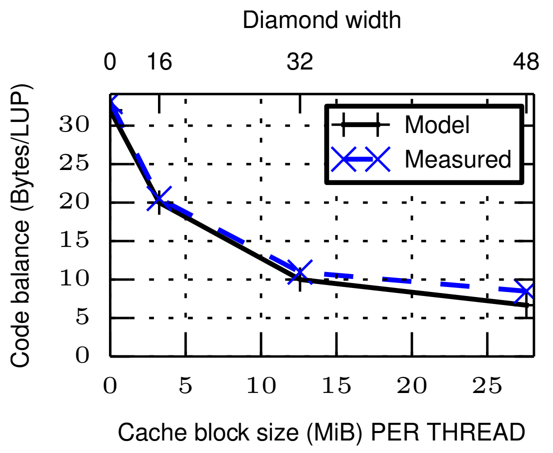
In this chapter, we showed how the best practices in spatial and temporal blocking are not sufficient to overcome the main memory bandwidth limitation in contemporary processors. These approaches will become less efficient in future processors as the



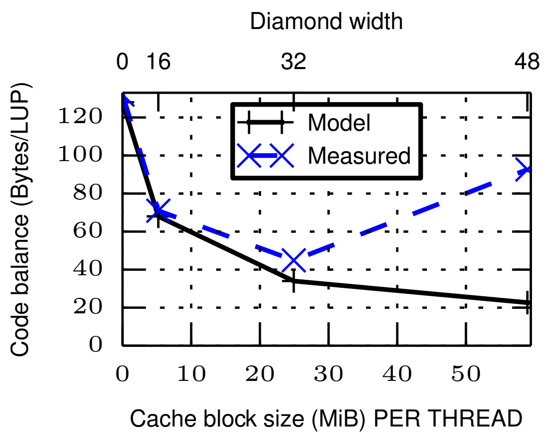
(a) 7-point constant-coefficient stencil using grid size $N = 960^3$



(b) 7-point variable-coefficient stencil using grid size $N = 680^3$



(c) 25-point constant-coefficient stencil using grid size $N = 960^3$



(d) 25-point variable-coefficient stencil using grid size $N = 480^3$

Figure 3.4: Cache block size vs. modeled and measured code balance using four corner-case stencil operators in Intel 18-core Haswell processor. Several diamond tile sizes are evaluated using unit wavefront tile width. Cache block size and code balance are computed using the models in sections 3.4.2 and 3.4.3, respectively. All cases show accurate prediction of the code balance when the cache block size falls within the usable cache size (i.e., half the cache size of the processor).

machine balance decreases, cache size per thread decreases, and concurrency increases. In particular, variable-coefficient and high-order stencils suffer the most from these limitations as they demand more cache memory to decouple from the main memory bandwidth bottleneck.

Chapter 4

Approach: Multi-dimensional intra-tile parallelization

In this chapter, we describe our intra-tile multi-dimensional parallelization algorithm in detail and its wavefront diamond tiling implementation in a structured grid. We also describe the components of our open-source testbed framework, called *Girih* [22]. In addition to the shared memory optimizations, the framework includes a proof-of-concept distributed memory parallelization along the diamond tiling dimension. We utilize the same diamond tile shape for temporal blocking in shared memory and for relaxing the communication of the distributed memory MPI communication.

4.1 Multi-dimensional intra-tile parallelization algorithm

We showed in Section 3.4 that using a single thread per cache block is not sufficient to decouple from main memory in several situations, even with very efficient temporal blocking approaches. Larger cache block sizes than available cache memory are required to reduce the main memory bandwidth requirements. To resolve these issues, we introduce an advanced cache block sharing scheme. It alleviates the pressure on the cache size to provide sufficient data reuse that decouples the computations from the main memory bandwidth bottleneck through larger shared cache blocks.

We introduce a (d+1)-dimensional intra-tile parallelization algorithm for tiled d-dimensional grids. Each Cartesian dimension, i, of the tiles is divided equally into T_i chunks that can be updated concurrently. Additionally, the components of each grid cell may be divided into T_c chunks, when at least T_c equations per grid cell can be updated concurrently. The number of threads updating a tile ("thread group") are equal to $T_c \times \prod_{n=1}^{d} T_n$. Multiple Thread Groups (TGs) may exist to update tiles concurrently. The data reuse in the threads' private caches can be improved by making the boundaries of the sub-tiles parallel to the time dimension, where each thread would be reusing its local grid points across the time steps in the tile.

Our multi-dimensional cache block sharing algorithm has several advantages. First, it requires less main memory bandwidth by enabling more in-cache data reuse through larger cache blocks in the shared cache levels. It has less penalty on the cache block size requirement compared to other cache block sharing approaches, where they require more space to provide sufficient concurrency, as will be shown in the related work, Chapter 7.

Cache block sharing along the leading dimension reduces the need for tiling along it, resulting in better utilization of the hardware data prefetching to the LLC. That is, long contiguous strides are utilized in the shared cache level, while each thread updates a block of the unit stride. On the other hand, tiling along the leading dimension would cause the hardware prefetching unit to load data beyond the cache block boundaries along the leading dimension, which will occupy space in the cache memory and will be evicted from the cache memory before being used.

Finally, coupled with auto-tuning, our cache block sharing algorithm provides a rich set of run-time configurable options that allow architecture-friendly data access patterns for various setups. For example, the auto-tuner finds cases when sharing the

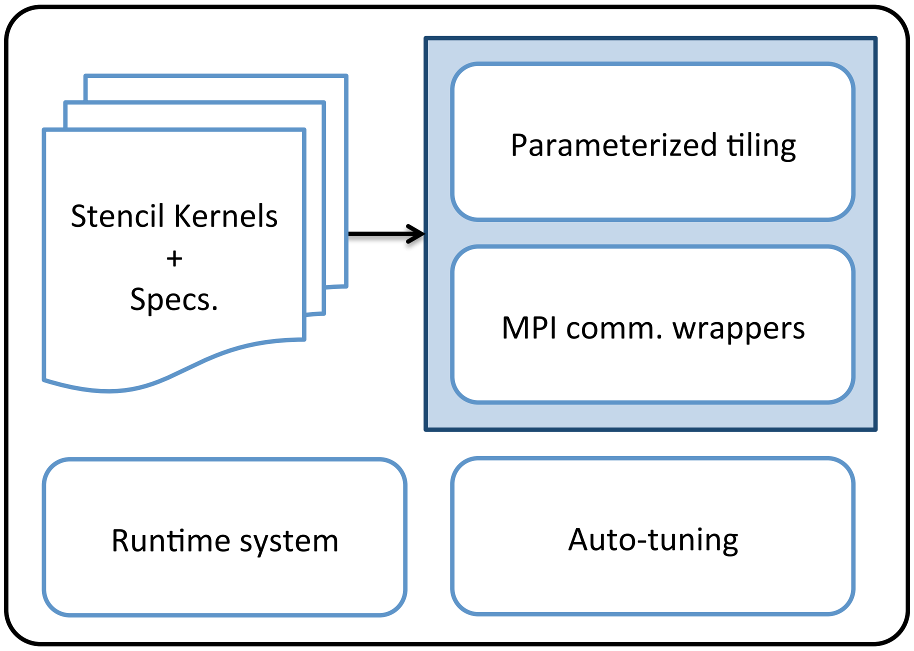


Figure 4.1: *Girih* framework diagram.

leading dimension among multiple threads is beneficial to reduce the cache memory pressure, as will be shown in the results Chapter 5.

4.2 Girih framework

Our system diagram is shown in Fig. 4.1. It consists of our parametrized tiling kernels that use the loop body of the stencil computation and its specifications, for example, stencil radius, as described in Sect. 4.2.1. In Sect. 4.2.3 we describe our runtime system, which dynamically schedules thread groups to the tiles. We use auto-tuning that searches for the best performing parameter set, as described in Sect. 4.2.2. Finally, we use MPI wrappers to handle the distributed memory communication for the tiles at the boundaries of the subdomain, which is described in more detail in Sect. 4.2.4.

4.2.1 Multi-core wavefront temporal blocking

We present a practical implementation of our approach using Multi-threaded Wavefront Diamond blocking (MWD) in Fig. 4.2. Diamond tiling is performed along the y-axis and wavefront blocking is performed along the z-axis. Fig. 4.2a shows the tile decomposition among the threads in three spatial dimensions at a single time step of the tile update. The thread group size is equal to the product of the threads number in all dimensions. Fig. 4.2b shows extruded diamond tile, and Fig. 4.2c and 4.2d show two perspectives of the extruded diamond along the z- and y-axes, respectively. We use this wavefront-diamond tiling implementation because it uses provably-optimal tiling techniques, as described earlier in Chapter 2. Slicing the x-axis into small tiles is known to be impractical as it would negatively affect the TLB misses, hardware prefetching, and the control-flow overhead. We replaced tiling along x-axis with our intra-tile parallelization along x-axis, although it may be useful to combine them in very large grid sizes.

At each time step in the tile update, each thread performs updates of multiple grid points before proceeding to the next time step. All threads update the equal number of grid points to maintain load-balanced work, and they synchronize at certain points to ensure correctness.

We present two wavefront parallelization schemes in Fig. 4.2c and 4.2e. The first one assigns a fixed location for the thread in the wavefront tile, allowing the use of a simple relaxed-synchronization scheme in the thread group, where each thread has data dependency over its left neighbor only. This wavefront approach has the disadvantage of pipelining the data across the cores, resulting in larger data volume transfers in the cache hierarchy. The parallelization scheme in Fig. 4.2e resolves this issue by using a Fixed-Execution to Data (FED) scheme that updates each subset of grid points by a single thread, regardless of the wavefront tile location, allowing wavefront cache block to span multiple cache domains. The limitation of this scheme

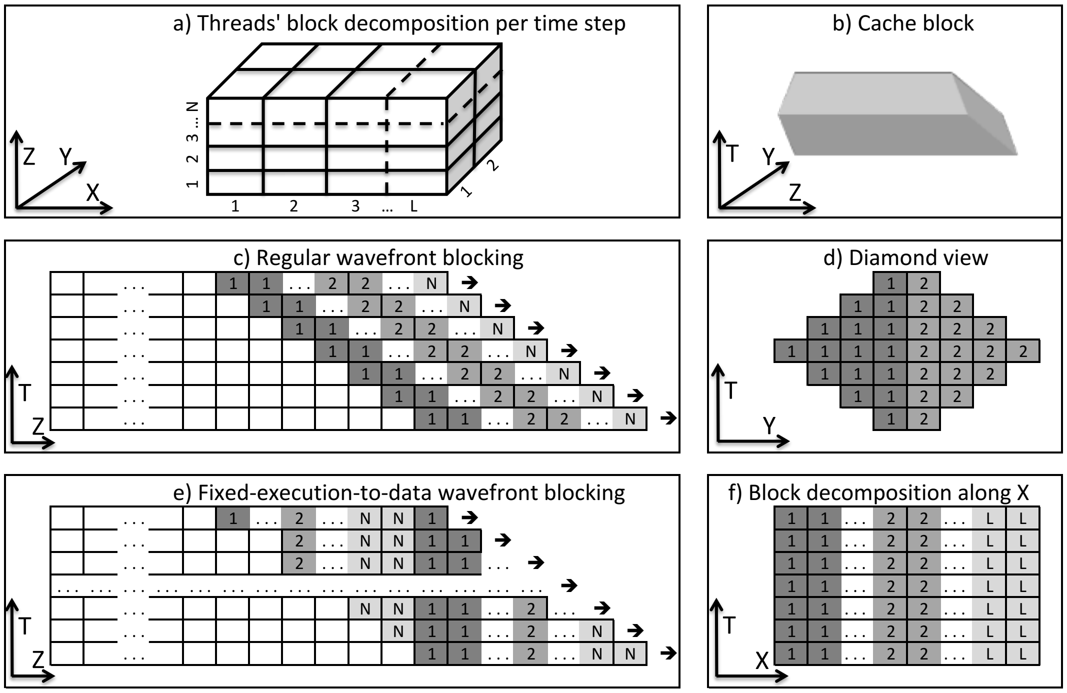

a) Threads' block decomposition per time step
b) Cache block
c) Regular wavefront blocking
d) Diamond view
e) Fixed-execution-to-data wavefront blocking
f) Block decomposition along X
Z
Y
X
T
1 2 3 ... N
1 2 3 ... L

Figure 4.2: Example of implementing multi-dimensional intra-tile parallelization over wavefront diamond tiles in three-dimensional grid.

is the more complex synchronization within the thread group, so we use OpenMP barrier after each time step update in this FED wavefront. The FED wavefront scheme is useful for stencils with high number of bytes per grid cell, such as the variable coefficient and high-order stencils. It is particularly important for high-order stencils because almost no reuse is usually achieved between the time steps of the wavefront due to the steep temporal blocking slope.

Our intra-tile parallelization scheme along the wavefront in Fig. 4.2c and 4.2e allows an arbitrary number of threads to be assigned along the z-axis, with the cost of increasing the cache block size, without increasing the data reuse. An arbitrary number of threads can also be assigned along the x-axis of the tile. The parallelization along the x-axis replaces the tiling along this dimension, when tiling would be useful to reduce the cache block size of the thread. The diamond tile, shown in Fig. 4.2d, uses only one or two threads of parallelization to maintain load-balance and assign the same data to each thread (i.e., use a tile hyperplane parallel to the time dimension).

Relaxed synchronization is used along the z-axis in the regular wavefront approach (Fig. 4.2c). A software sub-group barrier is used to synchronize threads along y- and x-axes for each sub-tile along the z-axis.

Listing 4.1 shows a reduced version of our MWD tiling implementation. The code synchronizes the thread group using an OpenMP barrier after each time step and uses the regular wavefront blocking strategy (described in Fig. 4.2c). The full kernel, the relaxed-synchronization variant, and the FED variant are available online in [22]. We use an OpenMP parallel region to spawn the threads of the thread group. The three-dimensional coordinates of the threads are calculated in line 4. The tile is decomposed among the threads along the y-, x-, and z-axes in lines 7-8, 10-12, and 14, respectively.

Listing 4.1: A variant of MWD for wavefront steady-state update. Tile's bounds along x, z, and time are *xb-xe, zb-ze, tb-te*, respectively. y base bound is *yb-ye*.

```
#pragma omp parallel for ...
{ tid = omp_get_thread_num();
  //Thread 3D coordinates: tid = tid_z*(th_x*th_y) + tid_y*th_x + tid_x
  tid_x=tid%th_x;  tid_y=(tid/th_x)%th_y; tid_z=tid/(th_x*th_y);
  //Decompose the tile along the y-axis
  //Update tile size for each time step using stencil radius (R)
  if (tid_y == 0){ ye=(yb+ye)/2; b_inc=R; e_inc=0;}
  else           { yb=(yb+ye)/2; b_inc=0; e_inc=R;}
  //Decompose the tile along the x-axis
  q=(xe-xb)/th_x; r=(xe-xb)%th_x;
  if(tid_x<r) { ib=xb+tid_x*(q+1); ie=ib+(q+1);}
  else        { ib=xb+r*(q+1)+(tid_x-r)*q; ie=ib+q;}
  //Decompose the tile  along the z-axis, of size bs_z
  zbi = bs_z/th_z * tid_z;  zei = bs_z/th_z * (tid_z+1);
  for(zi=zb; zi<ze; zi+=bs_z) { //wavefront loop along z-axis
    ybi=yb; yei=ye; kt=zi; //Tile's Base index init. along y- and z-axes
    for(t=tb; t<te; t++){ //Tile loop in time
      for(k=kt+zbi; k<kt+zei; k++){ //Tile loop in z
        for(j=ybi; j<yei; j++) { //Tile loop in y
#pragma simd
          for(i=ib; i<ie; i++) { //Tile loop in x
            stencil_operator_loop_body_macro()
      }}}
      //Update block size along y-axis
      if(t<diam_height/2) ybi-=b_inc; yei+=e_inc; //Diamond's lower half
      else                ybi+=e_inc; yei-=e_inc; //Diamond's upper half
      kt -= R; // Update wavefront base index for each time step
#pragma omp barrier //Synchronize after each time step
      ptmp=u; u=v; v=ptmp; /*Swap pointers*/  }}}
```

4.2.2 Auto-tuning

We perform the auto-tuning as a preprocessing step, once the user selects the problem details, for example, the stencil type and grid size. The auto-tuner allocates and initialize the required arrays for its own use and deallocates them once it is completed. This saves significant tuning time when the grid array is very large, in the order of 100 GiB.

Fig. 4.3 shows the details of our auto-tuning approach. It tunes several parameters: diamond width, wavefront tile width, threads along x-, y-, and z-axes. The auto-tuner starts with fixed user-selected parameters, then determines the feasible set of intra-tile thread dimensions by factorizing the available number of threads. It tests the performance of all valid TGS combinations. A local search hill-climbing algorithm is applied to tune the diamond and wavefront tile widths for each TGS case. The auto-tuner uses our cache block size model in Section 3.4.2 and the processor's available cache size (specified by the user) to reduce the parameter search space.

Selecting proper test size, i.e., number of time steps, for auto-tuning test cases is challenging. A very small test may be affected by the system jitter and other sources of noise, which produces a false indication of the achievable performance of the test case. A large test, on the other hand, may increase the tuning time significantly. We use multiples of diamond rows to set the number of time steps. The run time of a single diamond row has many dependencies. It relies on the grid size, stencil type, processor speed, tiling efficiency, etc., so a priori setting of the test size is not practical. To resolve this issue, for each test case, we dynamically change the test size until "acceptable performance" is obtained. We repeat running the test case with increasing number of diamond rows. Once the performance variation between two repetitions falls within a certain threshold, we consider that the largest test case size produces "acceptable performance" and we use it to report the test case performance.

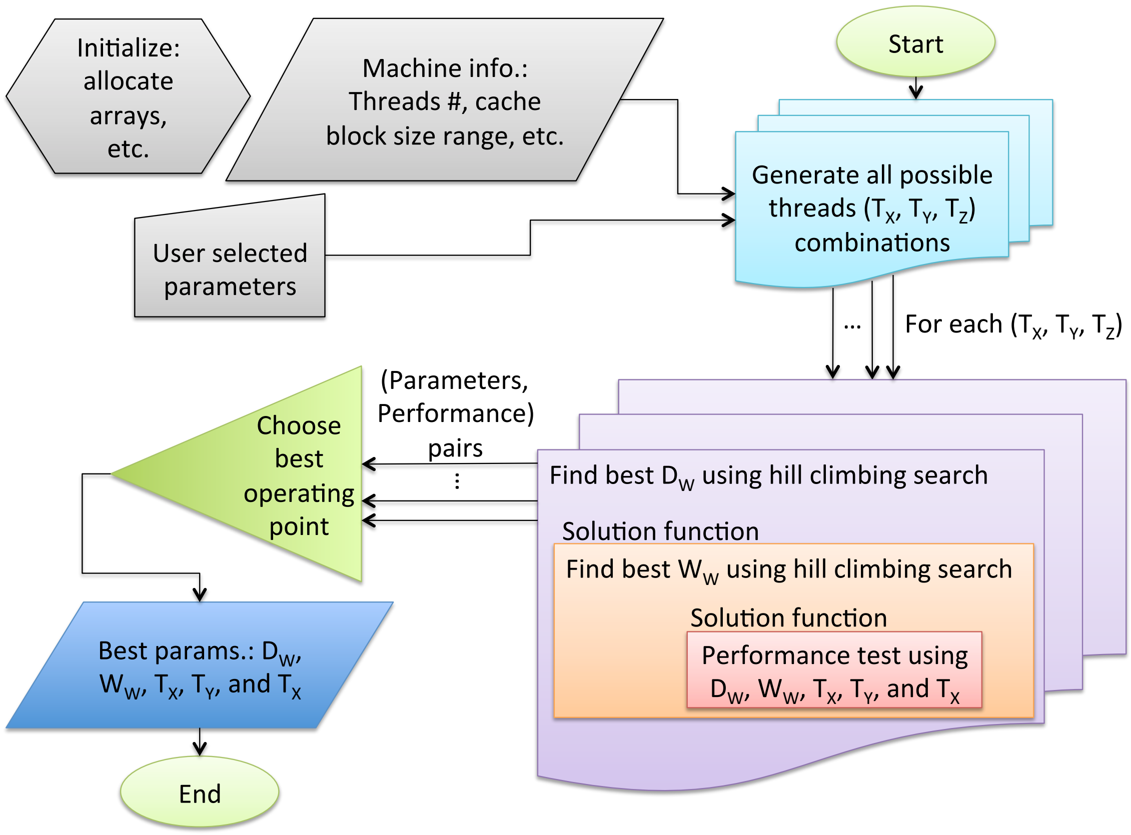

Initialize: allocate arrays, etc.
Machine info.: Threads #, cache block size range, etc.
Start
Generate all possible threads (T_X, T_Y, T_Z) combinations
User selected parameters
...
For each (T_X, T_Y, T_Z)
(Parameters, Performance) pairs
Choose best operating point
Find best D_W using hill climbing search
Solution function
Find best W_W using hill climbing search
Solution function
Performance test using D_W, W_W, T_X, T_Y, and T_X
Best params.: D_W, W_W, T_X, T_Y, and T_X
End

Figure 4.3: *Girih* auto-tuner flow chart.

4.2.3 Runtime system

Threads can be scheduled to the extruded diamonds in a variety of ways. Orozco *et al.* [11] and the current diamond tiling implementation in the PLUTO framework [23] use global synchronization after each row of diamonds update to respect the inter-diamond dependencies. Strzodka *et al.* [7] preassign the diamond tiles to threads before starting the stencil computations, taking the inter-diamond data dependency during tile updates into account. These approaches are sufficient to avoid idling threads as long as the workload is balanced. Workload variation can result from domain boundary handling: In this work, diamond tiles at the boundary of the subdomain exchange data and synchronize with neighbor processes. This causes load imbalance in processing diamond tiles, which varies according to the used network interconnect. To resolve this issue, we schedule the diamond tiles dynamically to the thread groups. The runtime system performs dynamic scheduling of the tiles while respecting data dependencies using a multi-producer multi-consumer FIFO queue that holds the available tiles for update. The First In First Out (FIFO) queue maintains a list of available tiles for update. When a thread has completed updating a tile, it pushes its dependent diamond tile(s) to the queue if those tiles have no other unmet dependencies. "Pop" operations are performed to assign available tiles to thread groups. The FIFO queue is protected from concurrent updates by an OpenMP critical region. Since the queue updates are performed infrequently, the synchronization overhead is negligible.

Our implementation relies on OpenMP nested parallelization. The outer parallel region spawns one thread per thread group (We call them "group masters"), where the tiles are scheduled to thread groups at this level. The inner parallel region spawns the threads of each group to update the grid cells in the tile. Spawning the inner parallel region happens in undefined order many times during runtime, As a result, threads are grouped in different subsets during runtime.

For example, let us consider a case of running six threads in two groups, with pinning order {0,3,1,2,4,5}. The outer parallel region uses threads 0 and 3 for the group master threads. When thread 0 spawns its threads first, the threads are divided into {0,1,2} and {3,4,5} subsets. Threads may also be divided into {0,4,5} and {3,1,2} subsets.

Grouping threads in different subsets does not affect the performance in Uniform Memory Access (UMA) setups, as in single socket Ivy Bridge processor. On the other hand, this issue has to be resolved in Non-Uniform Memory Access (NUMA) cases, such as in the Intel Knight's Corner (KNC) or when running systems with multiple processors.

OpenMP 4.0 provides thread affinity features. The `OMP_PLACES` environment variable allows the user to set thread groups or pools of threads. It also supports thread groups binding in nested parallel regions through the `proc_bind` clause in the parallel region OpenMP pragma. The `proc_bind` feature in the parallel region initialization allows for setting a certain affinity. For example, it is possible to use the `scatter` option at the outer parallel region, which uses 1 thread from each thread group. The `master` option is used at the inner parallel region, so that each thread is spawned from the same thread group as its parent thread. Unfortunately, using this feature in the Intel C compiler 15 degraded the performance of our code to 80% at a single socket. The performance degradation may be a result of sub-optimal use of these features or the Intel implementation is not well optimized yet for these new features.

To resolve this issue efficiently, we use the UNIX low-level interface of `sched_setaffinity` to set the threads affinity manually. Thread affinity is set inside the inner parallel region every time it is executed. It does not hurt the performance much, as there core regions have work in order of hundreds of micro seconds.

It is possible to avoid the thread groups affinity issue by using the pthreads library instead of OpenMP, where thread pinning is needed only once at the initialization stage. We believe the pinning overhead in OpenMP is negligible, so we use it to exploit its portability and usability features.

Another option would be to use single level OpenMP parallel region, but this adds complexity to the thread group scheduling. It also requires creating point-to-point synchronization constructs within each thread group, instead of using OpenMP barriers at the thread groups.

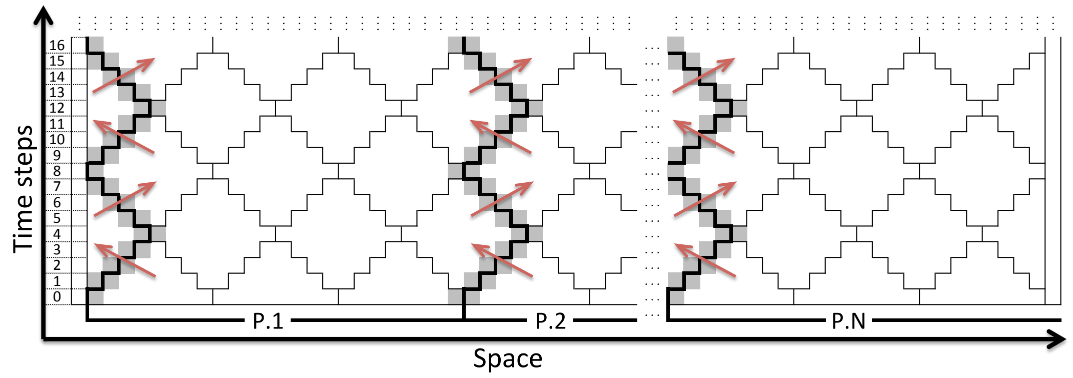
4.2.4 Distributed-memory parallelization

Parallelizing stencil computations over distributed memory nodes is quite straightforward if no temporal blocking is involved. Each time step update is followed by halo data communication. In such a bulk-synchronous scheme, strong scalability is naturally limited by data transfer overhead. A partial remedy is provided by the halo-first update scheme, in which domain boundaries are updated first, and then asynchronous message passing is performed while updating the bulk of the domain.

Distributed memory parallelization can be combined with diamond tiling as shown in Figure 4.4. The arrows represent the data dependencies across subdomains, and the same number of adjacent tiles is assigned to each process except the rightmost one (largest y coordinate). To maintain load balance in terms of computation and communication, the leftmost half diamond tile is assigned to the rightmost process. Regular diamond tiles are used at the boundary of subdomains, with the difference of performing communication before and after the tile update. Thread groups handling boundary diamond tiles are blocked until their MPI communication is complete. Extra delay can occur if no thread group is updating the diamond tile at the other end of the communication. Adding priority in scheduling the tiles at the boundary to thread groups can alleviate this issue, which is left for future work.

Since domain decomposition is performed along the middle space dimension (y), the boundary data to be communicated does not reside in contiguous memory locations. User-defined strided MPI data types are not efficient in our multi-threaded implementation, as MPI implementations handle the required packing/unpacking operations purely sequentially. We use explicit multi-threaded halo data packing/unpacking to resolve this issue.

Diamond tiling offers several advantages in distributed memory parallelization. The tessellation of the diamond tiles allow using a unified tile structure everywhere. It also allows maximum stencil updates in space-time without relying on exchanging boundaries with neighbor processes after each grid sweep. Finally, there is a natural overlap of computation with communication. Communication does not block all threads, and no thread has to be sacrificed for asynchronous communication. Threads can handle communication or perform stencil updates as needed.



(a) Distributed memory parallelization with diamond tiling for a one-dimensional space grid. Arrows represent the data dependencies across subdomains. The leftmost column of half diamonds is assigned to the rightmost process to achieve load balance in computation and communication.

```
update_tile(y_coord, t_coord){
  update_tile_grid_points(y_coord);
  if(y_coord==0 && t_coord%2==0){
      //Leftmost diamond in even diamonds row
      send_left(); recv_right();
      wait_send_left(); wait_recv_right(); }
  if(y_coord==n-1 && t_coord%2==1){
      //Rightmost diamond in odd diamonds row
    send_right(); recv_left();
    wait_send_right(); wait_recv_left(); }}
```

(b) MWD pseudo code for the data exchange of the subdomains. *y_coord*/*t_coord* represent the horizontal/vertical local coordinates of the diamond in the space/time of the subdomain. The number of the diamonds in the row of the subdoamin is n. *t_coord* is used to identify the even/odd rows to select the communication direction of the row. Communication is performed at once for both sides to use single thread for the MPI communication. To maintain the correct tiles update order, boundary diamonds of the rows are updated in chronological order.

Figure 4.4: Distributed memory parallelization along the diamond tiling dimension. We show a diagram of the diamond tiles tessellation at the subdomain boundary in 4.4a and the pseudo code for the computation and communication order in 4.4b.

Chapter 5

Performance results

In this chapter, we perform our experiments over the four stencils under consideration using Intel 10-core Ivy Bride and 18-core Haswell processors, over a broad range of grid sizes (cubic domain). Our MWD approach is faster than 1WD/CATS [7], PLUTO [21, 23], and Pochoir [24] for all stencils on both architectures with most grid sizes. MWD is also the only approach that provides a significant improvement over an efficient spatial blocking implementation in all cases. To understand the strength of our approach in potentially memory bandwidth-starved future processors, we study the impact of MWD cache block sharing over the performance, code balance, memory transfer volumes, and energy consumption. We show significant memory bandwidth usage and memory transfer volumes saving. In order to have better understanding of the different temporal blocking approaches we present thread scaling results. Finally, we present strong scaling performance results of our distributed memory implementation.

5.1 Frameworks setup

We set up both PLUTO and Pochoir frameworks in the presented results. To ensure fair comparison, we investigated and used the best setup of the tested frameworks.

In all the experiments, we run each test case twice and report the performance of the faster one. Since the test cases are large enough, the repeated tests achieve very similar performance. In Sects. 5.2 and 5.3, we run the experiments several times to measure different hardware counter groups. To demonstrate the stability of our results, we plot the reported performance of the repeated experiments in our performance figures. In most cases, the results are aggregated in a single point. At some small grid sizes, where the noise is relatively more significant, the performance points are stretched vertically.

5.1.1 PLUTO setup

PLUTO framework uses `polycc` executable to perform the source-to-source transformation. We use the flags "`--tile --parallel --pet --partlbtile`" to generate efficient codes for our stencil computation kernels. Intel compiler version 15 is used with flags: "`-O3 -xHost -ansi-alias -ipo -openmp`", which are also the default configuration in PLUTO examples.

Compiling the tiled stencil codes using the Intel C compiler 15 achieves twice the performance compared to the Intel C compiler 12. The more recent compiler optimizes PLUTO tiled codes more efficiently and have improvements in vectorizing and using prefetching in the compiled codes.

The selected tiling transformations perform diamond tiling along the z-axis and parallelepiped tiling along the y- and x-axes. Since parameter tuning is essential to run the tiled code efficiently, we implemented a Python script to tune the parameters of each test case in the presented results. We performed brute-force parameter search in diverse setups (i.e., different stencils, processors, and grid sizes) to ensure fair and efficient tuning. We found that the parameter search space is convex, where only a single maximum exists in the tested processors and stencil kernels. In the results presented here, we use a recursive local search algorithm, in the same manner discussed earlier in *Girih* auto-tuner for the MWD code, to tune the three tiling parameters.

5.1.2 Pochoir setup

Pochoir does not have performance-critical tuning parameters for tiling, as it relies on cache-oblivious algorithms. The default compiler flags of Pochoir examples are used "`-O3 -funroll-loops -xHost -fno-alias -openmp`".

5.1.3 Girih setup

We use Intel compiler options: "`-O3 -xAVX -fno-alias -openmp`" in *Girih* codes. Our autotuning code is used to select the diamond and wavefront tiles widths and thread group parallelization parameters in the results.

The wavefront implementation parameter is selected according to the stencil type. A relaxed-synchronization wavefront scheme is used for the 7-point stencils, as this implementation has lower synchronization overhead. FED is not important here as sufficient data reuse is possible using reasonable wavefront width. On the other hand, FED is used in the 25-point stencils, to allow sufficient data reuse in the wavefront update. The efficiency of these options are also confirmed by manually testing several cases.

We also tune the 1WD case separately. This is the nearest implementation we have to CATS2 algorithm of [7].

5.2 Performance at increasing grid size

In this section, we present a performance comparison between MWD, PLUTO, Pochoir, 1WD/CATS, and spatial blocking. In all the results, we use a range of cubic grid sizes, where each dimension is set to multiples of 64 in the Ivy Bridge processor and multiples of 128 in the Haswell processor. We maximize the grid size range such that it reaches the memory capacity limits (32 GiB in the Ivy Bridge and 128 GiB in the Haswell). For a better identification of relevant bottlenecks we also show memory bandwidth usage and memory transfer volume per LUP.

Our major finding is that MWD is faster than 1WD, PLUTO, and Pochoir for all stencils on both architectures with most grid sizes. MWD is also the only approach that provides a significant improvement over the efficient spatial blocking in all cases. Especially for the high-order (25-point) stencils it is the only efficient solution. In general, the Haswell processor shows better speedups for temporal blocking vs. spatial blocking due to its large number of cores (18), which leads to a low machine balance of 0.14 bytes/flop (assuming fused multiply-add is not used; with FMA, the machine balance goes down further to 0.07 bytes/flop). In contrast, the Ivy Bridge CPU only has 10 cores, a 5% lower clock speed, and a 20% lower memory bandwidth for a better machine balance of 0.23 bytes/flop. A low machine balance generally indicates a more "bandwidth-starved" situation with a higher potential for temporal blocking techniques, although a quantitative analysis requires a more elaborate performance model. See below for details.

More details of the hardware counter measurements of the presented results are available in Appendix B. We refer to them whenever the hardware counters measurements of the cache levels and the CPU are mentioned in the text.

At large grid sizes, where boundary effects become negligible, it is possible to construct a phenomenological ECM performance model by combining measured data traffic per LUP in all memory levels with code composition characteristics such as the number of load instructions per LUP, as described in Sect. 2.4. Tables 5.1 and 5.2 summarize the comparison between the phenomenological models and the measurements. Details are discussed below.

Stencil	Model [cy]	Pred. [GLUP/s]	Meas. [GLUP/s]
7-pt const. coeff.	$\{12 \,\|\, 14 \,\|\, 14 \,\|\, 8.3 \,\|\, 2.2\}$	4.6	4.1
7-pt var. coeff.	$\{14 \,\|\, 28 \,\|\, 30 \,\|\, 24 \,\|\, 11\}$	1.6	1.4
25-pt const. coeff.	$\{12 \,\|\, 56 \,\|\, 40 \,\|\, 28 \,\|\, 11\}$	1.3	1.2
25-pt var. coeff.	$\{12 \,\|\, 76 \,\|\, 115 \,\|\, 50 \,\|\, 40\}$	0.44	0.36

Table 5.1: Phenomenological ECM models, predictions, and performance measurements for the four stencils under investigation with MWD at large grid sizes on the Intel Ivy Bridge CPU.

Stencil	Model [cy]	Pred. [GLUP/s]	Meas. [GLUP/s]
7-pt const. coeff.	$\{12 \,\|\, 14 \,\|\, 7 \,\|\, 7.5 \,\|\, 1.8\}$	10	8.0
7-pt var. coeff.	$\{14 \,\|\, 21 \,\|\, 14 \,\|\, 25 \,\|\, 4.8\}$	3.9	2.6
25-pt const. coeff.	$\{12 \,\|\, 56 \,\|\, 20 \,\|\, 30 \,\|\, 7.4\}$	2.5	2.2
25-pt var. coeff.	$\{12 \,\|\, 38 \,\|\, 56 \,\|\, 50 \,\|\, 26\}$	0.71	0.65

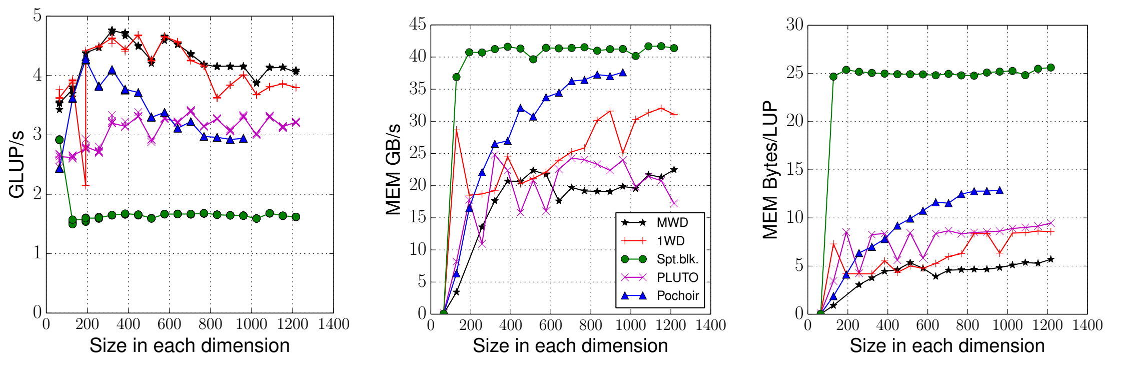
Table 5.2: Phenomenological ECM models, predictions, and performance measurements for the four stencils under investigation with MWD at large grid sizes on the Intel Haswell CPU.

5.2.1 7-point stencil with constant coefficients

We present the Intel Ivy bridge performance results using different frameworks in Fig. 5.1a. We also show the measured memory bandwidth usage in Fig. 5.1b and the memory transfer volumes normalized by the number of lattice-site updates in Fig. 5.1c. Likewise, we present the Intel Haswell results in Fig. 5.2

This stencil performs seven flops per LUP and has a code balance for pure spatial blocking of 24 bytes/LUP in double precision. The memory-bound maximum performance is thus (41 GB/s)/(24 bytes/LUP) $\approx$ 1.7 GLUP/s in Ivy Bridge and (50 GB/s)/(24 bytes/LUP) $\approx$ 2.1 GLUP/s in Haswell.

As shown in Figures 5.1a and 5.2a, all temporal blocking variants outperform optimal spatial blocking by far. MWD and 1WD are consistently faster than PLUTO and Pochoir, with MWD taking a clear lead for larger grid sizes. MWD also exhibits the lowest memory bandwidth usage, as shown in Figs. 5.1b and 5.2b and the lowest code balance (memory traffic per LUP), as shown in Figs. 5.1c and 5.2c. For most

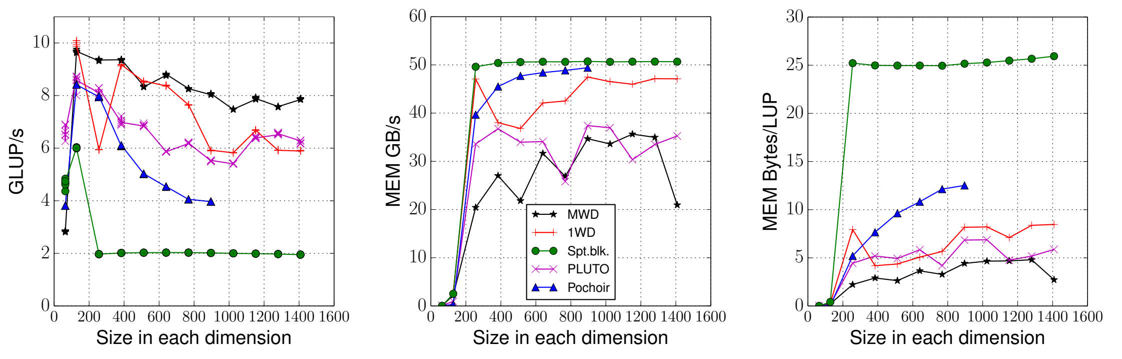


(a) Performance.

(b) Measured memory bandwidth usage.

(c) Measured memory transfers per LUP.

Figure 5.1: Ivy Bridge 7-point constant-coefficient stencil results, using increasing cubic grid size. Showing performance and memory transfer measurements of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.



(a) Performance.

(b) Measured memory bandwidth usage.

(c) Measured memory transfers per LUP.

Figure 5.2: Haswell 7-point constant-coefficient stencil results, using increasing cubic grid size. Showing performance and memory transfer measurements of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

grid sizes the auto-tuner selects a thread group size of 2 or 6, except for very large and very small problems.

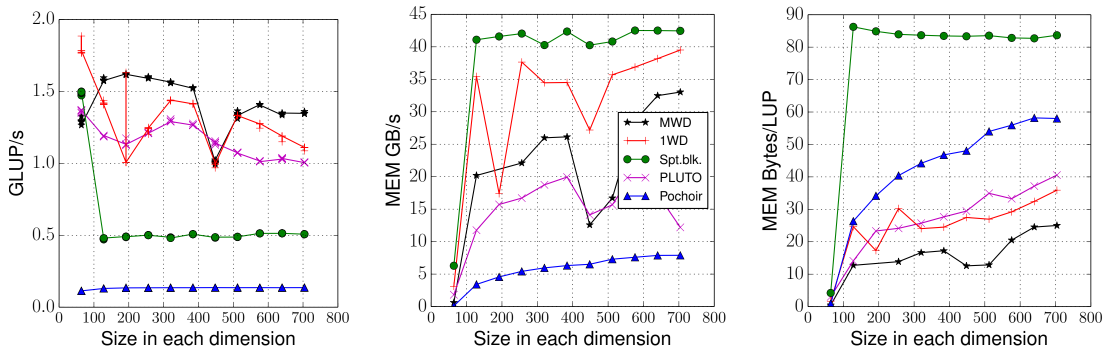
We construct a phenomenological ECM performance model [15] from results at large grid sizes. We combine measured data traffic per LUP in all memory levels (presented in Appendix B) with code composition characteristics such as the number of load instructions per LUP. The 7-point stencil with constant coefficients uses 28 half-wide (16-byte) load instructions per 8 LUP, which require 14 cycles to execute on both architectures. The required data traffic in the L2 cache is 56 bytes/LUP since the L1 cache is too small to hold three consecutive rows of the source array. Hardware counter measurements confirm this estimate for MWD. On Ivy Bridge (Haswell), this transfer takes 14 (7) cycles. The L3 cache traffic is hard to predict due to the blocking strategy. We use the measured value of 33 (30) bytes/LUP on Ivy Bridge (Haswell), leading to a transfer time of about 8.25 (7.5) cycles. In memory, we observe a code balance of about 5 bytes/LUP. With the chosen input data the ECM model predicts a socket-level performance of about 4.6 GLUP/s on Ivy Bridge and 10.9 GLUP/s on Haswell. We observe that our performance prediction is within a 10% margin of the measurements on Ivy Bridge and about 25% above the measurements on Haswell. Overall the ECM model describes the performance characteristics of the MWD code at larger problem sizes quite well, proving that the MWD code is operating at the limits of the hardware. This observation is also valid for the other stencils described below. Further substantial performance improvements requires optimizations that reduce the amount of work.

We work out the details of the ECM model for this stencil in the Ivy Bridge processor, using the notation described in Section 2.4. The ECM model data is $\{12 \,\|\, 14 \,|\, 14 \,|\, 8.25 \,|\, 2.2\}$ cy, which leads to $T_{\mathrm{ECM}} = \{14 \rceil 28 \rceil 36.25 \rceil 38.45\}$ cy. The performance per core is thus $\mathrm{Perf}_{\mathrm{ECM}} = \{1.26 \rceil 0.63 \rceil 0.49 \rceil 0.46\}$ GLUP/s, where we compute it by multiplying the 8 LUPs work unit by the CPU clock frequency (2.2

GHz) and dividing by the T_{ECM} cycles. The single core ECM prediction leads us to a full socket performance of 4.6 GLUP/s.

5.2.2 7-point stencil with variable coefficients

We present the Intel Ivy bridge performance results using different frameworks in Fig. 5.3a. We also show the measured memory bandwidth usage in Fig. 5.3b and the memory transfer volumes normalized by the number of lattice-site updates in Fig. 5.3c. Likewise, we present the Intel Haswell results in Fig. 5.4

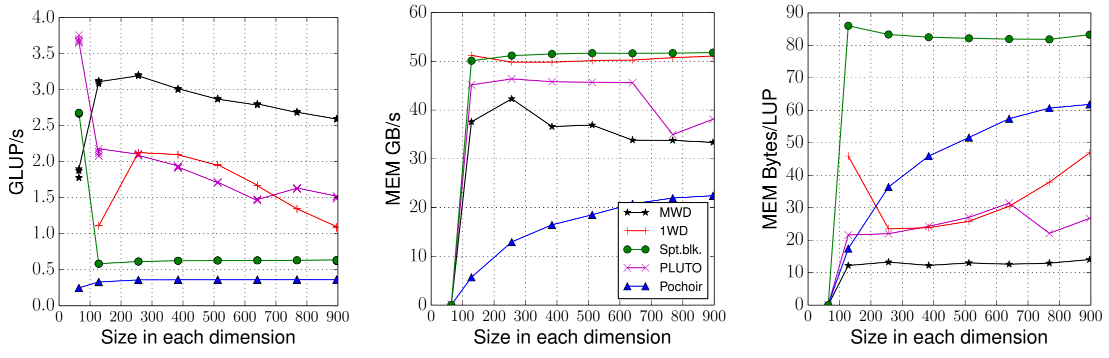


(a) Performance. (b) Measured memory bandwidth usage. (c) Measured memory transfers per LUP.

Figure 5.3: Ivy Bridge 7-point variable-coefficient stencil results, using increasing cubic grid size. Showing performance and memory transfer measurements of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

This stencil performs 13 flops/LUP, but it has a higher pressure on memory for purely spatial blocking (80 bytes/LUP instead of 24). Hence, we expect more potential for temporal blocking and accordingly a higher speedup for MWD, which exhibits the lowest in-memory code balance of all, as shown in Figs. 5.3c and 5.4c. Indeed the speedup of MWD compared to spatial blocking is 2.8× – 3.2× on Ivy Bridge and 4.5× – 5.2× on the more bandwidth-starved Haswell.

On Ivy Bridge, Pochoir has the lowest memory bandwidth usage at large problem sizes, but it is five times slower than even spatial blocking. This memory bandwidth saving shows that low memory bandwidth usage does not guarantee high performance.



(a) Performance. (b) Measured memory bandwidth usage. (c) Measured memory transfers per LUP.

Figure 5.4: Haswell 7-point variable-coefficient stencil results, using increasing cubic grid size. Showing performance and memory transfer measurements of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

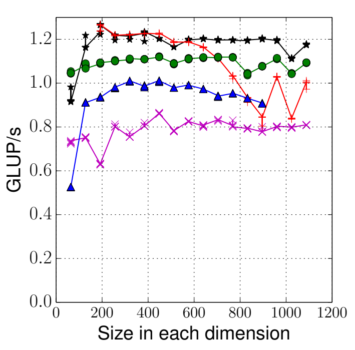
In this particular case, the compiler appears to have generated exceptionally slow code. Even if the memory bandwidth were fully utilized (40 GB/s instead of 8 GB/s) the performance would hardly surpass spatial blocking.

The phenomenological ECM model for MWD with stencil predicts a performance of 1.6 GLUP/s (3.9 GLUP/s) at large problem sizes for the full Ivy Bridge (Haswell) socket assuming perfect scalability. The deviation from the measurement may be attributed to the cost of the fine-grain synchronization in large thread groups, which are used in large grid size to decouple from the main memory bottleneck.

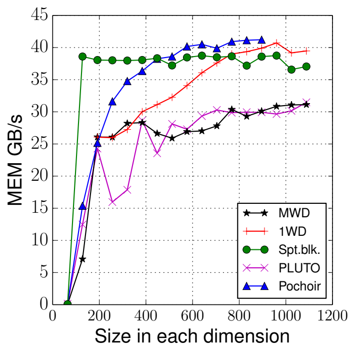
5.2.3 25-point stencil with constant coefficients

We present the Intel Ivy bridge performance results using different frameworks in Fig. 5.5a. We also show the measured memory bandwidth usage in Fig. 5.5b and the memory transfer volumes normalized by the number of lattice-site updates in Fig. 5.5c. Likewise, we present the Intel Haswell results in Fig. 5.6

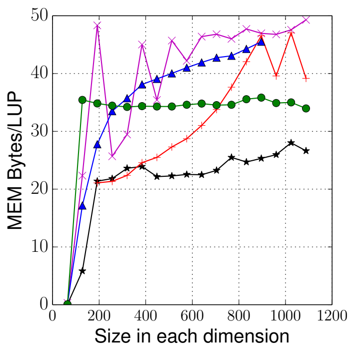
Due to its high ratio of 33 flops/LUP and its large radius this stencil poses a challenge for temporal blocking schemes. Many layers of grid points have to be supplied by the caches, so the time needed for memory transfers is rather small [15]



(a) Performance.

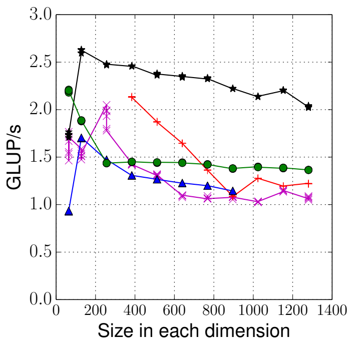


(b) Measured memory bandwidth usage.

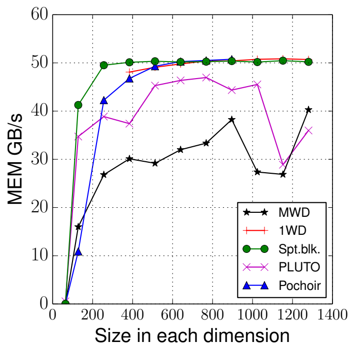


(c) Measured memory transfers per LUP.

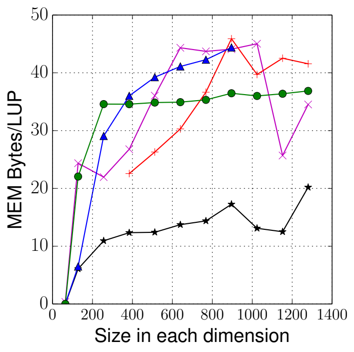
Figure 5.5: Ivy Bridge 25-point constant-coefficient stencil results, using increasing cubic grid size. Showing performance and memory transfer measurements of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.



(a) Performance.



(b) Measured memory bandwidth usage.



(c) Measured memory transfers per LUP.

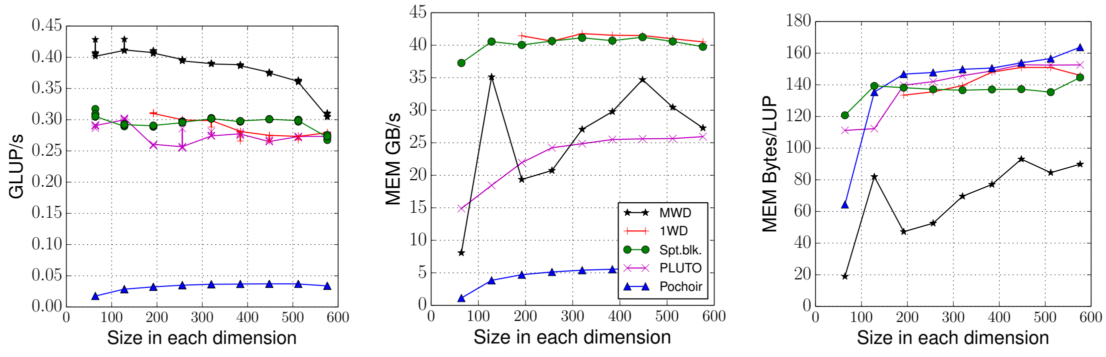
Figure 5.6: Haswell 25-point constant-coefficient stencil results, using increasing cubic grid size. Showing performance and memory transfer measurements of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

and the cache size required for temporal blocking is large. As shown in Figs. 5.5a and 5.6a, only MWD reduces the memory pressure significantly, since it can leverage the multi-threaded wavefronts to reduce the need for cache size. As a consequence, only MWD is consistently faster than pure spatial blocking by a factor of 1.1× on Ivy Bridge and by 1.5× – 1.7× on Haswell. Pochoir and 1WD even exhibit a memory code balance larger than spatial blocking for most problem sizes, as shown in Figs. 5.5c and 5.6c.

With MWD the compiler produces some register spilling, adding to the dominance of the in-core and in-cache contributions to the runtime. At large grid sizes, the ECM model predicts a full-socket MWD performance of 1.3 GLUP/s on Ivy Bridge and of 2.5 GLUP/s on Haswell. Both predictions are reasonably close to the measurements.

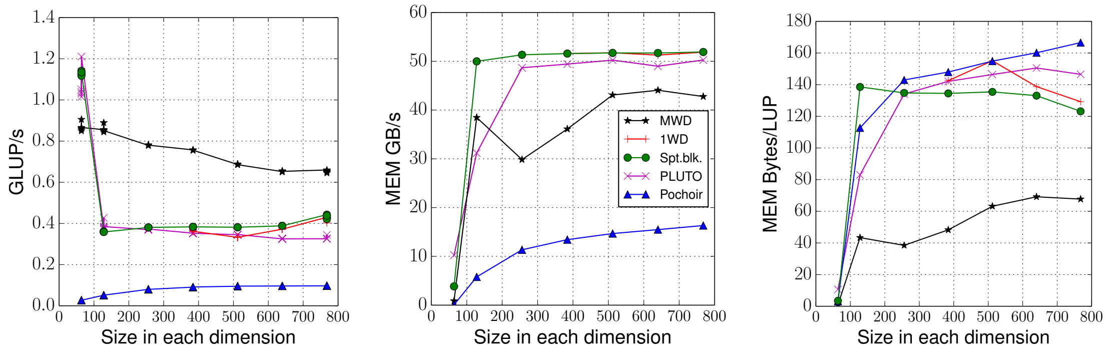
5.2.4 25-point stencil with variable coefficients

We present the Intel Ivy bridge performance results using different frameworks in Fig. 5.7a. We also show the measured memory bandwidth usage in Fig. 5.7b and the memory transfer volumes normalized by the number of lattice-site updates in Fig. 5.7c. Likewise, we present the Intel Haswell results in Fig. 5.8



(a) Performance. (b) Measured memory bandwidth usage. (c) Measured memory transfers per LUP.

Figure 5.7: Ivy Bridge 25-point variable-coefficient stencil results, using increasing cubic grid size. Showing performance and memory transfer measurements of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.



(a) Performance. (b) Measured memory bandwidth usage. (c) Measured memory transfers per LUP.

Figure 5.8: Haswell 25-point variable-coefficient stencil results, using increasing cubic grid size. Showing performance and memory transfer measurements of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

This stencil performs 37 flops/LUP but requires much more data due to the variable coefficients array (128 bytes/LUP for efficient spatial blocking). Although it has a lower intensity compared to the 25-point constant-coefficient stencil, it poses the same cache size problems. Again, only MWD can reduce the code balance significantly, as shown in Figs. 5.7c and 5.8c. MWD is the only scheme that is faster than spatial blocking at all (by 1.2× – 1.3× on Ivy Bridge and by 1.5× – 2× on Haswell), as shown in Figs. 5.7a and 5.8a. Pochoir shows low memory bandwidth usage at the same memory code balance as spatial blocking. Measurements, in Figs. B.4e and B.8e in Appendix B, show that Pochoir requires a massive amount of data traffic between the L1 and L2 cache for this stencil, which is a contributor (in addition to slow low-level code) to its extremely low performance.

The phenomenological ECM model for MWD yields socket-level estimates of 0.44 GLUP/s on Ivy Bridge and 0.71 GLUP/s on Haswell, both of which are in line with the measurements.

5.3 MWD tile sharing impact on performance, memory transfer, and energy consumption

While the cache block sharing reduces the memory bandwidth requirements of the stencil codes, it increases the overhead by performing fine-grain synchronization among more threads. As a result, the auto-tuner would select the minimum thread groups size that sufficiently decouple from the main memory bandwidth, when allowed to tune all the parameters.

In this section, we run the MWD approach using fixed thread group sizes to study their impact on the performance, memory bandwidth and transfer, and energy consumption. The auto-tuner selected the parallelization dimensions and tiling parameters in these experiments. We limit the energy consumption analysis to the Intel Ivy Bridge processor. The memory modules of the Haswell processor consume a similar amount of energy regardless of the memory bandwidth usage, making our optimization techniques oblivious to the energy consumption rate.

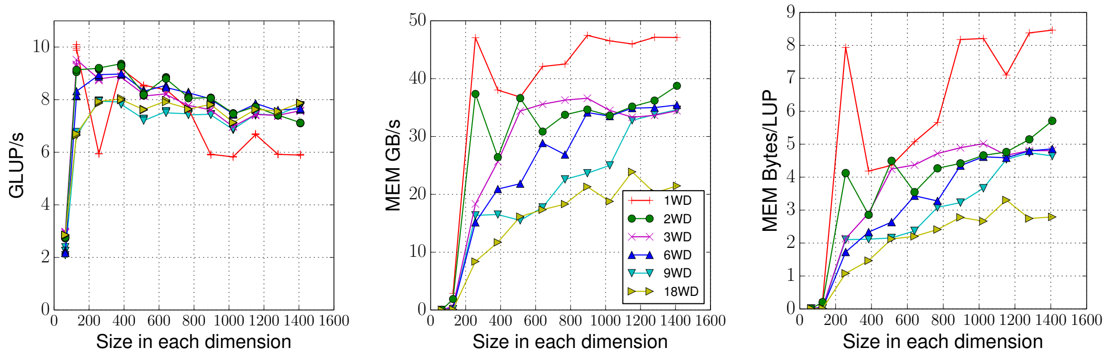
We perform our experiments over the four studied stencil kernels in this work. We observe similarity in the studied characteristics of these stencils, so we describe representative subset of these results here. The remaining results are moved to Appendix C.

In all the results, we use a range of cubic grid sizes, where each dimension is set to multiples of 64 in the Ivy Bridge processor and multiples of 128 in the Haswell processor.

We describe the Haswell processor's results of the 7-point constant-coefficient and the 25-point variable-coefficient stencils, to show MWD behavior at two extremes in the code balance. The energy consumption behavior is illustrated using the 25-point constant-coefficient stencil results in the Ivy Bridge processor.

5.3.1 7-point stencil with constant coefficients

We present the Intel Haswell performance results using different thread group sizes in Fig. 5.9a. We also show the measured memory bandwidth usage in Fig. 5.9b, and the memory transfer volumes normalized by the number of lattice-site updates in Fig. 5.9c.



(a) Performance. (b) Measured memory bandwidth usage. (c) Measured memory transfers per LUP.

Figure 5.9: Haswell performance and memory transfer measurements of the 7-point constant-coefficient stencil using increasing cubic grid size. We compare various thread group sizes in MWD.

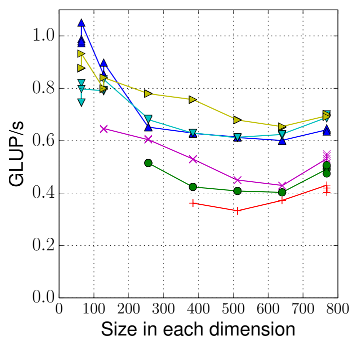
The 1WD implementation does not saturate the memory bandwidth at smaller grid size than 800^3, as this stencil has small bytes requirements and moderate code balance. When 1WD runs at grid sizes larger than 800^3 the memory bandwidth saturates and the performance drops, as larger cache blocks cannot fit in the L3 cache memory. 2WD and larger thread group sizes are sufficient to decouple from the main memory bandwidth bottleneck at large grid sizes.

At small grid sizes, the synchronization overhead of 9WD and 18WD is relatively significant compared to the computations, resulting less efficient performance. As the computations increase at larger grid sizes, the synchronization cost of 9WD and 18WD becomes less significant, where all MWD variants achieve similar performance.

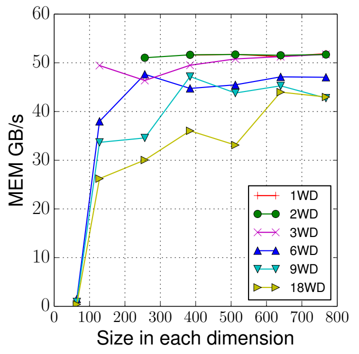
Larger thread group sizes save more cache memory as they keep a smaller number of tiles in the cache memory. The cache size saving provides space for larger diamond tiles, which increases the in-cache data reuse and decreases the main memory bandwidth usage and data traffic. Figures 5.9b and 5.9c show how larger thread group sizes consume less memory bandwidth and transfer less data per lattice update. This memory transfers saving shows that our MWD approach is suitable for the future processors, which are expected to be more memory bandwidth starved. Although the synchronization overhead of MWD impacts the performance, we find it acceptable compared to the memory bandwidth saving and the performance improvements.

5.3.2 25-point stencil with variable coefficients

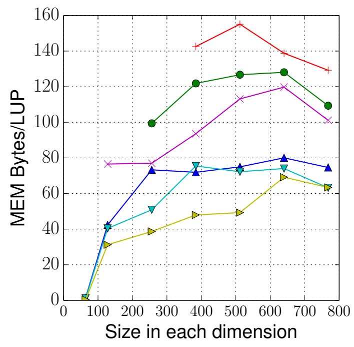
We present the Intel Haswell performance results using different thread group sizes in Fig. 5.10.



(a) Performance.



(b) Measured memory bandwidth usage.



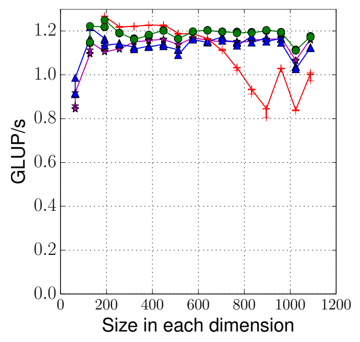
(c) Measured memory transfers per LUP.

Figure 5.10: Haswell performance and memory transfer measurements of the 25-point variable-coefficient stencil using increasing cubic grid size. We compare various thread group sizes in MWD.

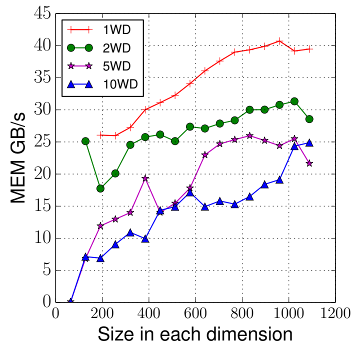
The 25-point variable-coefficient stencil has high code balance and large cache block size requirements, so large thread group size is necessary to decouple from the main memory bandwidth bottleneck. As shown in Fig. 5.10a, 18WD achieves the best performance at most of grid sizes. Other MWD variants saturate or nearly saturate the main memory bandwidth at large grid sizes, as shown in Fig. 5.10b, and incur large memory data volume transfer, as shown in Fig. 5.10c.

5.3.3 25-point stencil with constant coefficients

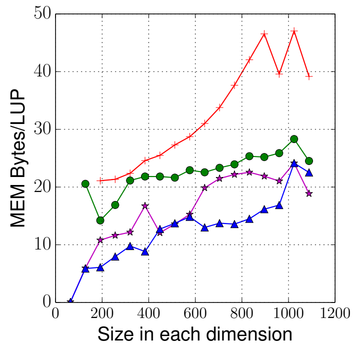
We present the Intel Ivy bridge performance results using different thread group sizes in Fig. 5.11a, along with the measured memory bandwidth usage in Fig. 5.11b. The memory transfer volumes in Fig. 5.11c and the energy estimates in Figs. 5.11d, 5.11e, and 5.11f are normalized by the number of lattice updates.



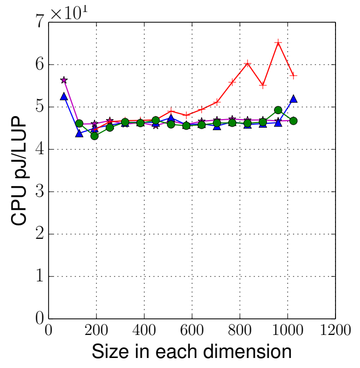
(a) Performance.



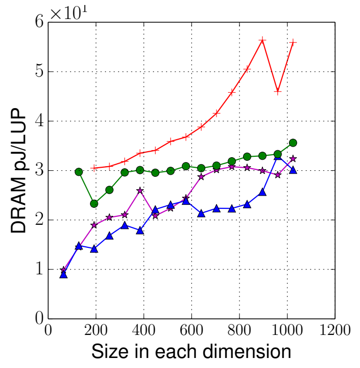
(b) Measured memory bandwidth usage.



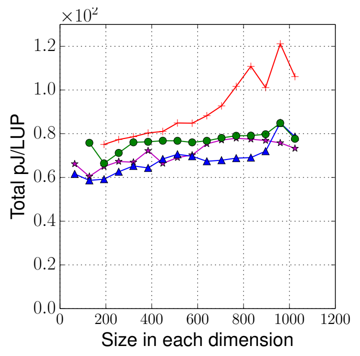
(c) Measured memory transfers per LUP.



(d) CPU energy consumption estimates.



(e) DRAM energy consumption estimates.



(f) Total energy consumption estimates.

Figure 5.11: Ivy Bridge performance, memory transfer measurements, and energy consumption estimates of the 25-point constant-coefficient stencil using increasing cubic grid size. We compare various thread group sizes in MWD.

1WD achieves the best performance up to the grid size 512^3 before it starts saturating the memory bandwidth interface (shown in Fig. 5.11b) and increase its data transfer volumes (shown in Fig. 5.11c).

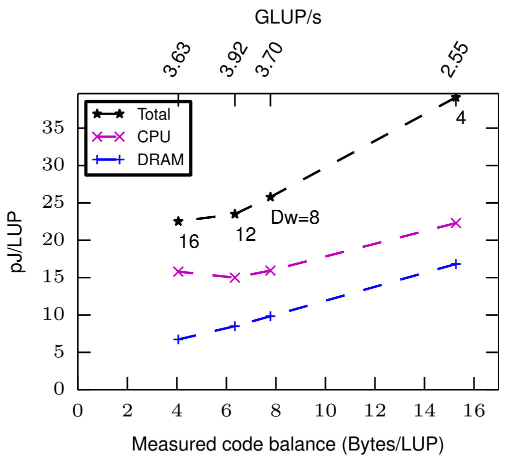
Although 2WD achieves the best performance with grid sizes larger than 512^3, all MWD variants achieve similar performance. This similarity indicates negligible synchronization overhead among all MWD variants.

The energy results in Fig. 5.11f shows that the usual law of “faster code uses less energy”, or “race-to-halt” as defined by Hennessy and Patterson [25], is not always true. That is, we observe that 10WD achieves the lowest energy to the solution in most cases, although it does not achieve the best performance among the other MWD variants. We obtain this result because the DRAM energy consumption is proportionally correlated to its bandwidth usage. The memory bandwidth saving of 10WD results in decreasing its DRAM energy consumption, as we observe in Fig. 5.11e, while consuming the same CPU energy of other variants.

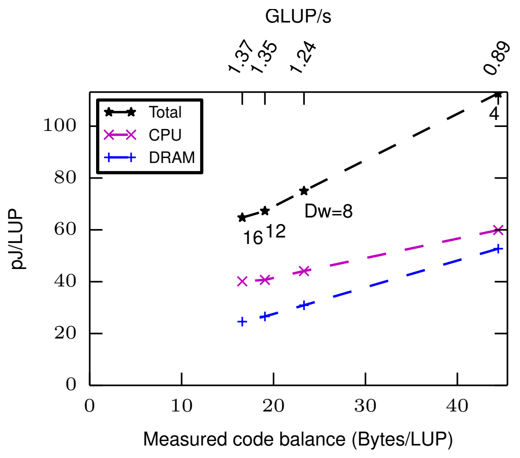
5.4 Code balance and energy consumption analysis

These findings would not justify favoring the maximum thread group size over all other options on the Ivy Bridge processor. However, these findings show clearly that if the future moves towards more memory bandwidth-starved systems and higher relative power dissipation in the memory subsystem, it should use algorithms that exhibit lowest possible code balance. This view is corroborated by another observation in our data: The *overall* energy savings of temporal blocking vs. standard spatial blocking are roughly accompanied by equivalent runtime savings. But when the energy consumption of CPU and DRAM are inspected separately it is evident that this equivalence emerges from the mutual cancellation of two opposing effects: While the CPU energy is less strongly correlated with the code performance, the DRAM energy shows an over-proportional reduction for temporal blocking.

This can be seen more clearly in Fig. 5.12 where we have measured the energy to solution with respect to the code balance for 5WD (as a consequence of setting different diamond tile sizes) for both 7-point stencils (the diagram for the 25-point stencil would only contain a single data point per set). In both cases the DRAM energy depends much more strongly on the code balance than the CPU energy. This was expected from the observations described above, but the CPU energy dependence is far from weak. Overall there is an almost linear dependence of energy on code balance, making the latter a good indicator of the former.



(a) 7-point constant-coefficient stencil at grid size $N = 960^3$.



(b) 7-point variable-coefficient stencil at grid size $N = 480^3$.

Figure 5.12: Using Intel Ivy Bridge, energy vs. code balance for the seven-point stencils at several diamond tile sizes, separately for DRAM and CPU and as a total sum. The corresponding performance of each experiment is shown on the top x-axis. The annotation at each point represents the used diamond width. 5WD is used in the experiments.

5.5 Thread scaling performance

All measurement results discussed so far were taken on a full socket. In order to better understand the shortcomings and advantages of the different temporal blocking approaches we present thread scaling results in this section. For each stencil we show the scaling behavior of performance, memory bandwidth, and measured code balance at a fixed grid size on the 18-core Haswell socket.

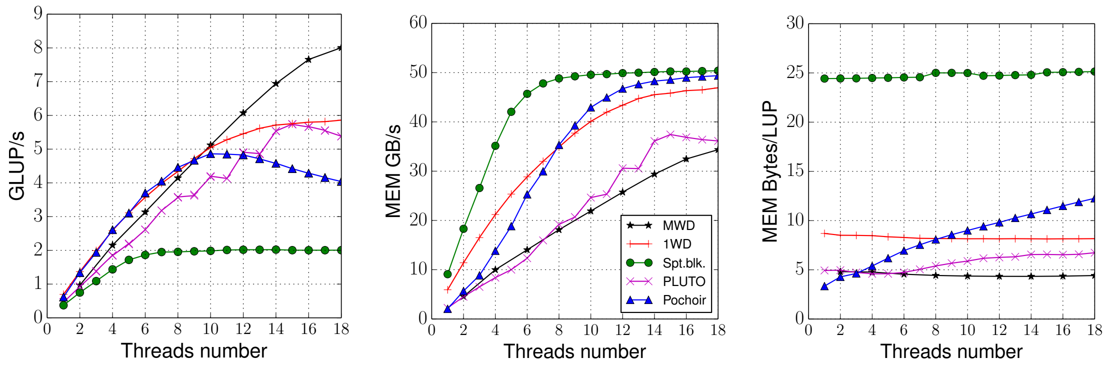
In the cases of 1WD, MWD, and PLUTO, the parameters are tuned only at 18 threads. Experiments with less number of threads use the same parameters of the full socket of the stencil. This results in having less points in MWD plots, as the thread count must be multiple of the thread group size.

5.5.1 7-point constant-coefficient stencil

The thread scaling results for the 7-point constant-coefficient stencil are shown in Figures 5.13a, 5.13b, and 5.13c. All temporally blocked variants except Pochoir show a roughly constant code balance with increasing thread count, but only MWD shows good scaling across the whole chip. Pochoir and 1WD clearly run into the bandwidth bottleneck; the limited scalability of PLUTO is not caused by traffic issues. MWD shows a linearly rising memory bandwidth utilization, indicating bottleneck-free and balanced execution.

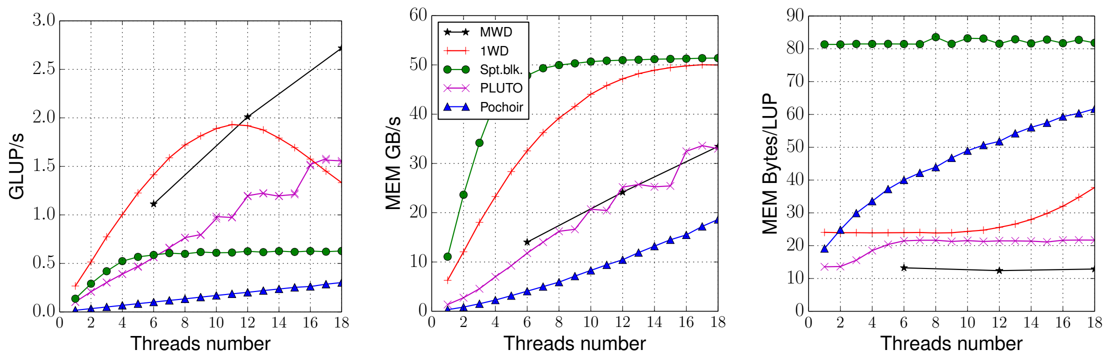
5.5.2 7-point variable-coefficient stencil

The thread scaling results for the 7-point variable-coefficient stencil are shown in Figures 5.14a, 5.14b, and 5.14c. Again, MWD exhibits a constant, low code balance and good scaling. Starting at six threads, PLUTO also shows constant code balance but on a 60% higher level. Since the memory bandwidth is roughly the same as with MWD, performance also scales at a much lower level. An interesting pattern can be observed with 1WD: at rising thread count the shared cache becomes too small to accommodate the required tiles for maintaining sufficient locality, leading to a steep increase in code balance beyond ten cores. Since the memory bandwidth is already almost saturated at this point, performance starts to break down. This behavior was expected from the discussion in the earlier sections, but it is evident now that 1WD would be the best choice on a CPU with only ten cores but with the same cache size. 1WD is also the only temporal blocking variant that is not decoupled from the memory bandwidth. In case of Pochoir the data shows that the decoupling is due to very slow low-level code, as shown before.



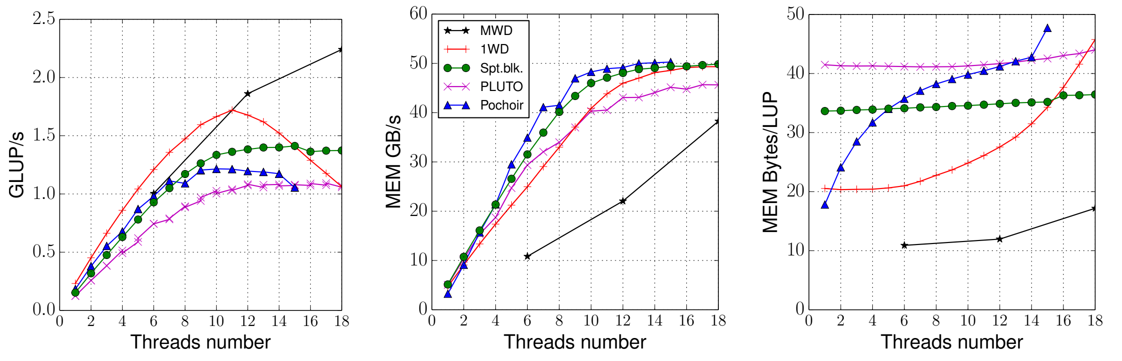
(a) Performance. (b) Measured memory bandwidth. (c) Measured memory transfers per LUP.

Figure 5.13: Thread scaling for the 7-point constant-coefficient stencil, showing performance and memory transfer measurements. We compare PLUTO, Pochoir, 1WD, MWD, and spatially blocked code variants on the 18-core Haswell socket at a grid size of 896^3.



(a) Performance. (b) Measured memory bandwidth. (c) Measured memory transfers per LUP.

Figure 5.14: Thread scaling for the 7-point variable-coefficient stencil, showing performance and memory transfer measurements. We compare PLUTO, Pochoir, 1WD, MWD, and spatially blocked code variants on the 18-core Haswell socket at a grid size of 768^3.



(a) Performance. (b) Measured memory bandwidth. (c) Measured memory transfers per LUP.

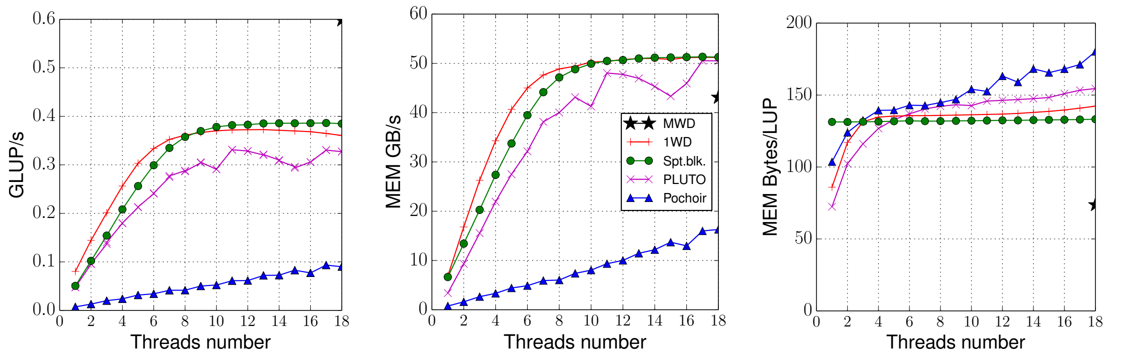
Figure 5.15: Thread scaling for the 25-point constant-coefficient stencil, showing performance and memory transfer measurements. We compare PLUTO, Pochoir, 1WD, MWD, and spatially blocked code variants on the 18-core Haswell socket at a grid size of 896^3.

5.5.3 25-point constant-coefficient stencil

Thread scaling results for the 25-point constant-coefficient stencil are shown in Figures 5.15a, 5.15b, and 5.15c. Due to the massive cache size requirements of this stencil only MWD is still able to decouple from the memory bandwidth. All other variants show strong saturation, or even a slowdown in case of 1WD beyond ten threads, which is caused by the same cache size issues as with the 7-point variable-coefficient stencil. Again, 1WD would be the method of choice if the chip only had ten cores but the same shared cache size.

5.5.4 25-point variable-coefficient stencil

Thread scaling results for the 25-point variable-coefficient stencil are shown in Figures 5.16a, 5.16b, and 5.16c. Note that there is now only a single data point in each figure for MWD (at 18 threads). All variants except MWD exceed even the spatial blocking code balance beyond five threads and thus show strong performance saturation, with the exception of Pochoir, which again suffers from code quality issues. Even if one were able to accelerate the Pochoir code so that it could saturate the



(a) Performance. (b) Measured memory bandwidth. (c) Measured memory transfers per LUP.

Figure 5.16: Thread scaling for the 25-point variable-coefficient stencil, showing performance and memory transfer measurements. We compare PLUTO, Pochoir, 1WD, MWD, and spatially blocked code variants on the 18-core Haswell socket at a grid size of 768^3.

bandwidth, it would still end up at a lower performance level than all others (about 0.3 GLUP/s).

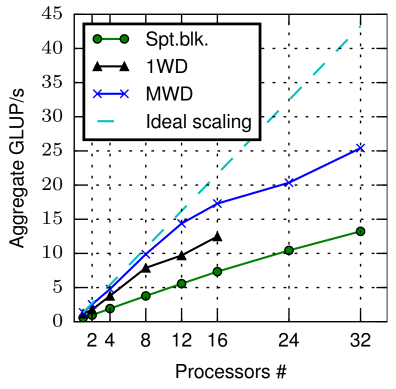
5.6 Distributed memory strong scaling performance

We perform strong scaling experiments using the 7-point variable-coefficient and 25-point variable-coefficient stencils using Intel MPI library 5.0. The domain is decomposed across the y axis using MPI, as described in Sect. 4.2.4. An Intel Ivy Bridge socket is assigned to each MPI process, using ten OpenMP threads per socket.

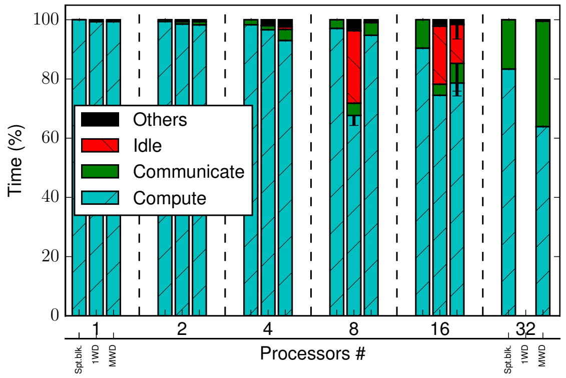
We perform domain decomposition along the diamond tile dimension in this implementation to show the effectiveness of diamond tiling for both shared and distributed memory setup. In this thesis, we do not provide fully distributed memory implementation with multi-dimensional domain decomposition.

5.6.1 7-point stencil with variable coefficients

We show the strong scaling performance results of the 7-point variable-coefficient stencil using our distributed memory MWD implementation in Figure 5.17.



(a) Performance. Ideal scaling based on MWD performance at a single process.



(b) Time distribution, showing time percentage spent in stencil computation, MPI communication, etc. Each cluster of bars corresponds to a fixed number of MPI processes. Error bars with standard deviation under 3% are suppressed.

Figure 5.17: Distributed memory strong scaling performance of the 7-point stencil with variable coefficients at a grid size of 768^3. Each MPI process uses a ten core Intel Ivy Bridge processor.

1WD does not work beyond 16 processes because smaller subdomains in the y axis cannot provide sufficient concurrency to run all the available threads (“concurrency condition”). To run 1WD at 24 MPI processes the minimum subdomain size would be 4 [min. diamond width] *10 [threads/process] * 24 [processes] = 960 grid points along the y axis. MWD has less concurrency requirements in the diamond tiling dimension, as it introduces other dimensions of shared memory concurrency. For example, 10WD achieves the concurrency condition using 50% of the minimum grid size that satisfies the concurrency condition at 1WD.

The auto-tuner selects 2WD in processes range 1–16 and 10WD at 24 and 32 processes. 2WD would achieve less performance compared to 10WD at 24 and 32 processes due to the restrictions imposed by the concurrency condition. 2WD is limited to small diamond tile sizes at 24 and 32 processes ($D_w = 4$) compared to $D_w = 8$

at 16 processes. On the other hand, 10WD can use larger tile sizes while satisfying the concurrency condition. For example, 10WD use $D_w = 24$ at 32 processes. The same concurrency limitation causes the performance drop of 1WD at 12 and 16 processes. 1WD uses $D_w = 4$ at 12 and 16 processes compared to $D_w = 8$ at 8 processes.

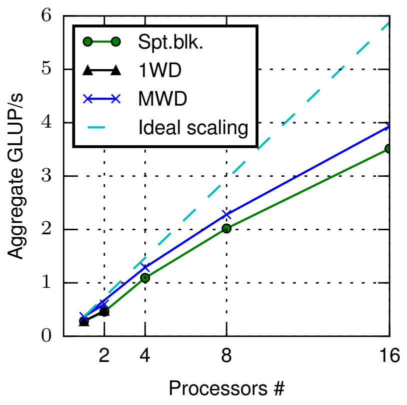
We use timing routines in the code to profile the major parts. We present the time distribution of the distributed memory results in Figure 5.17b. The run time involves performing stencil updates ("Compute"), communicating the halo data across MPI processes ("Communicate"), and thread groups idle time when the task queue is empty in the MWD implementation ("Idle"). Thread groups can have different time distribution as they perform their tasks independently from each other. We use error bars in Fig. 5.17b to present the standard deviation of the thread groups' run time for each component in the stacked bars.

When 1WD or MWD approach their concurrency limit, the idle time percentage increases. For example, in Figure 5.17b, the subdomain of 1WD at 16 processes has 12 diamond tiles in the row ($N_y/(D_w \times P) = 768/(4 \times 16) = 12$ diamond tiles/row, where P is the number of processes), which is very close to the concurrency limit of the ten threads. Updating boundary tiles takes more time compared to updating interior tiles because of the data exchange. When the runtime updates the interior diamond tiles of a row before the boundary tiles, less concurrency will be available. This concurrency limitation causes some thread groups to remain idle when the subdomain size is near the concurrency limit.

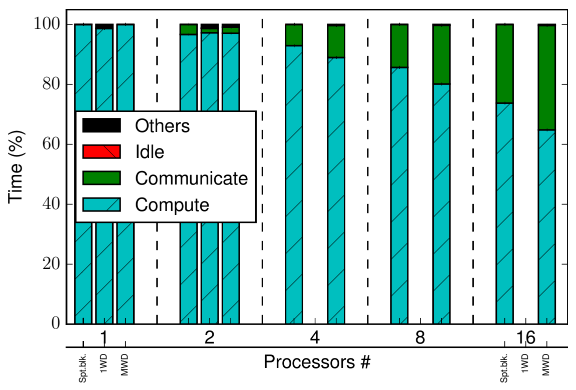
As shown in Figure 5.17a, the MWD implementation scales well up to 16 processes and demonstrates the compatibility of MWD for strong thread scaling with distributed memory scaling, up to a surface-to-volume limitation. The large surface-to-volume ratio of the subdomains at 24 and 32 processes results in large communication overhead, as shown in Figure 5.17b. In fact, at 32 sockets the time for communication with neighboring processes (calculated from the asymptotic network bandwidth and the data volume) is about half the time needed to perform the stencil updates (calculated from the raw update performance and the number of lattice points and time steps). Performing domain decomposition at additional dimensions would allow the code to have a scalable performance at more processes.

5.6.2 25-point stencil with variable coefficients

We present strong scaling performance results of the 25-point variable-coefficient stencil in Figure 5.18. Because of the larger stencil semi-bandwidth, halos comprise multiple grid layers. This high-order stencil has large communication volume, causing the communication time percentage to increase quickly, as shown in Fig. 5.18a). The auto-tuner uses $D_W = 32$ at most of the MWD test cases. As a result, the halo layers of four time steps are communicated at once. The diamond tile at the subdomain boundary communicates $4 \times 4 = 16$ x-z-planes for each diamonds row update.



(a) Performance. Ideal scaling based on MWD performance at a single process.



(b) Time distribution, showing time percentage spent in stencil computation, MPI communication, etc. Each cluster of bars corresponds to a fixed number of MPI processes. Error bars with standard deviation under 3% are suppressed.

Figure 5.18: Distributed memory strong scaling performance of the 25-point stencil with variable coefficients at a grid size of 512^3. Each MPI process uses a ten core Intel Ivy Bridge processor.

Chapter 6

Application: Accelerating a Maxwell Equations solver for a solar cell simulation

Understanding and optimizing the properties of solar cells is becoming a key issue in the search for alternatives to nuclear and fossil energy sources. A theoretical analysis via numerical simulations involves solving Maxwell's Equations in discretized form and typically requires substantial computing effort. We start from a hybrid-parallel (MPI+OpenMP) production code that implements the THIIM with Finite-Difference Frequency Domain (FDFD) discretization, introduced by Pflaum *et al.* [26]. Although this algorithm has the characteristics of a strongly bandwidth-bound stencil update scheme, it is significantly different from the popular stencil types that have been exhaustively studied in the high performance computing literature to date. In this chapter, we apply a our MWD approach and describe in detail the peculiarities that need to be considered due to the special stencil structure.

It is worth mentioning that we investigated another application with multicomponent and variable-coefficient stencils for wave equation solvers. Our analysis showed high arithmetic intensity in the stencil computations, which would result in low or

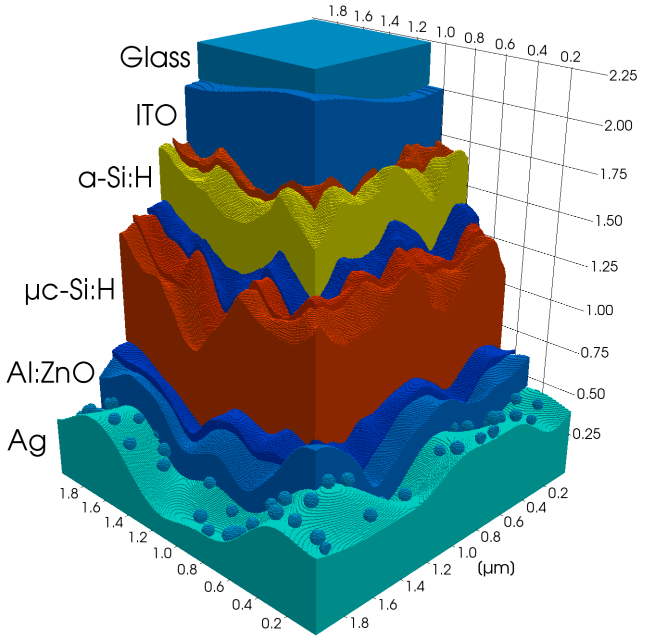


Figure 6.1: Cross-section of a sample simulation setup of a tandem thin-film solar cell. The amorphous (a-Si:H) and microcrystalline silicon (μc-Si:H) layers have textured surfaces to increase the light trapping ability of the cell. SiO_2 nano particles are incorporated to further increase light scattering at the bottom electrode (Ag).

no benefit of our temporal blocking techniques in contrast to the fruitful situation herein, as described in Appendix E. This contrast emphasizes the wide variability of code balance among computational expressions of physical models.

6.1 Introduction

We apply our stencil optimization technology to a challenging problem in computational physics. Photovoltaic (PV) devices play a central role in the recent transition from nuclear and fossil fuels to more environmentally friendly sources of energy. There exist various different PV technologies, ranging from well-established polycrystalline silicon solar cells with thicknesses up to 300 μm to more recent thin-film technologies with active layer thicknesses of only 1 μm or less. To improve these thin-film PV devices and make them more competitive against other renewable energy sources, optimization of their optical properties is decisive. The importance of an optimal collection of the incident light can be seen in any of the recently developed most efficient solar cell designs [27].

To understand the effects of different light trapping techniques incorporated into PV devices and improve upon them, detailed optical simulations are necessary. The simulation code we investigate here uses the THIIM [28], which is based on the staggered grid algorithm originally proposed by Yee [29] and uses the FDFD method to discretize Maxwell's Equations.

Although the algorithm performs stencil-like updates on the electric and magnetic field components, the code is more complex than the well-studied standard stencil benchmarks such as the 7-point stencil with constant coefficients emerging from a discretized Laplace operator. Multiple components per grid cell are involved since six coupled partial differential equations discretized by finite differences must be solved. It uses staggered grids, which results in non-symmetric data dependencies that affect the tiling structure. The loop kernels of the simulation code have very low arithmetic intensity (0.18 flops/byte) for the naive implementation), leading to a memory bandwidth-starved situation. The number of bytes per grid cell is large (40 double-complex numbers), which makes it difficult to maintain a sufficiently small working set to have the necessary in-cache data reuse for decoupling from the main memory bandwidth bottleneck. Despite the complexity, this application lends itself to the optimizations studied earlier in much simpler contexts.

The time-harmonic variants of Maxwell's equations are given by

$$i\omega\hat{\mathbf{E}} = \frac{1}{\epsilon}\nabla \times \hat{\mathbf{H}} - \frac{\sigma}{\epsilon}\hat{\mathbf{E}} \;\;, \tag{6.1}$$

$$i\omega\hat{\mathbf{H}} = -\frac{1}{\mu}\nabla \times \hat{\mathbf{E}} - \frac{\sigma^\star}{\mu}\hat{\mathbf{H}} \;, \tag{6.2}$$

with permittivity ϵ, permeability μ, the electric and magnetic conductivities σ and $\sigma^\star$, and the frequency of the incident plane wave ω. The time-independent electric and magnetic field components are related to the time-dependent fields by $\vec{\mathbf{E}} = \hat{\mathbf{E}}e^{i\omega\tau}$ and $\vec{\mathbf{H}} = \hat{\mathbf{H}}e^{i\omega\tau}$.

After discretization of Maxwell's equations in time and space the following iterative scheme is obtained:

$$\frac{e^{i\omega\tau}\ \hat{\mathbf{E}}_h^{n+1} - \hat{\mathbf{E}}_h^{n}}{\tau} = \frac{1}{\epsilon}\nabla_h \times \hat{\mathbf{H}}_h^{n+\frac{1}{2}}\ e^{i\omega\frac{\tau}{2}} - \frac{\sigma}{\epsilon}\hat{\mathbf{E}}_h^{n+1}\ e^{i\omega\tau} + \mathbf{S}_E\ , \tag{6.3}$$

$$\frac{e^{i\omega\frac{\tau}{2}}\ \hat{\mathbf{H}}_h^{n+\frac{1}{2}} - e^{-i\omega\frac{\tau}{2}}\ \hat{\mathbf{H}}_h^{n-\frac{1}{2}}}{\tau} = -\frac{1}{\mu}\nabla_h \times \hat{\mathbf{E}}_h^{n} - \frac{\sigma^\star}{\mu}\hat{\mathbf{H}}_h^{n+\frac{1}{2}} + \mathbf{S}_H\ , \tag{6.4}$$

with time step τ, time step index n and source terms $\mathbf{S}_E$ and $\mathbf{S}_H$. To model materials with negative permittivity ($\epsilon < 0$, for example, silver electrodes) the THIIM method applies a "back iteration" scheme to the electric field components of the corresponding grid points:

$$\frac{e^{i\omega\tau}\ \hat{\mathbf{E}}_h^{n} - \hat{\mathbf{E}}_h^{n+1}}{\tau} = \frac{1}{\epsilon}\nabla_h \times \hat{\mathbf{H}}_h^{n+\frac{1}{2}}\ e^{i\omega\frac{\tau}{2}} - \frac{\sigma}{\epsilon}\hat{\mathbf{E}}_h^{n+1} + \mathbf{S}_E\ . \tag{6.5}$$

With this method, the optical constants of any material can be used directly in the frequency domain without the need for any approximation or auxiliary differential equations [30, 31, 32, 33]. THIIM has proven to be numerically stable and give accurate solutions for setups with metallic back contacts [34, 35] and also for the simulation of plasmonic effects, for example, around silver nano wires [36].

A perfectly matched layer (PML) is used to allow absorption of outgoing waves, employing the split-field technique originally presented by Berenger [37]: All six $\hat{\mathbf{E}}$ and $\hat{\mathbf{H}}$ field components are split into two parts each. For example, the $\hat{\mathbf{E}}_\mathbf{x}$ component of equation 6.1 is split into $\hat{\mathbf{E}}_\mathbf{x} = \hat{\mathbf{E}}_\mathbf{xy} + \hat{\mathbf{E}}_\mathbf{xz}$, resulting in two equations:

$$(i\omega\epsilon + \sigma_y)\,\hat{\mathbf{E}}_\mathbf{xy} = \frac{\partial}{\partial y}\left(\hat{\mathbf{H}}_\mathbf{zx} + \hat{\mathbf{H}}_\mathbf{zy}\right), \tag{6.6}$$

$$\left(i\omega\epsilon + \sigma_z\right)\hat{\mathbf{E}}_{\mathbf{xz}} = -\frac{\partial}{\partial z}\left(\hat{\mathbf{H}}_{\mathbf{yx}} + \hat{\mathbf{H}}_{\mathbf{yz}}\right). \tag{6.7}$$

For all six vector components this procedure is performed on Equations 6.3, 6.4 and 6.5 resulting in a total of 12 coupled equations.

In order to overcome the problem of the representation of complicated light-trapping geometries, such as rough interfaces between layers or curved particle surfaces, the Finite Integration Technique (FIT) [38] is applied on the rectangular structured grids. FIT allows to accurately treat curved interfaces by integrating the material data on an unstructured tetrahedron grid and mapping the data back to the structured grid.

Figure 6.1 shows a sample simulation setup of a thin-film tandem solar cell that can be simulated by the methods mentioned above [39]. The amorphous and microcrystalline silicon layers are used to absorb different ranges of the incident spectrum. Their surfaces are etched to increase the trapping of light inside the cell. Atomic force microscopy is used to obtain height information that is then introduced between the layers in the simulation. Additionally, at the back electrode (Ag) SiO_2 nano particles can be deposited to increase the scattering of light. For such a setup PML boundary conditions are chosen vertically. Horizontally periodic boundary conditions are used.

We optimize the multi-threaded (OpenMP-parallel) part of the THIIM code using our MWD approach. Our multi-dimensional intra-tile parallelization implementation shows a significant reduction in the cache block size requirement, providing sufficient data reuse in the cache to decouple from the main memory bandwidth bottleneck. As a result, we obtain a $3\times - 4\times$ speedup compared to an efficient spatially blocked code. In addition to the performance improvements, our results show significant memory bandwidth savings of 38% – 80% off the available memory bandwidth, making it immune to more memory bandwidth-starved systems. Via appropriate cache block size and code balance models we prove that cache block sharing is essential for decoupling from the memory bandwidth bottleneck. We validate these models by analyzing different tile sizes and measuring relevant hardware performance counters.

6.2 Intra-tile parallelization implementation

Figure 6.2 shows the diamond tiling implementation of the THIIM stencil kernel. We split the $\hat{\mathbf{H}}$ and $\hat{\mathbf{E}}$ field updates in the figure as they have different data dependency directions. The $\hat{\mathbf{H}}$ and $\hat{\mathbf{E}}$ fields have dependencies over the positive and negative directions, respectively, as illustrated in Fig. 6.3. Splitting the fields allows more data reuse in the diamond tile and provides proper tessellation of diamond tiles. As a result, a full diamond tile update starts and ends with an $\hat{\mathbf{E}}$ field update. The horizontal (blue) lines in Fig. 6.3 divide the components in three regions, which can be handled by three threads. See below for details.

The extruded diamond tile is shown in Figure 6.4. We perform the wavefront traversal along the z-axis (outer dimension) and the diamond tiling along the y-axis (middle dimension). We do not tile the x-axis (fast moving dimension), as we split its work among multiple threads with simultaneous updates in the TG.

The staggered grid and multicomponent nature of this application requires different intra-tile parallelization strategies than "standard" structured grid implementations.

We allow a concurrent update of the x-axis grid cells by the threads in the TG while the data is in cache. This handling of the x-axis has two advantages: it reduces the pressure on the private caches of the threads, and it maintains data access patterns that allow for efficient use of hardware prefetching.

The fixed amount of work per time step in the z and x-axes leads to good load balancing, but parallelizing the diamond tile along the y-axis can be inefficient since the odd number of grid points at every other time step in the diamond tile makes

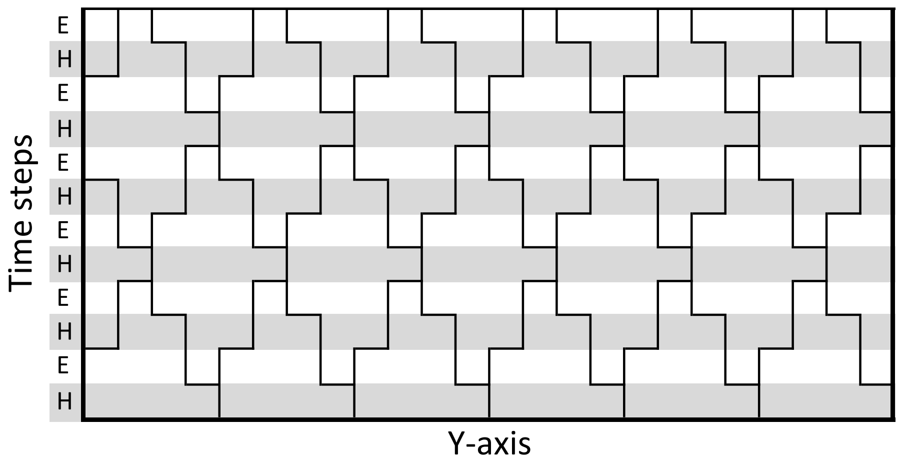


Figure 6.2: Diamond tile shape along the y-axis for the THIIM stencil. Although the $\hat{\mathbf{H}}$ and $\hat{\mathbf{E}}$ fields are updated in the same iteration of the simulation code, we split them in our tiling implementation to achieve better data reuse and better diamond tile tessellation.

load balancing impossible for more than one thread along the y-axis. Doubling the diamond tile width is possible, but it would result in doubling the cache block size without increasing the data reuse. Moreover, a load-balanced implementation cannot make the intra-tile split parallel to the time dimension, so more data will have to move between the private caches of the threads. As a result, we do not perform intra-tile parallelization along the diamond tiling dimension for this stencil.

We exploit the concurrency in the field component updates by adding a further dimension of thread parallelism. Each field update can update six fields concurrently. We parameterize our code to allow 1, 2, 3, and 6-way parallelism in the field update so that the auto-tuner selects the most performance-efficient configuration. For example, Fig. 6.3 shows a case of parallelizing the components update using three threads.

In our spatial blocking and MWD benchmark implementations we use homogeneous Dirichlet boundary conditions in all dimensions to study the performance improvements of our techniques. We expect no significant changes in performance with periodic boundary conditions. The code can use periodic boundary conditions along the x-axis by unrolling the first and last iteration of the x-loop to explicitly specify the

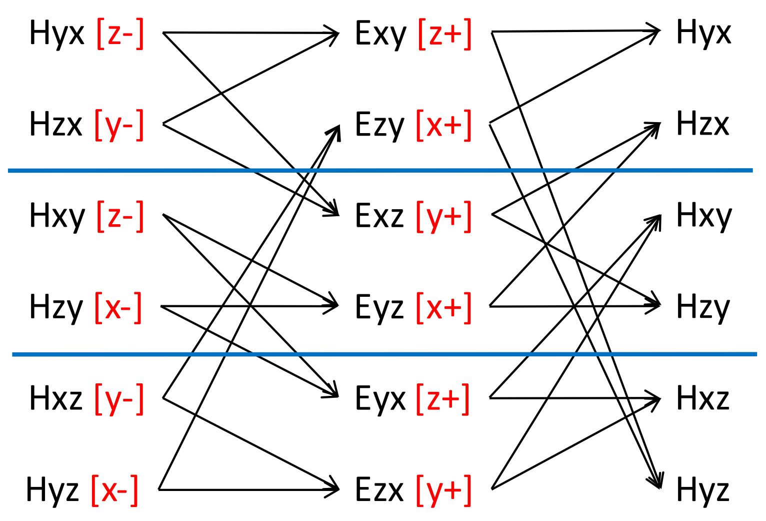


Figure 6.3: $\hat{\mathbf{H}}$ and $\hat{\mathbf{E}}$ field dependencies of the THIIM stencil kernel. Each field is updated by reading six domain-sized arrays of the other field. The arrows indicate dependencies over the same location in the grid and a unit index offset. The (red) labels in square brackets indicate the axis and the direction of the offset. The (blue) horizontal lines split the components in three regions to indicate the components update parallelism using three threads.

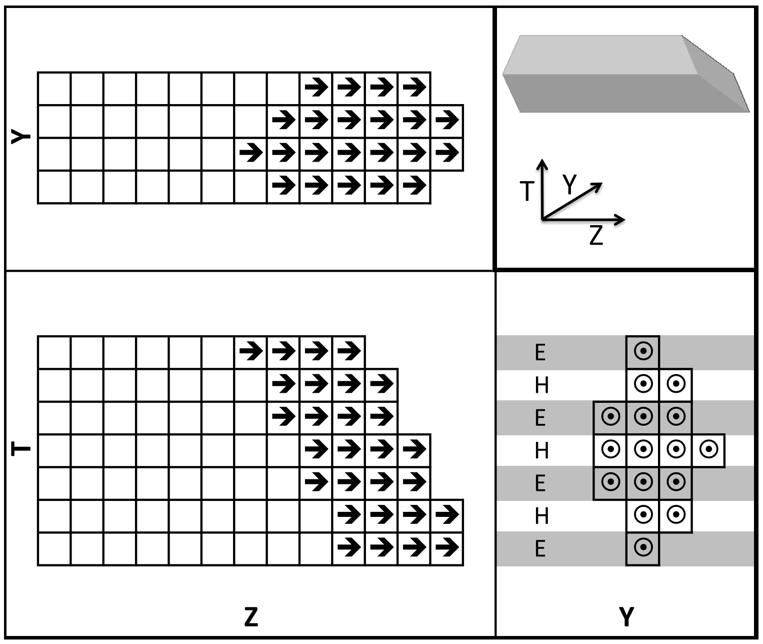


Figure 6.4: Extruded diamond tiling of the THIIM kernels, showing an example of $D_w = 4$ and $W_w = 4$. The data dependencies of the $\hat{\mathbf{H}}$ and $\hat{\mathbf{E}}$ fields allow more data reuse in the wavefront.

contributing grid points at the other end of the domain. Diamond tiling is suitable to apply periodic boundary conditions. We can use the leftmost half-diamond in Fig. 6.2 to complete the rightmost half-diamond. The rightmost new full-diamond will have data dependency with the leftmost diamond. We can ensure correct update order by using similar techniques to the distributed memory diamond tiling in Section 4.2.4, except that we perform memory data copy instead of MPI communication.

6.3 Detailed analysis of the solver's stencil codes

Here we analyze the data traffic requirements per lattice site update (i.e., the code balance) of the stencil code for the naïve, spatially blocked, and temporally blocked variants. The stencil codes of the solver are listed in Appendix D.

As described above, six components each are used for the electric field $\hat{\mathbf{E}}$ and the magnetic field $\hat{\mathbf{H}}$. We show the code of two component updates in the THIIM stencil in Listings 6.1 and 6.2. The remaining three and seven components updates have very similar memory access and computation patterns. The H_{XY} update in Listing 6.1 uses three coefficient arrays (`tHyx`, `cHyx`, `SrcHy`) and the H_{ZX} update in Listing 6.2 uses two coefficient arrays (`tHzx`, `cHzx`). Overall (i.e., considering all component updates) this results in $4 \cdot 3 + 8 \cdot 2 = 28$ domain-sized arrays for the coefficients. In total, $12 + 28 = 40$ domain-sized arrays have to be stored using double-complex numbers, leading to a storage requirement per grid cell of $16 \cdot 40\,\text{bytes} = 640\,\text{bytes}$ per grid cell.

6.3.1 Naïve kernel arithmetic intensity

We count the total floating-point operations per LUP in the stencil code. The loop nests in Listings 6.1 and 6.2 perform 22 flops and 20 flops, respectively. In total we count $4 \cdot 22 + 8 \cdot 20 = 248$ Double Precision (DP) flops/LUP. For calculating the

Listing 6.1: Kernel for the magnetic field H_{YX} component update. Accesses with index shifts along the outer dimension are highlighted. Similar computations and memory access patterns are performed in H_{XY}, E_{YX}, and E_{XY} updates.

```
for(k=zb; k<ze; k++) {
 for(j=yb; j<ye; j++) {
  ib=2*((k*Ny+j)*Nx+xb); ie=2*((k*Ny+j)*Nx+xe);
  for(i=ib; i<ie; i+=2) {
   ishift = i+2*(-Nx*Ny);
   Re=Exy[i]-Exy[ishift]+Exz[i]-Exz[ishift];
   Im=Exy[i+1]-Exy[ishift+1]+Exz[i+1]-Exz[ishift+1];
   t=Hyx[i]*tHyx[i]-Hyx[i+1]*tHyx[i+1]+SrcHy[i]
           -cHyx[i]*Re+cHyx[i+1]*Im;
   Hyx[i+1]=Hyx[i]*tHyx[i+1]+Hyx[i+1]*tHyx[i]
            +SrcHy[i+1]-cHyx[i]*Im-cHyx[i+1]*Re;
   Hyx[i] = t; }}}
```

data traffic we note that the loop in Listing 6.1 writes two double precision numbers, reads twelve numbers with no index shift, and reads four numbers with an outer dimension index shift (`ishift`). If we assume that all accesses to arrays with an outer dimension index shift (in Listing 6.1 these are `Exy` and `Exz`) actually go to main memory we have a total traffic of 18 double precision numbers in this loop. Whether this is true or not depends on the problem size: if two successive *x*-*y* layers of those grids fit into the cache, the shifted and non-shifted accesses to the same arrays come at half the data transfer cost because the access with the smaller index comes from cache. This reasoning is well known in stencil optimizations [40, 15]. At a problem size of 512^3 two layers take up $512^2 \cdot 16 \cdot 2 = 8\,\text{MiB}$ of cache *per thread and per array*, which exceeds the available cache size by far. See the next section on how this can be corrected.

The code in Listing 6.2 writes two numbers and reads ten numbers without large index shifts. The shifted accesses to `Exz` and `Exy` can be ignored since the shift is only along the middle dimension, and two rows of the data easily fit into some cache. The third variant of array updates is identical to the second in terms of data transfers since it has a very small shift of -2 along the inner dimension only.

Listing 6.2: Kernel for the magnetic field H_{ZX} component update. Accesses with index shifts along the middle dimension are highlighted. Similar computations and memory access patterns are performed in H_{ZY}, H_{XZ}, H_{YZ}, E_{XZ}, E_{YZ}, E_{ZX}, E_{ZY} updates. The offset direction in `ishift` variable differs in the components updates as presented in Fig. 6.3.

```
for(k=zb; k<ze; k++) {
 for(j=yb; j<ye; j++) {
  ib=2*((k*Ny+j)*Nx+xb); ie=2*((k*Ny+j)*Nx+xe);
  for(i=ib; i<ie; i+=2) {
   ishift = i+2*(-Nx);
   Re=Exy[ishift]-Exy[i]+Exz[ishift]-Exz[i];
   Im=Exy[ishift+1]-Exy[i+1]+Exz[ishift+1]-Exz[i+1];
   t = Hzx[i]*tHzx[i]-Hzx[i+1]*tHzx[i+1]
       -cHzx[i]*Re+cHzx[i+1]*Im;
   Hzx[i+1] = Hzx[i]*tHzx[i+1]+Hzx[i+1]*tHzx[i]
              -cHzx[i]*Im-cHzx[i+1]*Re;
   Hzx[i] = t; }}}
```

Overall we thus have a code balance of

$$B_{\mathrm{C}} = 4 \cdot (18 + 12 + 12) \cdot 8\,\text{bytes/LUP} = 1344\,\text{bytes/LUP}\ , \tag{6.8}$$

leading to an arithmetic intensity of $I = 248/1344\,\text{flops/byte} = 0.18\,\text{flops/byte}$, which results in very high pressure on the main memory bandwidth.

6.3.2 Spatial blocking arithmetic intensity

The total load/store operations to memory can be reduced by standard spatial blocking techniques, which establish “layer conditions” along the outer grid dimensions (see, for example, [15] and references therein). Spatial blocking results in a reduction of the memory traffic in the four loop nests that are structured as shown in Listing 6.1 by four double precision numbers each, if the blocking sizes in the inner and/or middle dimensions are chosen such that two successive layers of an array with index shifts in the outer dimension fit into a cache. The new code balance is thus

$$B_{\mathrm{C}} = 4 \cdot ([18 - 4] + 12 + 12) \cdot 8\,\text{bytes/LUP} = 1216\,\text{bytes/LUP}\ , \tag{6.9}$$

and the arithmetic intensity becomes $I = 248/1216\,\text{flops/byte} = 0.20\,\text{flops/byte}$. The spatial blocking optimization improves the performance of the code by a mere 10% because the main contributors to the data traffic are not the electric and magnetic fields but the coefficient arrays. Spatial blocking is not effective for these because they are accessed with no temporal locality.

We can now predict the maximum performance for optimal spatial blocking using a simple bottleneck model [41]: The limit due to the maximum memory bandwidth b_S of the CPU is $P_\mathrm{mem} = b_\mathrm{S}/B_\mathrm{C}$. The Haswell chip we used for our experiments has $b_\mathrm{S} \approx 50\,\text{GB/s}$, hence

$$P_\mathrm{mem} = \frac{b_\mathrm{S}}{B_\mathrm{C}} = \frac{50\,\text{GB/s}}{1216\,\text{bytes/LUP}} = 41\,\text{MLUP/s}\ . \tag{6.10}$$

This prediction is in very good agreement with the measurements. See Sect. 6.4 for details.

6.3.3 Diamond tiling arithmetic intensity and cache size requirements

The performance of temporal blocking techniques relies on a reduction of data traffic, especially to and from main memory. Data traffic models are very useful for understanding the expected or observed performance gains. We modify our cache block size model in Equation 3.6 to estimate the usable tile sizes in this application. We also use similar code balance model to Equation 3.8. The cache block size model estimates the maximum tile size that fits in the cache memory for this application by counting the working data set in the diamond-wavefront tile, such as the one shown in the y-z plane in Fig. 6.4. The total required number of bytes per tile is:

$$C_s = 16 \cdot N_x \cdot \left[40 \cdot \left(\frac{D_w^2}{2} + D_w \cdot (B_Z - 1) \right) + 12 \cdot (D_w + W_w) \right]\ . \tag{6.11}$$

Each point in the diamond-wavefront tile extends over the full length of the x dimension (N_x) with double-complex values ($8 \cdot 2$ bytes). The area of the wavefront-tile is $\frac{D_w^2}{2} + D_w \cdot (B_Z - 1)$, which depends on the diamond tile width (D_w) and the tile size along the z-axis (B_Z). Since each grid cell requires loading 12 components and 32 coefficients, we multiply the wavefront-diamond tile area by 40 numbers per grid cell. Finally, the $12 \cdot (D_w + W_w)$ part corresponds to the neighbor access of the 12 components around the wavefront-diamond tile, where the wavefront tile width $W_w = D_w + B_Z - 1$. For example, in Fig. 6.4 we have $D_w = 4$, $B_Z = 4$, and $W_w = 7$, so we have $C_s = 14912 \cdot N_x$ bytes per cache block.

For the code balance model we have to estimate the potential reduction of memory bandwidth pressure by temporal blocking. If the tile fits entirely in the L3 cache, the code loads each grid point once from main memory and stores it back only after completing the wavefront updates. We count the total reads and writes per diamond tile and divide by the diamond area (i.e., data reuse). Each diamond update consists of writing six $\hat{\mathbf{H}}$ field components per cell at full diamond width (D_w) and writing six $\hat{\mathbf{E}}$ field components per cell at $D_w - 1$. In total, each diamond requires $6 \cdot (2 \cdot D_w - 1)$ writes. The diamond tile requires reading 40 numbers per cell and accessing the neighbors of the 12 components $(40 \cdot D_w + 12)$. The diamond area is $(D_w^2/2)$. The code balance for double-complex numbers of the kernel is thus:

$$B_{\mathrm{C}} = \frac{16 \cdot [6 \cdot (2 \cdot D_w - 1) + (40 \cdot D_w + 12)]}{D_w^2/2} \frac{\text{bytes}}{\text{LUP}} \ . \tag{6.12}$$

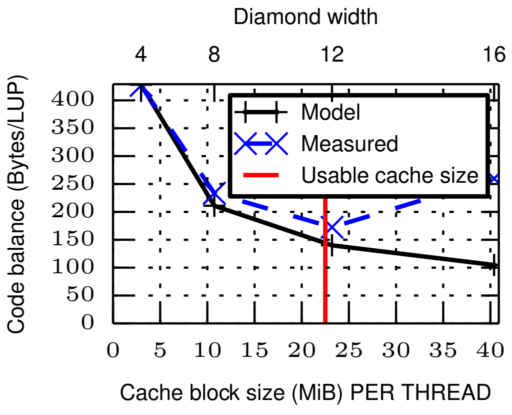
We validate our models and study the potential impact of our temporal blocking techniques using our 1WD implementation. Figure 6.5 shows the model predictions (solid black lines) of the code balance and cache block size and the code balance measurements (dashed blue lines). The latter is based on a direct measurement of the memory data traffic via hardware performance counters. We test four diamond tile widths (4, 8, 12, and 16). The red vertical lines indicate the estimated usable block size in the L3 cache of the Haswell processor (as a rule of thumb we assume that half the overall cache size, i.e., 22.5 MiB, is available for tile data). Figures 6.5a– 6.5c correspond to three wavefront width sizes (B_Z =1, 6, and 9), where more concurrency is achievable along the z-axis at the cost of using a larger cache block size. We perform our tests at a grid of size 480^3 in the 18-core Haswell processor using a single core and a single cache block.

The measurements show that the model accurately predicts the usable cache block size. The measured code balance diverges from the model when more than half of the L3 cache is used (the right side of the vertical red line), which is expected.

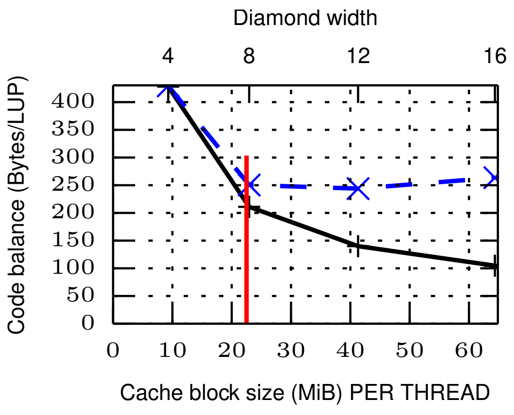
Our results also emphasize the importance of multi-dimensional intra-tile parallelism compared to parallelizing the wavefront only, where the maximum number of threads per tile is restricted by the wavefront tile width. Using $B_Z = 6$ would require three thread groups at the Haswell processor. As a result, the minimum diamond width $D_w = 4$ requires a cache block size $C_s = 30$ MiB, which exceeds the available cache memory. Although the cache block size of $B_Z = 9$ would fit in the L3 cache at $D_w = 4$, it cannot use larger diamond tiles, to enable more data reuse. On the other hand, our approach provides parallelism along the other dimensions without increasing the cache block size (i.e., it uses smaller wavefront tile widths), which saves space for larger diamond tiles. For example, we can set $B_Z = 1$ and use nine threads per cache block along the other dimensions. This setup provides a $D_w = 8$ that uses $C_s =$ 20MiB, allowing more data reuse within the usable cache block size limit.

6.4 Results

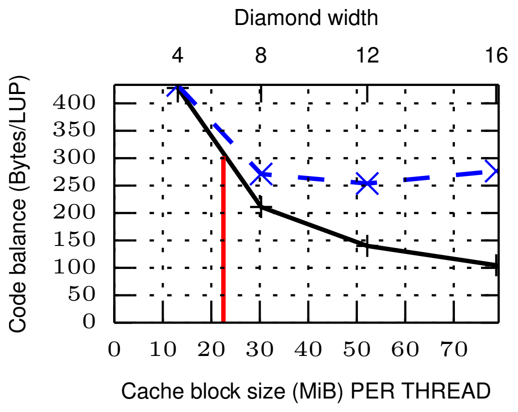
We present results for the spatially blocked code, 1WD, and MWD with full parameters auto-tuning to show the performance improvements. In order to get more insight



(a) $B_Z = 1$.



(b) $B_Z = 6$.



(c) $B_Z = 9$.

Figure 6.5: The cache block size requirements of the application's kernels at three wavefront widths (B_Z). We use an 18-core Haswell at grid size 480^3, running a single thread with the 1WD approach. Smaller wavefront tile widths, which provide less concurrency along the z-axis, enable more data reuse.

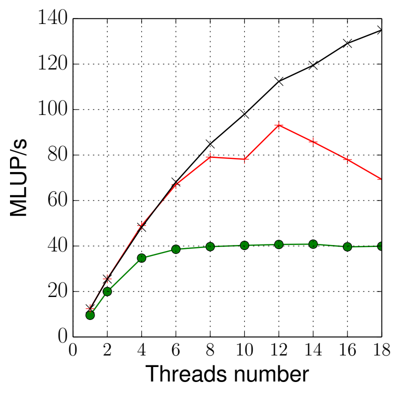
into performance properties we show thread scaling at fixed grid size and full socket performance at increasing grid size (cubic domain). Since 1WD generally performs better than PLUTO and Pochoir in the four stencil benchmarks in Chapter 5, we do not implement the solar stencil in their frameworks to compare their performance. We also present results using different thread group sizes to show the impact of the cache block sharing over the memory transfer volume and the memory bandwidth. All experiments are performed using the 18-core Haswell processor.

We use our Fixed-Execution to Data (FED) wavefront parallelization approach, which always assigns the same grid points to each thread while the wavefront traverses the tile. This idea maximizes the data reuse in the thread-private caches, since only boundary data instead of the full tile has to travel between threads. The corresponding performance improvement is very limited for simple stencils, but the THIIM stencil and high-order stencils do benefit from it.

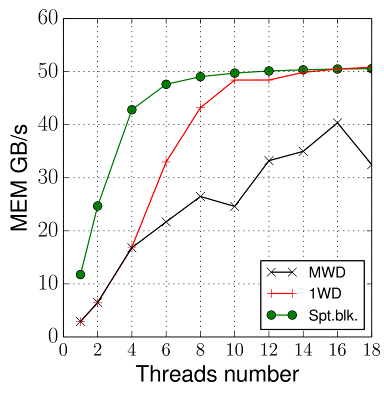
6.4.1 Thread scaling results

We present performance results of the THIIM kernel at increasing number of threads for a fixed problem size in Figure 6.6a. We also show the memory bandwidth measurements in Fig. 6.6b, measured code balance in Fig. 6.6c, and the auto-tuned MWD diamond width parameter in Fig. 6.6d.

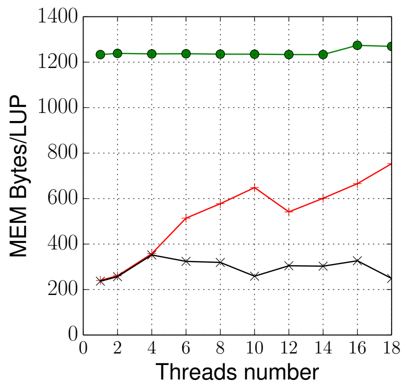
The spatially blocked code saturates the memory interface already with six cores, resulting in a performance of around 40 MLUP/s. This is in very good agreement with the bandwidth-based prediction of 41 MLUP/s that was derived in Sect. 6.3.2. Using separate cache blocks per thread in 1WD alleviates the memory bandwidth pressure and achieves better performance than spatial blocking code at smaller thread counts, but the cache is too small to accommodate sufficient blocks at larger thread counts so that a performance drop is observed beyond twelve cores. This can be seen more clearly in Fig. 6.6b: 1WD goes into bandwidth saturation at ten cores. In contrast,



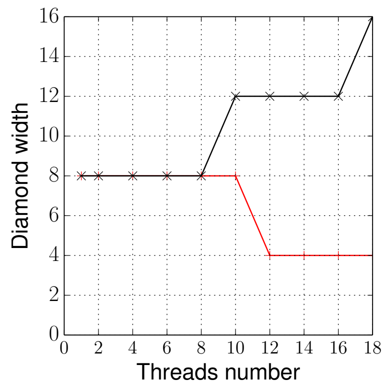
(a) Performance.



(b) Measured memory bandwidth usage.



(c) Measured memory transfers per LUP.



(d) Diamond width.

Figure 6.6: The THIIM stencil performance and memory transfer measurements, comparing 1WD, MWD, and spatially blocked code variants on an 18-core Haswell socket at increasing number of threads using grid size 384^3.

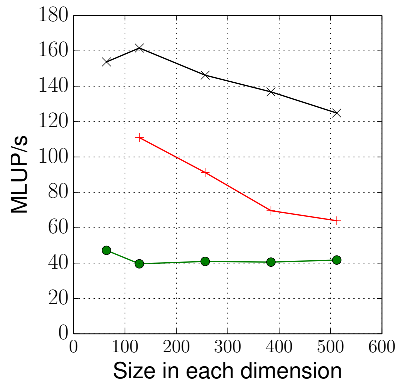
MWD does not saturate the memory bandwidth and can still profit from more cores up to the chip limit, showing a parallel efficiency of about 75% on the full chip. It can maintain a low code balance of 200–400 bytes/LUP for all thread counts (see Fig. 6.6c). The comparison of diamond width parameters selected by the autotuner in Fig. 6.6d is quite revealing: at larger core counts, 1WD requires smaller diamonds to meet the stringent cache size limit per core, whereas MWD can employ larger diamonds due to several threads sharing a diamond tile for wavefront updates.

6.4.2 Increasing grid size results

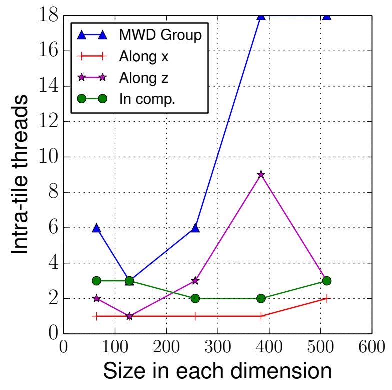
Although thread scaling, as shown in the previous section, reveals many interesting features of the 1WD and MWD algorithms, it is also instructive to study their behavior with changing problem size. We therefore present performance results of the THIIM kernel at different (cubic) grid sizes in Figure 6.7a, ranging from 64 to 512 with an increment of 64. We also show the auto-tuned MWD intra-tile parallelization parameters in Fig. 6.7b, the memory bandwidth measurements in Fig. 6.7c, and measured code balance in Fig. 6.7d.

1WD performance decays at larger grid size because of the increasing cache requirements as the leading dimension grows. The rise in the memory transfer volume seen in Fig. 6.7d suggests that the larger cache blocks cause more capacity misses in the L3 cache. Our auto-tuner selects a very small $D_W = 4$ at all grid sizes of 1WD, which already exceeds the available cache memory.

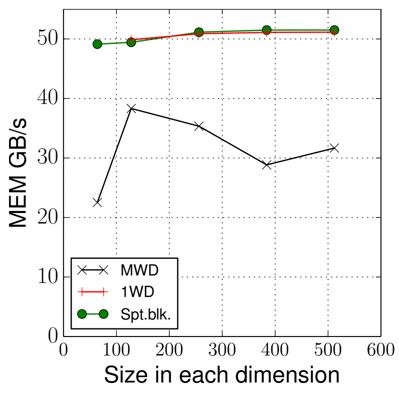
Our MWD implementation is decoupled from the memory bandwidth bottleneck over the full range of problem sizes. Compared to the spatially blocked code it has a $6\times$ lower code balance, resulting in a $3\times - 4\times$ speedup. The memory bandwidth measurements in Fig. 6.7c show that our approach is immune to even more memory bandwidth-starved situations, where the machine balance (ratio of memory bandwidth to computational performance) would be lower.



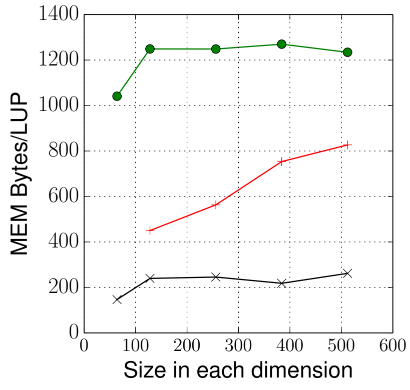
(a) Performance.



(b) Thread group size along x, z, and the field components. *in comp* refers to componentwise parallelism.



(c) Measured memory bandwidth usage.



(d) Measured memory transfers per LUP.

Figure 6.7: The THIIM stencil kernel performance and memory transfer measurements, comparing 1WD, MWD, and spatially blocked code variants on an 18-core Haswell socket at increasing cubic grid size.

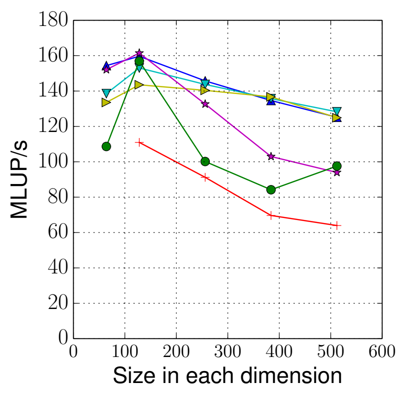
The auto-tuner selects larger thread groups as the grid size increases, as shown in Fig. 6.7b, to reduce the cache size requirements. This allows diamond widths in the range 8–16. For all grid sizes, two or three threads are used for the parallel components update. The components parallelism is a major contributor in reducing the cache block size requirements while maintaining high intra-tile concurrency. On the other hand, parallelizing the wavefront dimension alone would result in a larger cache block size, as described in Sect. 6.3.3.

6.4.3 Thread group size impact on performance and memory transfers

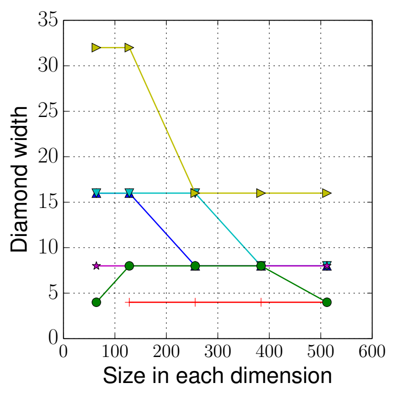
In this section we show the impact of the thread group size (i.e., cache block sharing) on the THIIM kernel performance in Figure 6.8a, on the memory bandwidth measurements in Fig. 6.8c, and on the code balance in Fig. 6.8d. We also show the tuned MWD diamond tile width in Fig. 6.8b.

The cases 6WD, 9WD, and 18WD are able to decouple from the memory bandwidth bottleneck at large grid sizes, allowing them to achieve similar performance. The small performance variations make the auto-tuner select different thread group sizes, as shown in the case of MWD performance at grid size of 512 in Figs. 6.7 and 6.8.

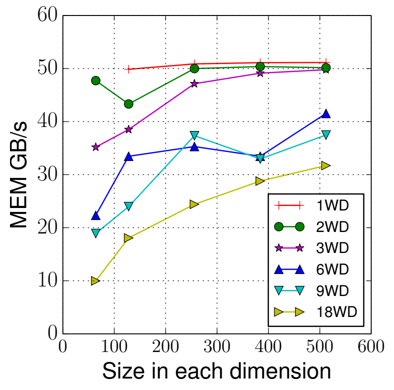
Larger thread group sizes reduce the need for cache size. As a result, increasing the thread group size allows the auto-tuner to select a larger diamond tile width, resulting in more in-cache data reuse, less memory bandwidth, and less memory transfer volumes. The 18WD version uses at least $D_W = 16$ at all grid sizes, as shown in Fig. 6.8b. The massive in-cache data reuse of 18WD results in saving more than 38% of the memory bandwidth at all grid sizes. On a CPU with smaller machine balance we expect an even more pronounced advantage of large thread group sizes.



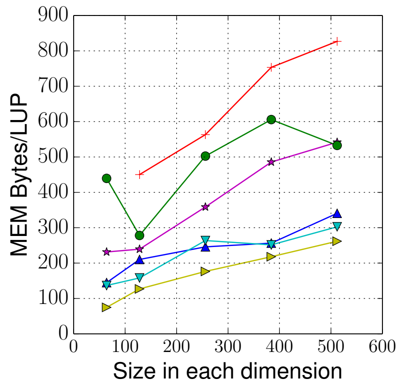
(a) Performance.



(b) Diamond width.



(c) Measured memory bandwidth usage.



(d) Measured memory transfers per LUP.

Figure 6.8: The THIIM stencil kernel performance and memory transfer measurements, comparing various thread group sizes in MWD on an 18-core Haswell socket at increasing cubic grid size. The results show the ability of our approach to reduce significantly the required memory bandwidth and transfer volumes.

6.5 Summary and future work

In this chapter, we applied multicore wavefront diamond temporal blocking with multi-dimensional intra-tile parallelization to a Maxwell's Equations solver used in a solar cell simulation application, achieving a $3\times - 4\times$ speedup and a 38% – 80% memory bandwidth saving. This stencil code has very low arithmetic intensity (0.20 flop/byte for optimal spatial blocking) and requires many bytes of storage per grid cell (640 bytes). Applying thread parallelism inside shared cache blocks as well as across electric and magnetic field components was decisive in lowering the severe cache size constraints of the code. Using a validated cache block size and code balance model we were able to describe the impact of the tiling parameters and the cache size on the memory traffic and thus limit the effort of the auto-tuner.

The design and optimization process of solar cells requires thousands of parallel runs of this code. In order to cover the whole visible wavelength spectrum for only a single solar cell configuration, about 80–160 simulations are needed. Our performance improvements reduce the turnaround time of each individual run and also the overall cost of the computations. We believe that our approach is applicable to many algorithms with similar characteristics, i.e., where the code has significant demand for memory bandwidth and cache size.

In the future we plan to investigate further the performance limitations within the core (in particular the SIMD vectorization) and the cache hierarchy, since the code runs at only about 5% of the theoretical peak performance of the CPU despite being cache bound. Hardware performance counter measurements and subsequent chip-level performance modeling will provide more insight here. The temporal blocking optimization will change the communication versus computation characteristics of the code, which also deserve an in-depth analysis.

Chapter 7

Related work

The importance of stencil computations and the inefficient performance of their naïve implementations on modern processors motivates researchers to study them extensively. The optimizations required to achieve the desired performance depend on the properties of the stencil operator and the capabilities of different resources in the processor. This case is made by Datta [42], where the performance of several combinations of optimization techniques, processors, and stencil operators is reported.

The high Bytes per LUP requirement of many stencil computations and the increasing performance gap between the arithmetic operations and the data transfer are the major concerns in achieving high performance. Spatial and temporal blocking improve the performance by increasing the data reuse in the cache memory of modern processors.

Spatial blocking is an established technique that changes the grid traversal order to maximize the data reuse in the desired memory level [4, 43]. Temporal blocking allows more data reuse in the cache memory by reordering grid traversal across space iterations, where blocks of grid points are accessed multiple times before completing the traversal of a single spatial grid level.

Temporal blocking techniques require careful handling of data dependencies across space iterations to ensure correctness. Several tiling techniques are proposed in the literature including: parallelogram, split, overlapped, diamond, and hexagonal. These block shapes optimize for data locality, concurrency, or both. Reviews of these techniques can be found at Orozco *et al.* [44] and Zhou [45]. Diamond tiling is promising for efficiently providing both concurrency and data locality over the problems and computer architectures of our interest. Its attractiveness in recent years is evident in: [44], [45], Strzodka *et al.* [7], Bandishti *et al.* [21], and Grosser *et al.* [46], where a GPU implementation of hexagonal tiling is proposed, then a study of hexagonal and diamond tiling is performed [47]. Jin *et al.* [48] performed parallelogram temporal blocking on Graphics Processing Units (GPUs) with domain sizes larger than the GPU memory. They pipeline domain chunks between the GPU and the CPU with optimization consideration to maximize performance and decrease the memory size footprint on the GPU.

The wavefront technique, which was introduced by Lamport [5] (using the name "hyperplane"), performs temporal blocking at adjacent grid points. This technique has been combined with other tiling approaches using single-thread wavefront temporal blocking as in [7], Wonnacott *et al.* [6], and Nguyen *et al.* [8], and using multi-threaded wavefront temporal blocking, as in Wellein *et al.* [9].

Cache optimization techniques can be classified into cache-oblivious and cache-aware techniques. Cache-oblivious techniques [49, 50, 24, 51] do not need prior knowledge or tuning to find optimal cache block size to achieve high performance stencil computations. On the other hand, cache-aware techniques utilize auto-tuning as in [42], which performs parameter search over the optimization techniques to achieve best performance, as defined in [52]: "using computer time rather than human time to search a space of code variations for a fixed problem". Another cache-aware algorithm is introduced in [7], where cache block size calculations are used to set the cache block size that achieves best performance.

Several frameworks have been developed to produce optimized stencil codes. Physis, a Domain Specific Language (DSL) framework that generates optimized GPU codes with the necessary MPI calls for heterogeneous GPU clusters, was proposed by Maruyama *et al.* [53]. PLUTO [23] is a source-to-source transformation tool that uses polyhedral model, CATS [7] is a library, Pochoir [24] uses cache-oblivious algorithms in Domain Specific Languages (DSL), PATUS [54] uses auto-tuning with a DSL, and Henretty *et al.* [55] develop a DSL that uses split-tiling. Unat *et al.* [56] introduced Mint, a programming model that produces highly optimized GPU code from a user's annotated traditional C code. Gysi *et al.* [57] introduced a stencil framework, named MODESTO, that uses models to decide on stencil transformations based on the program and the target architecture. A recent review paper of stencil optimization tools that use polyhedral model has been prepared by Wonnacott and Strout [58].

We classify the related work to our MWD approach in two categories: using separate cache block per thread and utilizing cache block sharing.

7.1 Related work using separate cache block per thread

We describe the temporal blocking issues in using dedicated cache block per thread in Section 3.4. Insufficient data reuse results in these approaches, given the LLC size limitation, having to use long strides in the leading dimension, and the increasing number of cores in contemporary processors.

Our work is close to the work of Nguyen *et al.* [8], the diamond tiling extension of Bandishti *et al.* [21] in PLUTO, and CATS2 algorithm of Strzodka *et al.* [7]. They combine wavefront temporal blocking with diamond tiling and parallelogram tiling in the case of [8]

Nguyen *et al.* [8] introduce a technique called 3.5D blocking. They perform wavefront blocking along the z-axis and parallelogram tiling along the y-axis. The whole domain is divided among the threads across the y-axis, where each iteration is advanced at once using a global barrier.

Our 1WD implementation is very similar to CATS2 and PLUTO's diamond tiling. While we use CATS2 tiling choices along each dimension, PLUTO performs diamond tiling along the z-axis, and parallelogram tiling along the y- and x-axes. 1WD is similar to PLUTO because one diamond tiles is scheduled to each thread, where a wavefront of short parallelogram tiles is updated in-order along the y-axis. The tiles along the x-axis are usually kept long, mostly at the length of the domain along the x-axis.

These three approaches have the advantage of efficiently utilizing the performance of multi-core processors. PLUTO and CATS2 require minimal thread synchronization and offer very efficient data reuse. However, we believe that two particular aspects make our work important.

First, PLUTO and CATS2 rely on a large domain size in the diamond/parallelogram tiling dimension to have sufficient concurrency for the available threads. If this condition cannot be met PLUTO does not utilize all the threads if diamond tiling is used. Strzodka proposes reverting to CATS1, which uses wavefront traversal in the same dimension of space-time parallelogram tiles. To extract parallelism for many-core processors, it is possible to use tiling approaches that allow concurrent update, for example, split-tiling, in more dimensions, but the code complexity would increase. Moreover, with the emergence of many-core architectures, it would be difficult to find sufficient concurrency for hundreds of threads in a reasonable grid size.

Second, no cache sharing among threads is assumed, so each thread requires space in the cache memory and bandwidth from main memory for its own use. As a result, memory bandwidth-starved stencil computations run out of cache and memory bandwidth, as will we show in Section 3.4. Moreover, it is unclear whether the cache size and memory bandwidth per thread as seen in contemporary multi-core designs will be available in future architectures. For example, the Intel Xeon Phi has 128KiB and 8KiB cache per thread in the L2 and L1 caches, respectively, and it achieves only about 3GB/s per core of memory bandwidth in full saturation.

Cache-oblivious stencil computations where introduced by Frigo *et al.* [50]. Strzodka *et al.* [51] introduced a parallel cache-oblivious approach that uses parallelograms to have the desired recursive tessellation. Tang *et al.* [24] introduced the Pochoir stencil compiler. Recursive trapezoidal tiling is used by performing space and time cuts. The algorithm was improved recently by Tang *et al.* in [59] by reducing the artificial data dependencies produced in previous works.

While these implementations use asymptotically optimal algorithms, the constant factor can be large, more than double compared to other tile shapes in one-dimension. Trapezoidal and parallelogram tiling are provably sub-optimal in maximizing the data reuse of loaded cache blocks compared to diamond tiling, as shown in [44], and compared to wavefront temporal blocking, as shown in this work. The need of using long strides along the leading dimension significantly increases the cache block size requirement, making practical implementations far from the ideal promise in the asymptotically optimal cache-oblivious algorithms.

To the best of our knowledge, utilizing cache block sharing among the thread does not work in cache-oblivious algorithms. For example, the performance of the selected thread group size is not only affected by the cache subsystem bandwidth and size, but also by the synchronization cost of the threads. As a result, cache block sharing techniques can achieve better performance as they use larger cache blocks than the optimal ones of dedicated cache block per thread, given the constraint of using long strides in the leading dimension.

7.2 Related work utilizing cache block sharing

The first work to utilize cache block sharing by multiple cores is proposed by Wellein *et al.* [9]. They use parallelogram tiling along the y-axis and multi-core wavefront temporal blocking along the z-axis. This work is extended by Wittmann *et al.* [10] and Treibig *et al.* [60]. They use relaxed synchronization in the thread group and assign one thread group per cache domain. To maintain the intermediate values across parallelogram tiles' update, data is copied to temporary storage to keep the intermediate time steps from the boundaries of the tiles.

Their work alleviates the cache capacity limitation that appears when using separate cache block per thread. It has the advantage of reducing the total cache memory requirement by using less number of cache blocks, while utilizing all the available threads. They also introduce the thread group concept, where each thread group share a tile. This thesis improves on the cache block sharing concept to achieve further cache block saving, more data reuse, more concurrency, flexibility with parameterized tiling, and auto-tuning.

The wavefront data is pipelined across the threads in Wellein *et al.* 's work, by assigning one or more time step for each thread. A space is imposed between the working set of each thread to achieve concurrency while ensuring correctness. This requires extending the cache block size in the spatial dimension, without increasing the in-cache data reuse. For example, the cache block size is doubled to achieve the same data reuse, compared to single thread wavefront variant, when a stencil radius of unity is used and a single time step is assigned per thread. This cost is significant when using variable-coefficient stencils, which load many bytes per grid cell. Since each thread updates data from different time steps, equal amount of work across the time steps of the wavefront is required. They achieve load-balancing by using parallelogram tiling in the other dimension. This does not allow their work to explode more advanced tiling techniques, such as diamond tiling.

A more recent cache block sharing work is proposed by Shrestha *et al.* [61]. They call the it Jagged tiling. Polyhedral model is utilized to generate code for intra-tile parallelization using PLUTO framework. This work is applied over one- and two-dimensional stencil computations. They use a multi-core wavefront of tiles, similar to our work along the z-axis, and use an optimized runtime system for fine grain parallelization to schedule work to threads. Intra-tile threads synchronization is performed through a dependency mask table of the size of the intra-tile tasks to track the ready-to-update work. When a thread takes a task, atomics are used to avoid race conditions. This work is extended by Shrestha *et al.* [62] They combined their introduced jagged tiling approach with the diamond tiling extension of PLUTO framework, to allow concurrent start at the inter- and intra-tile levels. The new approach is called fine-grain (FG) jagged polygon tiling. They show results for the three-dimensional 7-point constant-coefficient stencil. They run the experiments using grid of size 480^3, which is friendly to cache memory. The major difference in the structure of their tiling approach is that they provide intra-tile concurrency along the diamond tile dimension by stretching the diamond tile along the space dimension, i.e., without increasing the data reuse in time.

Shrestha's work has the advantage of targeting the polyhedral model in their cache block sharing algorithm. Their work is implemented in a general source-to-source transformation framework making it more generic and more usable. Their work is currently restricted to stencil computations, as diamond tiling in PLUTO works only with stencil computations.

By providing the diamond tiling and the intra-tile concurrency along the same dimension, they effectively revert to smaller diamond tile sizes and update groups of adjacent diamond tiles together. The only reuse in the thread group is at the boundary of the sub-tiles. On the other hand, our MWD approach allows the thread group to share one large diamond tile, allowing much more in-cache data reuse. Figure

5 of their paper [62] shows an example of their two-level tiling. The diamond tile is split into 9 sub-tile updates for fine grain parallelization. Using the same cache block size in space, our MWD approach allows for 15 sub-tile updates (i.e., 67% more data reuse) and our approach is not limited to thread group size of 3, as in their example. In other words, they compromises tile cache block size for more intra-tile concurrency. They also allow the intra-tile task to update more than one time step. This imposes unnecessary data dependencies across sub-tiles in the same row. Finally our work has the advantage of providing parameterized thread group size and using auto-tuning.

Chapter 8

Conclusion

Stencil computations are important kernels in PDE solver codes and in linear algebra kernels. These computations tend to be memory bound, resulting in low utilization of the compute power in contemporary processors. In this thesis, we propose novel temporal blocking algorithms that provide performance advantages over the state-of-the-art techniques in the literature. Our approach can also provide significant memory bandwidth savings that allows it to run in more memory bandwidth-starved systems.

We study the performance of corner-case stencil kernels (low-order with constant and variable coefficients stencils and high-order with constant and variable coefficients stencils) over a wide range of grid sizes using two contemporary Intel processors. Even with efficient spatial blocking techniques, most of the stencil kernels poorly utilize the compute resources in contemporary processors, due to the memory bandwidth limitations. We also demonstrate that using separate cache block per thread with advanced temporal blocking techniques (for example, CATS and cache-oblivious algorithms) is not sufficient to decouple the stencil computations from the main memory bandwidth bottleneck in many situations. This bandwidth saturation is more significant in high-order and variable-coefficient stencils than in other stencils, where the cache memory size becomes a more scarce resource. We also expect them to perform even more poorly in forecasted future processors, where the machine balance is expected to increase. We also demonstrate in our application chapter that existing cache block sharing techniques do not provide sufficient cache block size reduction. These techniques incur overhead in the cache block size that is proportional to the provided intra-tile concurrency.

In this thesis, we introduce a novel multi-dimensional intra-tile parallelization algorithm. Our approach has several advantages. It allows minimal or no overhead to the cache block size while maintaining high intra-tile concurrency, which is very advantageous for stencils with high bytes requirements. We introduce parallelism along the leading dimension, instead of tiling it, to reduce the working set in the private caches of the threads. The leading dimension parallelization allows better utilization of the hardware prefetching to the shared cache level. By making our intra-tile partitioning parallel to the time axis, we maximize the data reuse in the private caches of the threads. This tile partitioning allows us to perform hierarchical tiling; where the shared (large) tile resides in the shared cache level and the sub-tiles reside in the private caches of the threads.

We implement our algorithms in an open-source testbed framework, called *Girih*. Our implementation uses two advanced tiling techniques, diamond tiling and wavefront blocking. Our parameterized cache block sharing and tiling implementation provides a controllable trade-off between fine-grain synchronization and memory bandwidth usage, moving the pressure between the main memory interface and the CPU. Our implementation automatically tunes all of the parameters to maximize the performance benefit of our approach, by allowing architecture-friendly data access patterns for various setups. We provide a proof-of-concept distributed memory parallelization along the diamond tiling dimension. The MPI domain decomposition provides relaxed MPI communication and overlaps computation with communication.

We construct accurate models for the data transfer and cache size requirements of the wavefront diamond tiling approach. Our traffic model can predict the optimal code balance as a function of the stencil radius, the tile parameters, and the number of domain-sized streams. We have validated the models' predictions on two Intel processors by direct traffic measurements for the four studied stencil operators and the solar simulation stencil. The model is very accurate if the required cache block size (which is also predicted) fits into about half the shared outer level cache size. We utilize these models to use a model-guided automated tuning to reduce the auto-tuning work. The models are also important in motivating our work, where they reveal the cache and memory resources limitations in the other tiling approaches in the literature. Finally, the models provide an insight of the expected gain before implementing the MWD optimization to a given stencil code.

We compare our approach with other frameworks in two contemporary Intel processors using four stencil benchmarks over a wide range of grid sizes. Our MWD approach obtains a better performance than PLUTO, Pochoir, 1WD/CATS2, and an efficient spatially blocked code with most grid sizes. MWD is the only approach that constantly achieves better performance over spatial blocking, especially in the cases of high-order stencil computations. The experiments include hardware counters' measurements to have better understanding of the obtained performance. For example, the memory bandwidth measurements reveal the cases when the memory bandwidth is the bottleneck. By varying the thread group size, i.e., cache block sharing, in out experiments we show that our MWD approach can significantly save the memory bandwidth and memory transfers at the cost of slight performance degradation. On the Ivy Bridge processor, energy consumption reduction results from the memory bandwidth savings. We show cases where the "race to halt" concept does not apply all the time, where slower code with lower memory bandwidth usage can be more energy efficient than faster code running at the same CPU frequency. Finally, we present proof-of-concept distributed memory strong scaling performance results using one-dimension domain decomposition along the diamond tiling dimension. Our

MWD has scalable performance, up to a point when the surface-to-volume ratio of the communication-to-computation becomes significant.

We exploit our intra-tile parallelization techniques to improve the performance of the stencil code in a scientific application for solar cell simulations. Our MWD approach achieves $3\times - 4\times$ speedup over efficient spatial blocking implementation and saves $38\% - 80\%$ of the memory bandwidth on the 18-core Haswell processor. This application uses staggered grids and solves six equations in the simulation using FDFD discretization. The solar stencil has very low arithmetic intensity (0.22 flop/byte) and loads 640 bytes per grid cell, leading to high memory bandwidth saturation. To the extent of our knowledge, existing spatial and temporal blocking techniques are not efficient for this stencil, due to its very high data transfer requirements. Our approach minimizes the cache block size for the solar stencil by introducing intra-tile parallelization in the components and along the leading dimension of the tile, leading to the obtained performance improvements. The application requires special implementation of MWD to handle the staggered grid and multicomponent per grid cell. We also split the time step of the stencil kernel into two stages to maximize the temporal blocking data reuse. The runtime speedup of MWD in the solar stencil can result in a multi-fold reduction in solar cell design process time, as it requires running many simulations.

Our work shows that using separate cache block per thread may become deprecated in future processors, especially for memory bandwidth-starved kernels. On the other hand, our approach not only decouples the memory bandwidth bottleneck in stencil computations but also makes further savings in the memory bandwidth, making it suitable for processors with higher machine balance. Although we apply our techniques to stencil computations, they can be extended to applications that use other structures, such as unstructured grids.

Chapter 9

Future work and outlook

In this chapter, we discuss the applicability of our proposed approach in future HPC systems, which are expected to have deeper memory hierarchies and long vectorization units. We also discuss how the MWD approach can fit in the architecture of GPU accelerators. Stencil computations have many types and applications. We show how our method can handle other stencil types and particular application requirements, such as adaptive time stepping. We discuss a special variant of Krylov subspace solvers, which is an interesting application for stencil optimization frameworks. Our work can be integration with existing stencil frameworks, and it can utilize more optimization techniques, so we discuss these directions further.

9.1 Integrating MWD in future systems

Our MWD approach provides an efficient way to alleviate memory bandwidth-starved processors, where cache size per thread is not sufficient to hold a cache block that provides the required data reuse.

Future processors are anticipated to have deeper memory hierarchies [63] and long vectorization units. We discuss the integration of our MWD approach with other techniques to address these issues.

Figure 9.1 shows an example of potential future processors, where our MWD

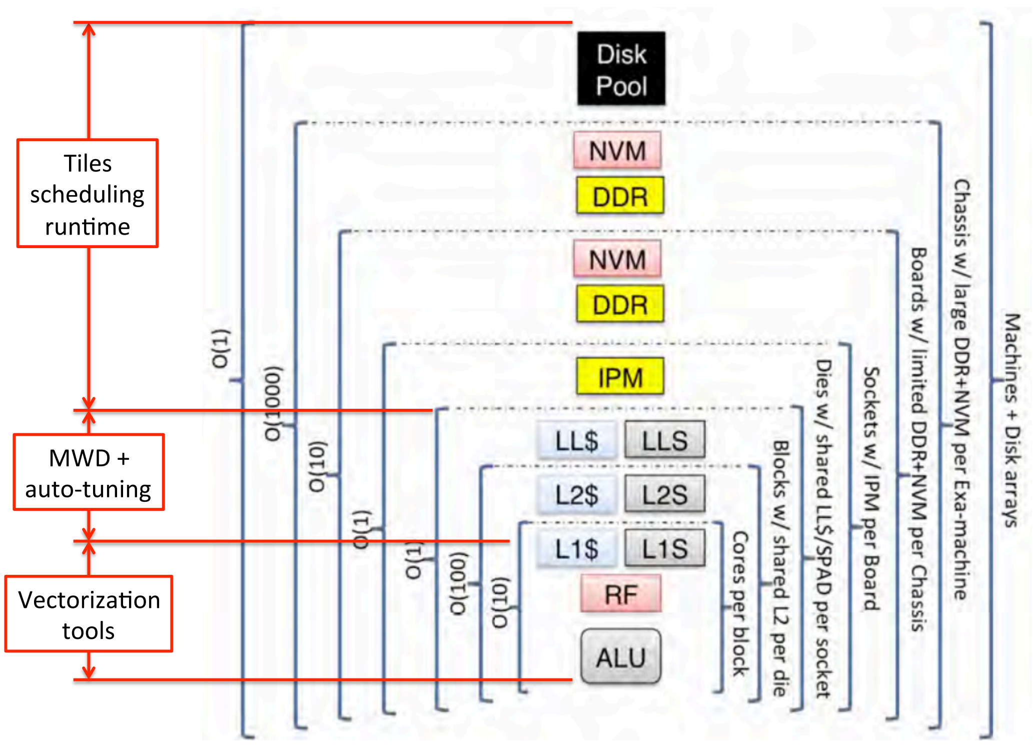


Figure 9.1: Outlook for integrating MWD with other techniques in future architectures. Figure with courtesy of Pete Beckman, Argonne National Laboratory.

approach can be used coherently with two approaches: 1) Scheduling runtime system of the MWD tiles can be used to perform hierarchical blocking of the larger and slower memory levels 2) Vectorization tools can utilize the long unit strides memory accesses provided by MWD tiles.

9.1.1 Handling deeper memory hierarchies with MWD

The runtime system would be invoked infrequently, as each MWD tile involves updating millions of LUPs. It may serialize (i.e., block) thread groups assignment to MWD blocks only for those requesting/completing new tiles. Our implementation shows negligible impact of this synchronization on the performance.

Cache oblivious techniques [49, 64, 24] are good candidates for the runtime system. They use space-filling-curves to provide automated hierarchical cache blocking for arbitrary memory levels with optimal asymptotic lower bound on the data transfer. As discussed in the 7, cache-oblivious algorithms face several challenges in utilizing the CPU and its nearby memory levels. On the other hand, utilizing them in the runtime system, at the granularity of our MWD tiles, does not affect these resources, as the smallest building block is large with architecture-friendly memory accesses.

The recursive tessellation nature of diamond tiles makes them good candidates for space-filling-curve algorithms, as shown in Fig. 9.2.

The extruded MWD tile may be split along the z-axis using parallelepiped-tiling to control the tile size and provide more concurrency. This would require using multi-dimensional space-filling-curve similar to the work of Tang *et al.* [24], with the difference of using diamond tiling along the y-axis, instead of split-tiling in all dimensions.

Other approaches may provide better performance with architectural and application considerations using priority queues with certain priority setup criteria. combined with other criterion, Last In First Out (LIFO) can improve data-locality across tiles

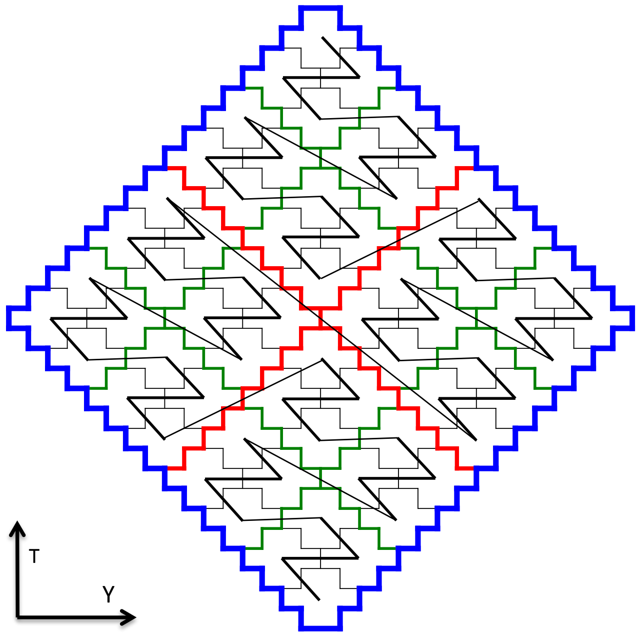


Figure 9.2: Utilizing Z-ordering space-filling-curves to visit diamond tiles hierarchically. Four tessellation levels are shown using black, green, red, and blue colors for the diamonds' boundaries of three tessellation levels. This recursive tessellation nature of diamond tiles make them good candidates for cache-oblivious algorithms

updates for the deeper memory hierarchy.

For example, it is shown in [65] that better performance can be achieved in Intel KNC many-core processor when the cores are assigned separate data without read sharing access. This suggests giving higher priority to non-adjacent extruded diamond tiles to be updated simultaneously by different cores. Ideally, updating the tiles in checkerboard fashion can guarantee no adjacent tiles updates by the cores. On the other hand, maximizing adjacent tiles updates is favorable in multi-core processors where uniform memory access is available at the shared level cache, as in the work of Wellein *et al.* [9].

9.1.2 Handling long vectorization units

In stencil computations, efficient utilization of long vector units usually require updating long unit-strides and performing register manipulation for adjacent stencil updates. The extruded diamond tiles in our MWD approach provides contiguous access of long strides along the x-axis. Moreover, our implementation provides array padding and aligned memory allocation to minimize vector loads across the cache lines from the L1 cache to the CPU. Contiguous and aligned memory accesses make our tiles friendly for efficient vectorization by the compiler and other vectorization techniques, as in [66, 67]. The long unit-strides also result in less loop and vectorization prologue and epilogue overhead.

9.2 Tiles software prefetching

We can utilize the saved memory bandwidth usage of our MWD approach to cover the potential latency in the main memory transfers. The software prefetching techniques can achieve this latency reduction at the cost of using more memory bandwidth.

Our MWD approach utilizes the hardware prefetching efficiently by allowing access to long contiguous strides of data. However, the hardware prefetching unit may not be able to bring all the required data from main memory to the LLC in the right time, which may be caused by the large number of data streams.

Since our approach successfully reduces the cache size and main memory bandwidth requirements, we can utilize the saving in these resources to perform software prefetching. The wavefront tiles in the extruded diamond tile are updated successively. A double-buffering technique can be utilized: while updating a given wavefront tile, use software-prefetching to load the data of the next wavefront tile. Since the data of successive wavefront tiles largely overlap, the ratio of loaded data to the performed update is usually minimal.

Prefetching distance is usually tuned to efficiently load and use the prefetched data in the right time. In our case, it might be sufficient to set up the code to prefetch the next block or two. The prefetching distance would be decided implicitly through the wavefront-diamond tile size in the cache memory, as the auto-tuner would select tile sizes that fill half the usable cache size to save the other half for prefetching.

9.3 Taking advantage of the memory bandwidth usage saving

High performance computing systems are expected to have significant power constraints in the future, as described in details in [68]. This may result in having insufficient supply to run the CPU and the DRAM at full power. Modern processors allow power capping in the DRAM and CPU resources. For example, contemporary Intel processors use RAPL power management interface. It estimates the power consumption and performs hardware power capping in both the CPU and the DRAM.

Our MWD approach significantly reduces the memory bandwidth requirements in memory bandwidth-bound stencil computations. As a result, the memory bandwidth interface is not fully utilized and the power consumption can be reduced in the DRAM. We propose to utilize this memory bandwidth usage savings to reduce the frequency of the memory interface and increase CPU frequency, given a particular power cap in the node. We expect to have performance improvements as the CPU is the bottleneck in our temporally blocked stencil computations.

This power control idea can be realized by including the power cap parameters of the CPU and the DRAM in our auto-tuning search space. The objective function of the current auto-tuning implementation aims to reduce time to solution. Multi-objective optimizations that accounts for memory transfer and power consumption are expected to make significant power savings, while maintaining near optimal time to solution. The current implementation of the auto-tuner utilizes our accurate cache block size model to prune the tile size parameter in the search space. We can also utilize our accurate memory transfer model to reduce the auto-tuning search space in the DRAM power capping (i.e., memory bandwidth) parameter.

The threads synchronization cost may be significant under certain conditions. Overlapped tiling techniques can mitigate this overhead by reducing memory transfers (hence synchronizations) at the expense of performing redundant computations. Therefore, the saved memory bandwidth may be used in this regard and compensate the overhead of the overlapped tiling cost.

9.4 Perspective on integration with accelerators

GPUs have been gaining more importance in HPC systems in recent years. Our generic framework is not only suitable for GPU architectures, but also the diamond tile shape allows decomposing the domain among CPUs and GPUs with relaxed synchronization scheme as we do in the distributed memory setup of the work. Moreover, the hierarchical shape of the diamonds (i.e., each diamond can be divided to 4 equal diamonds) allows setting different tile sizes for the CPUs and the GPUs while maintaining the tessellation of the subdomains. As for the update mechanism of the extruded diamond blocks in the GPU, each block can be updated by a Streaming Multiprocessor (SMX) of contemporary Nvidia GPUs. The threads of the SMX can be kept busy by exploiting the concurrency along the x-axis. In terms of cache block size, contemporary GPUs have sufficient cache memory to hold the wavefront data of small diamond tiles.

We present more ideas for using our MWD approach in GPU accelerators using NVIDIA GK110 Kepler as an example. The NVIDIA GK110 GPU [69] uses up to 15 SMX units, sharing 1536 KiB of L2 cache memory. Each SMX has 64 KiB of shared memory that can be configured as L1 cache (hardware-controlled) and shared memory (programmer-controlled). The hardware allows three L1/shared memory splits: 16k/48k, 32k/32k, and 48k/16k. Each SMX contains 192 single-precision

CUDA cores, 64 double-precision Floating-Point Units (FPUs), 65536 32-bit registers and is configured with 32 threads/warp.

NVIDIA Kepler provides synchronization mechanisms within the SMX, using the synchronization features of the Parallel Thread Execution (PTX) programming model. On the other hand, it may be impractical/difficult to perform fine-grain sharing across SMX units, as the hardware controls the scheduling order of the SMX units' work at runtime. This suggests assigning at least one extruded diamond tile per SMX. The shared memory of the SMX can be utilized to perform our intra-tile parallelization approach. This allows the threads of the SMX to maximize the data reuse in the shared memory and reduce the pressure on the global memory bandwidth.

Every 32 threads in a warp can be considered as a long Single Instruction Multiple Data (SIMD) unit, as they perform the same operation over multiple adjacent data elements. Since our MWD approach provides long unit-strides along the x-axis, it can efficiently utilize the warp's threads by assigning them contiguous 32-cells strides along x-axis.

Since the wavefront tiles in the extruded diamond are updated in-order, double-buffering technique can be used to reduce the memory latency. However, our relaxed synchronization scheme might be sufficient to cover the global memory latency, as the leading thread fetches new data while other threads update the shared in-cache data.

9.5 Handling other stencil types

We considered Jacobi-style updates in the stencil computation work of this thesis. Another important variant is the Gauss-Seidel-style, where the successive time iterations update the same solution array, as opposed to Jacobi-style iterations.

Gauss-Seidel-style schemes should work using our method in straightforward man-

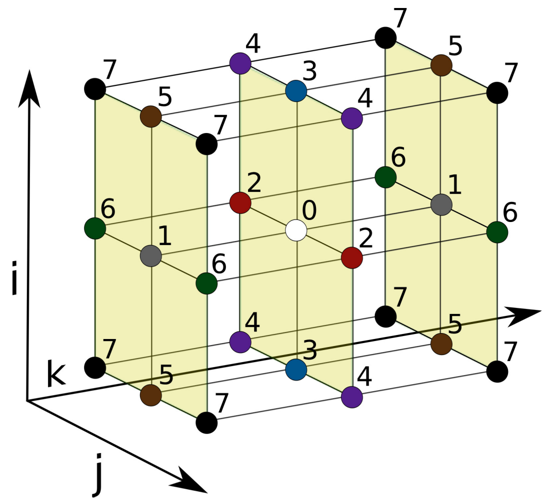


Figure 9.3: An example of box stencil operator, which extends diagonally to the axes

ner, as our method respects the data dependencies across time steps (i.e., iterations). However, the updates would not be ordered identically within each time step due to the spatial blocking performed by diamond tiling and the potential out-of-order scheduling of the diamond tiles in each row of diamonds. Using color-splitting like red-black Gauss-Seidel requires doubling the slope of the tiles (both the wavefront and the diamond) to respect the data dependency imposed by the red-black ordering in space. In contrast to regular Gauss-Seidel ordering, spatial blocking can obey the red-black ordering, as each point update has dependency over the direct neighbors in the same time step.

So far, star-shaped stencil operators are handled in our work, where the stencil operator extends along each axis separately. Other important stencil operators have shapes that extend diagonally, such as the 27-point box stencil operator shown in Fig. 9.3. The major difference in the application of these stencils is the diagonal data dependency of the stencil operator. The star stencil operator has data dependencies only over the faces of the tile and the subdomain, while the box stencil operator has additional dependencies over the corners and the edges. We use our MWD tiling techniques for a box stencil in our *Girih* framework. Our MWD implementation of the box stencil works because the tile shapes already account for data dependencies in the tile's corners and edges.

9.6 Integrating intra-tile parallelization techniques in stencil frameworks

Our MWD can be integrated in cache-aware stencil optimization techniques that perform explicit tiling. Several well established frameworks, such as PLUTO and Physis, use these techniques. For example, the tiled stencil codes of PLUTO can incorporate our techniques. PLUTO has options to perform diamond tiling and parallelepiped tiling along different dimensions. Concurrent start is achieved by scheduling diamond tiles to threads using OpenMP. It also has control over the tiles scheduling scheme. It may be possible to configure PLUTO to perform diamond tiling along the y-axis and small parallelepipled tiles along the z-axis. The wavefront-diamond tiling can be achieved by updating consecutive tiles along the z-axis. These tiles can be then parallelized using our thread group concept through nested OpenMP parallelization.

OpenMP does not provide straightforward mechanism to perform the intra-tile parallelization and synchronization. To avoid the complexity of using pthread libraries, it is possible to use the "phasers" introduced by Shirako *et al.* [70]. They proposed data structures and interface to provide threads point-to-point and subgroup synchronization that are suitable for parallelizing stencil computations.

9.7 Understanding MWD behavior in the cache subsystem

Our MWD approach removes the memory bandwidth bottleneck in memory bandwidth-starved stencil computations. Due to the data access pattern complexity of MWD, it is unclear how it utilizes the resources in the core and the cache subsystem. It would be interesting to have better understanding of MWD bottlenecks to investigate performance improvements opportunities. Cache simulators, for example, can be used to have better understanding of MWD intra-cache behavior. Gaining more insight about MWD can also benefit the hardware and software co-design for future processors. For example, how would MWD behave in the future architectures if they have contention over some resources and large latency issues.

9.8 Handling adaptive time stepping of PDE solvers with MWD

Adaptive time stepping is used in explicit PDE solvers when the maximum wave speed in the solution domain is not known *a priori* at the time of the simulation, or may vary significantly during the iterations of the solver. The Courant-Friedrichs-Lewy (CFL) condition [71] (the English translation of the paper [72]) determines the time step size limit to achieve correct convergence in the PDE solver. At any given time step (i.e., iteration), the ratio of the spatial to temporal discretization size ($\delta x/\delta t$) has to be smaller than the maximum wave speed in the solution domain. Otherwise, the solver is vulnerable to numerical instability and will not converge to the correct solution.

One of the solutions to this problem is to check the maximum wave speed after each iteration. If the maximum wave speed violates the CFL condition, the solver reverts to the solution domain of the previous iteration and repeat using smaller time step size that obeys the CFL condition.

Selecting very small time steps reduces the possibility of repeating iterations, but requires more iterations to arrive to the desired solution. On the other hand larger time steps increases the probability of violating the CFL condition that increases the time to solution. The “sweet spot” is application and data dependent.

Temporal blocking approaches advance the solution domain several iterations at once, which pose challenges to adaptive time stepping approach. Violating the CFL condition results in reverting multiple time steps, wasting more resource and time compared to naïve data update order. Moreover, regularly checkpointing at correct solution domain iterations is important. This is challenging in temporal blocking approaches that advance many time steps in some parts of the solution domain, such as the cache-oblivious algorithms.

Our MWD can be modified to handle adaptive time stepping. The most suitable place for checkpointing is the middle of diamond tiles, where the whole solution domain can be stored at one iteration. This requires modifying the wavefront approach to store the middle time step of certain diamond rows in separate arrays. The cost of reverting when the CFL condition is violated can be minimized. The runtime tile scheduler can be modified to increase the priority of tiles at earlier time steps and decrease the priority of tiles at later time steps. This would decrease the range of the updated time steps, hence decrease the number of reverted time steps when the CFL condition is violated.

It is possible to make further reduction in the cost of reverting to correct time iteration. During the extruded diamond update, the solver tracks the maximum wave speed after the wavefront update. If the CFL condition is violated at any point, all the threads halt their operation and the runtime reverts to the checkpoint using the updated time step size. This reduces the cost of waiting for complete domain update before reverting.

The same checkpointing for adaptive time stepping can be sued to handle system failure recovery. This requires storing the data of the checkpoint in a non-volatile storage for recovery.

9.9 Krylov subspace solvers, a promising applications for MWD

A particular variant of iterative Krylov subspace solvers uses expanded stencil operations in place of a series of individual SpMV operations [73]. The motivation for these pipelined methods is synchronization overhead reduction in distributed memory not cache efficiency in shared memory, but the interaction must be exploited for emerging hybrid programming environments. Krylov solvers use expanded stencil operations through either polynomial pre-conditioners or s-step methods. This approach is promising in the development of extreme scale Krylov solvers. The benefit of our MWD approach can be used to improve the intra-node performance of these approaches.

9.10 Transferring temporal order derivatives to spatial order derivatives in finite difference PDE solvers

PDE solvers with high-order temporal derivatives require reading from several domain-sized arrays for each grid point update, which come from several time steps in past iterations. These solvers have high memory size and high memory bandwidth requirements.

This memory pressure issue can be alleviated by using an algorithmic technique that transfers the high temporal order derivative to a high spatial order derivative in the finite-difference time-domain PDE solvers. This technique is presented in [74] for Reverse Time Migration (RTM) applications, where higher-order temporal derivatives are replaced with multiple evaluations of the Laplacian operator.

Combining these algorithmic techniques with our efficient MWD approach can improve the performance of these applications.

9.11 Handling thin domains in three-dimensional grids

The benchmarks and the application in this thesis use a cubic domain shape. In many applications, from climate models to reservoir models, etc., one dimension is significantly smaller than the other two, i.e., the domain is "thin." Our approach can benefit such applications significantly: Mapping the thin dimension to the leading array dimension helps both, tiling in shared memory and domain decomposition in distributed memory setups. For shared memory, we show in Eq. 3.6 that the cache block size is proportional to the leading dimension size, so we can use larger blocks in time with more data reuse. Although tiling a long leading dimension can also reduce the cache block size, it increases the pressure on the TLB and may lead to inefficient hardware data prefetching [15]. In distributed memory, decomposing the leading dimension is usually the most expensive, as the halo layer is not contiguous in memory. Thin domains reduce the requirement of decomposing the leading dimension while maintaining a favorable surface-to-volume ratio per subdomain. It is worth mentioning that very short leading dimensions (i.e., thin domains with less than about 50 cells) are inefficient because of bad pipeline utilization. This effect is amplified by long SIMD units, which lead to even shorter loop lengths and slow (scalar) remainder loops. In this situation the thin domain should be mapped to the middle or outer dimensions.

REFERENCES

[1] W. Schönauer, *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers.* Self-edition, 2000, http://www.rz.uni-karlsruhe.de/~rx03/book.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[3] F. Ortigosa, M. A. Polo, F. Rubio, J. Cela, R. de la Cruz, M. Hanzich *et al.*, "Evaluation of 3D RTM on HPC platforms," in *2008 SEG Annual Meeting.* Society of Exploration Geophysicists, 2008.

[4] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, 2009.

[5] L. Lamport, "The parallel execution of DO loops," *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974. [Online]. Available: http://doi.acm.org/10.1145/360827.360844

[6] D. G. Wonnacott, "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," in *International Parallel and Distributed Processing Symposium*, 2000, pp. 171–180.

[7] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, "Cache accurate time skewing in iterative stencil computations," in *Proceedings of the International Conference on Parallel Processing.* IEEE Computer Society, Sep. 2011, pp. 571–581.

[8] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, “3.5-D blocking optimization for stencil computations on modern CPUs and GPUs,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–13.

[9] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, “Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization,” in *Computer Software and Applications Conference. 33rd Annual IEEE International*, vol. 1, July 2009, pp. 579–586.

[10] M. Wittmann, G. Hager, J. Treibig, and G. Wellein, “Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters,” *Parallel Processing Letters*, vol. 20, no. 04, pp. 359–376, 2010. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626410000296

[11] D. Orozco and G. Gao, “Mapping the FDTD application to many-core chip architectures,” in *Proceedings of the International Conference on Parallel Processing*, Sept 2009, pp. 309–316.

[12] H. T. Kung, “Memory requirements for balanced computer architectures,” in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA ’86. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 49–54, dOI: 10.1145/17356.17362. [Online]. Available: http://dl.acm.org/citation.cfm?id=17407.17362

[13] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[14] G. Hager, J. Treibig, J. Habich, and G. Wellein, “Exploring performance and power properties of modern multi-core chips via simple machine models,” *Concurrency and Computation: Practice and Experience*, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3180

[15] H. Stengel, J. Treibig, G. Hager, and G. Wellein, “Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory Model,”

Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 207–216, 2015. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751240

[16] J. D. McCalpin, “STREAM: Sustainable memory bandwidth in high performance computers,” University of Virginia, Charlottesville, VA, Tech. Rep., 1991-2007, a continually updated technical report. [Online]. Available: http://www.cs.virginia.edu/stream/

[17] ——, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1995.

[18] J. Treibig, G. Hager, and G. Wellein, “LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments,” in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on.* IEEE, 2010, pp. 207–216.

[19] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann, “Power-management architecture of the Intel microarchitecture code-named Sandy Bridge,” *IEEE Micro*, vol. 32, pp. 20–27, 2012.

[20] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, 2010.

[21] V. Bandishti, I. Pananilath, and U. Bondhugula, “Tiling stencil computations to maximize parallelism,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2012, pp. 1–11.

[22] “Girih stencil optimization framework,” https://github.com/tareqmalas/girih.

[23] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, 2008.

[24] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, “The Pochoir stencil compiler,” in *Proceedings of the Twenty-third*

Annual ACM Symposium on Parallelism in Algorithms and Architectures. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508

[25] J. Hennessy, D. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann/Elsevier, 2012, pp. 26. [Online]. Available: http://books.google.com.sa/books?id=v3-1hVwHnHwC

[26] C. Pflaum and Z. Rahimi, "An iterative solver for the finite-difference frequency-domain (FDFD) method for the simulation of materials with negative permittivity," *Numerical Linear Algebra with Applications*, vol. 18, no. 4, pp. 653–670, 2011. [Online]. Available: http://dx.doi.org/10.1002/nla.746

[27] M. A. Green, K. Emery, Y. Hishikawa, W. Warta, and E. D. Dunlop, "Solar cell efficiency tables (version 46)," *Progress in Photovoltaics: Research and Applications*, vol. 23, no. 7, pp. 805–812, 2015. [Online]. Available: http://dx.doi.org/10.1002/pip.2637

[28] C. Pflaum and Z. Rahimi, "An iterative solver for the finite-difference frequency-domain (FDFD) method for the simulation of materials with negative permittivity," *Numerical Linear Algebra with Applications*, vol. 18, no. 4, pp. 653–670, 2011. [Online]. Available: http://dx.doi.org/10.1002/nla.746

[29] K. Yee, "Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media," *Antennas and Propagation, IEEE Transactions on*, vol. 14, no. 3, pp. 302–307, May 1966.

[30] R. Luebbers, F. Hunsberger, K. S. Kunz, R. Standler, and M. Schneider, "A frequency-dependent finite-difference time-domain formulation for dispersive materials," *Electromagnetic Compatibility, IEEE Transactions on*, vol. 32, no. 3, pp. 222–227, Aug 1990.

[31] D. Kelley and R. Luebbers, "Piecewise linear recursive convolution for dispersive media using FDTD," *Antennas and Propagation, IEEE Transactions on*, vol. 44, no. 6, pp. 792–797, Jun 1996.

[32] O. Gandhi, B.-Q. Gao, and J.-Y. Chen, "A frequency-dependent finite-difference time-domain formulation for general dispersive media," *Microwave Theory and Techniques, IEEE Transactions on*, vol. 41, no. 4, pp. 658–665, Apr 1993.

[33] D. M. Sullivan, "Frequency-dependent FDTD methods using Z transforms," *Antennas and Propagation, IEEE Transactions on*, vol. 40, no. 10, pp. 1223–1230, Oct 1992.

[34] Z. Rahimi and C. Pflaum, "Studying the effect of scattering layers on the efficiency of thin film solar cells," in *Numerical Simulation of Optoelectronic Devices (NUSOD), 2014 14th International Conference on*, Sept 2014, pp. 169–170.

[35] C. Pflaum, Z. Rahimi, and C. Jandl, "Simulation of optical waves in thin-film solar cells," in *Electromagnetics in Advanced Applications (ICEAA), 2010 International Conference on*, Sept 2010, pp. 24–26.

[36] S. Yan, J. Krantz, K. Forberich, C. Pflaum, and C. J. Brabec, "Numerical simulation of light propagation in silver nanowire films using time-harmonic inverse iterative method," *Journal of Applied Physics*, vol. 113, no. 15, pp. –, 2013. [Online]. Available: http://scitation.aip.org/content/aip/journal/jap/113/15/10.1063/1.4801919

[37] J.-P. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 114, no. 2, pp. 185 – 200, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999184711594

[38] Z. Rahimi, A. Erdmann, and C. Pflaum, "Finite integration (FI) method for modelling optical waves in lithography masks," in *Electromagnetics in Advanced Applications, 2009. ICEAA '09. International Conference on*, Sept 2009, pp. 809–812.

[39] S. Geißendörfer, M. Theuring, T. Titz, S. Mogck, C. Pflaum, B. Abebe, F. Schütze, D. Wynands, U. Kirstein, A. Schweitzer, V. Steenhoff, A. Neumüller, K. Borzutzki, R.-E. Nowak, A. Philipp, P. Klement, O. Sergeev, M. Vehse, and K. von Maydell, "The SiSoFlex project: Silicon based thin-film solar cells on

flexible aluminium substrates," in *29th European Photovoltaic Solar Energy Conference and Exhibition*, 2014, pp. 1667–1670.

[40] C. Leopold, "Tight bounds on capacity misses for 3D stencil codes," in *Computational Science – ICCS 2002*, ser. Lecture Notes in Computer Science, P. Sloot, A. Hoekstra, C. Tan, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2002, vol. 2329, pp. 843–852.

[41] R. W. Hockney and I. J. Curington, "$f_{1/2}$: A parameter to characterize memory and communication bottlenecks," *Parallel Computing*, vol. 10, no. 3, pp. 277–286, 1989.

[42] K. Datta, "Auto-tuning stencil codes for cache-based multicore platforms," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-177.html

[43] H. Dursun, M. Kunaseth, K. Nomura, J. Chame, R. F. Lucas, C. Chen, M. Hall, R. K. Kalia, A. Nakano, and P. Vashishta, "Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters," *The Journal of Supercomputing*, vol. 62, no. 2, pp. 946–966, 2012.

[44] D. Orozco, E. Garcia, and G. Gao, "Locality optimization of stencil applications using data dependency graphs," in *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2011, pp. 77–91.

[45] X. Zhou, "Tiling optimizations for stencil computations," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2013. [Online]. Available: http://polaris.cs.uiuc.edu/~zhou53/papers/Xing_Zhou.pdf

[46] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for GPUs," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 66.

[47] T. Grosser, S. Verdoolaege, A. Cohen, and P. Sadayappan, "The relation between diamond tiling and hexagonal tiling," *Parallel Processing*

Letters, vol. 24, no. 03, p. 1441002, 2014. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626414410023

[48] G. Jin, T. Endo, and S. Matsuoka, "A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of gpu," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1080–1087. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2013.58

[49] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on.* IEEE, 1999, pp. 285–297.

[50] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th Annual International Conference on Supercomputing.* New York, NY, USA: ACM, 2005, pp. 361–366. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088197

[51] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, "Cache oblivious parallelograms in iterative stencil computations," in *Proceedings of the 24th ACM International Conference on Supercomputing.* New York, NY, USA: ACM, 2010, pp. 49–59. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810096

[52] J. Demmel, S. Williams, and K. Yelick, "Automatic performance tuning (autotuning)," in *The Berkeley Par Lab: Progress in the Parallel Computing Landscape*, M. W. D. Patterson, D. Gannon, Ed. Microsoft Research, 2013, p. 337.

[53] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 2011, pp. 1–12.

[54] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *International Parallel and Distributed Processing Symposium*, May 2011, pp. 676–687.

[55] T. Henretty, R. Veras, F. Franchetti, L. N. Pouchet, J. Ramanujam, and P. Sadayappan, “A stencil compiler for short-vector SIMD architectures,” in *Proceedings of the 27th international ACM conference on supercomputing.* ACM, 2013, pp. 13–24.

[56] D. Unat, X. Cai, and S. B. Baden, “Mint: realizing CUDA performance in 3D stencil methods with annotated C,” in *Proceedings of the international conference on Supercomputing.* ACM, 2011, pp. 214–224.

[57] T. Gysi, T. Grosser, and T. Hoefler, “MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures.” ACM, Jun. 2015, accepted at ACM International Conference on Supercomputing (ICS’15).

[58] D. G. Wonnacott and M. M. Strout, “On the scalability of loop tiling techniques,” in *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, Berlin, 2013, pp. 3–11.

[59] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury, “Cache-oblivious wavefront: Improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency,” in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 205–214. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688514

[60] J. Treibig, G. Wellein, and G. Hager, “Efficient multicore-aware parallelization strategies for iterative stencil computations,” *Journal of Computational Science*, vol. 2, no. 2, pp. 130–137, 2011, simulation Software for Supercomputers.

[61] S. Shrestha, J. Manzano, A. Marquez, J. Feo, and G. R. Gao, “Jagged tiling for intra-tile parallelism and fine-grain multithreading,” in *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing, Hillsboro, OR, USA*, 2014.

[62] S. Shrestha, G. R. Gao, J. Manzano, A. Marquez, and J. Feo, “Locality aware concurrent start for stencil applications,” in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser.

CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 157–166. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738620

[63] A. Suresh, P. Cicotti, and L. Carrington, "Evaluation of emerging memory technologies for HPC, data intensive applications," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, Sept 2014, pp. 239–247.

[64] M. Frigo and V. Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 361–366. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088197

[65] D. Ernst, "Stencil codes on Intels Xeon Phi," Master's thesis, University of Erlangen-Nuremberg, 2014.

[66] T. Malas, A. J. Ahmadia, J. Brown, J. A. Gunnels, and D. E. Keyes, "Optimizing the performance of streaming numerical kernels on the IBM Blue Gene/P PowerPC 450 processor," *International Journal of High Performance Computing Applications*, 2012. [Online]. Available: http://hpc.sagepub.com/content/early/2012/05/17/1094342012444795.abstract

[67] D. Caballero, S. Royuela, R. Ferrer, A. Duran, and X. Martorell, "Optimizing overlapped memory accesses in user-directed vectorization," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15, 2015, pp. 393–404. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751224

[68] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, M. Kondo, and I. Miyoshi, "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 78:1–78:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807638

[69] NVIDIA, "Kepler GK110 whitepaper," 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[70] J. Shirako, K. Sharma, and V. Sarkar, "Unifying barrier and point-to-point synchronization in openmp with phasers," in *OpenMP in the Petascale Era*, ser. Lecture Notes in Computer Science, B. Chapman, W. Gropp, K. Kumaran, and M. Mller, Eds. Springer Berlin Heidelberg, 2011, vol. 6665, pp. 122–137. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21487-5_10

[71] R. Courant, K. Friedrichs, and H. Lewy, "Uber die partiellen differenzengleichungen der mathematischen physik," *Mathematische Annalen*, vol. 100, no. 1, pp. 32–74, 1928. [Online]. Available: http://dx.doi.org/10.1007/BF01448839

[72] ——, "On the partial difference equations of mathematical physics," *IBM Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, March 1967.

[73] W. Vanroose, P. Ghysels, D. Roose, and K. Meerbergen, "Hiding global communication latency and increasing the arithmetic intensity in extreme-scale Krylov solvers," *Examath position paper*, 2013.

[74] E. Dussaud, W. W. Symes, P. Williamson, L. Lemaistre, P. Singer, B. Denel, and A. Cherrett, "Computational strategies for reverse-time migration," in *Seg technical program expanded abstracts 2008*. Society of Exploration Geophysicists, 2008, pp. 2267–2271.

[75] R. J. LeVeque, *Finite volume methods for hyperbolic problems.* Cambridge university press, 2002, vol. 31.

[76] D. I. Ketcheson, M. Parsani, and R. J. LeVeque, "High-order wave propagation algorithms for hyperbolic systems," *SIAM Journal on Scientific Computing*, vol. 35, no. 1, pp. A351–A377, 2013.

APPENDICES

A Relevant papers submitted and in preparation

- Tareq Malas, Georg Hager, Hatem Ltaief, and David Keyes, “Multi-dimensional intra-tile parallelization for memory-starved stencil computations.” In preparation.

- Tareq Malas, Julian Hornich, Georg Hager, Hatem Ltaief, Christoph Pflaum, and David Keyes, “Optimization of an electromagnetics code with multicore wavefront diamond blocking and multi-dimensional intra-tile parallelization.” In preparation.

- Tareq Malas, Georg Hager, Hatem Ltaief, and David Keyes. “Advanced tiling techniques for memory-starved streaming numerical kernels,” Technical Poster Session in Supercomputing 2015.

- Tareq Malas, Georg Hager, Hatem Ltaief, and David Keyes.“Towards fast reverse time migration kernels using multi-threaded wavefront diamond tiling,” The Second EAGE Workshop on High Performance Computing for Upstream in Dubai, UAE. September 2015

- Tareq Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and David Keyes. “Multicore-optimized wavefront diamond blocking for optimizing stencil updates.” *SIAM Journal on Scientific Computing*, 37(4):439-464, 2015

- Tareq Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and

David Keyes. "Optimizing Stencil Computations: Multicore-optimized wavefront diamond blocking on Shared and Distributed Memory Systems," Technical Poster Session in Supercomputing 2015.

- Tareq Malas, Aron J. Ahmadia, Jed Brown, John A. Gunnels, and David E. Keyes. "Optimizing the performance of streaming numerical kernels on the IBM Blue Gene/P PowerPC 450 processor." *International Journal of High Performance Computing Applications*, 27(2):193-209, 2013.

B Additional hardware counters measurements of the results

In this appendix, we show measurements details of the results presented in Sect. 5.2. The measurements include various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking results.

We show eleven figures for each stencil and processor combination, where the y-labels and the legends in the figures indicate the measurements:

- Measured performance (GLUP/s).
- Hardware counters values of L1 cache data TLB loads miss rate average.
- Hardware counters values of the CPU load to store ratio.
- The data transfer volumes from the main memory, L3 cache, and L2 cache, normalized by the total LUPs updates (in the second row of sub-figures).
- The selected tile size parameters by the auto-tuner, consisting of the diamond tile and wavefront tile sizes in 1WD, MWD, and PLUTO. In addition, we show the tile size along the x-axis in PLUTO (in the last sub-figure).
- The total Cache block size estimate by our model for 1WD and MWD.
- The selected intra-tile parallelization of MWD by the auto-tuner (the figure with "Intra-tile threads" y-label).

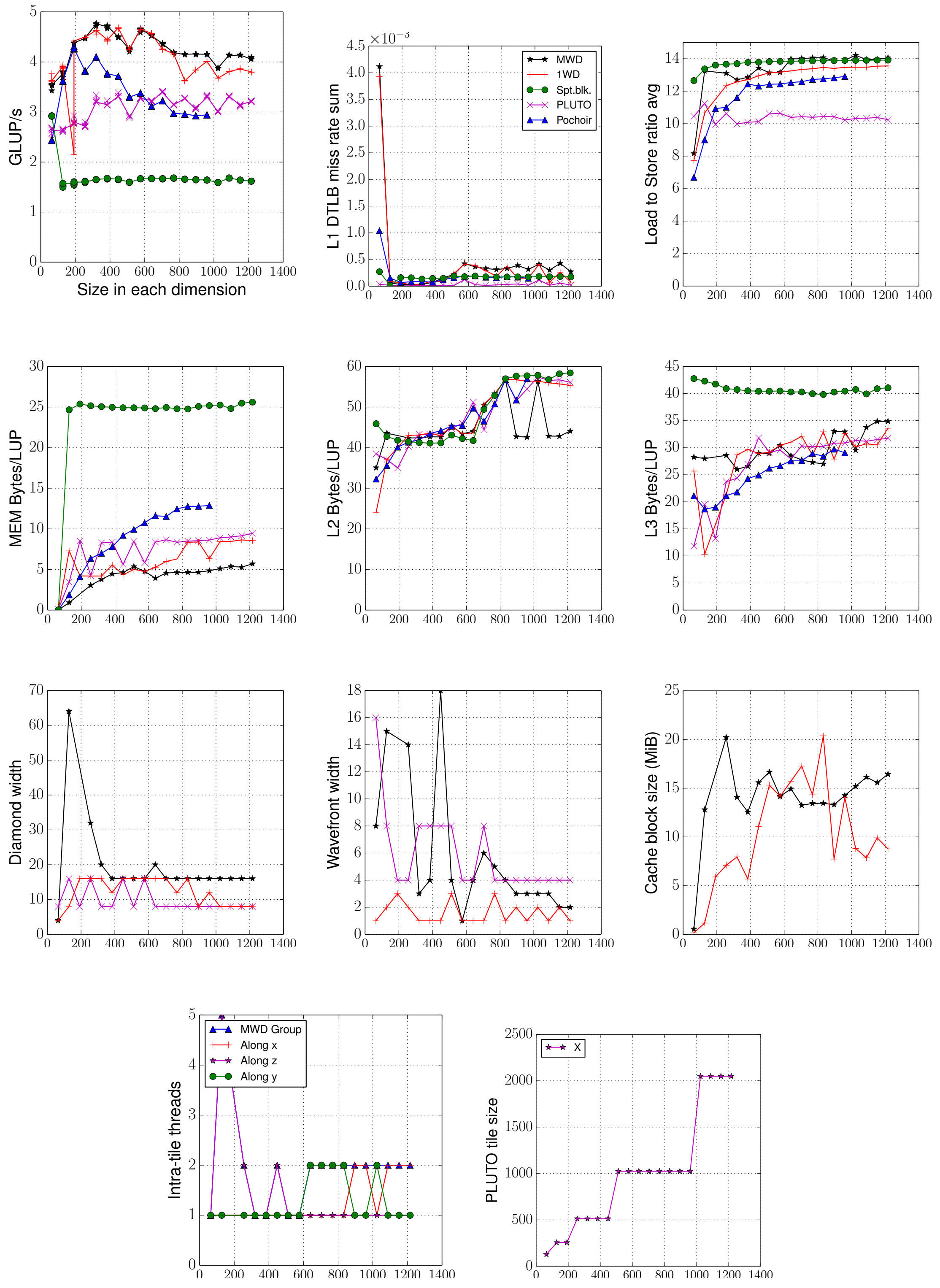


Figure B.1: Ivy Bridge 7-point constant-coefficient stencil results, using increasing cubic grid size. Showing various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

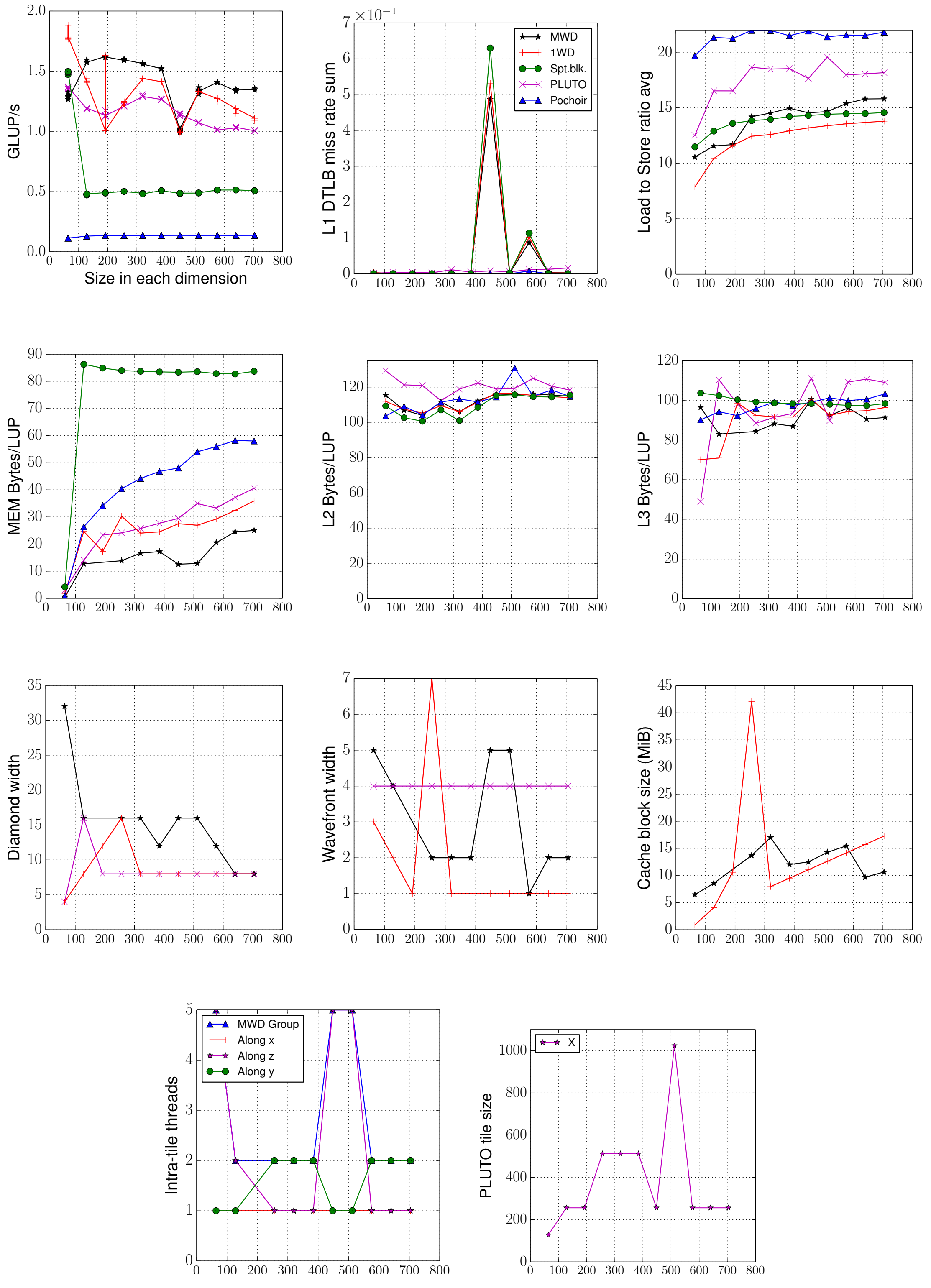


Figure B.2: Ivy Bridge 7-point variable-coefficient stencil results, using increasing cubic grid size. Showing various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

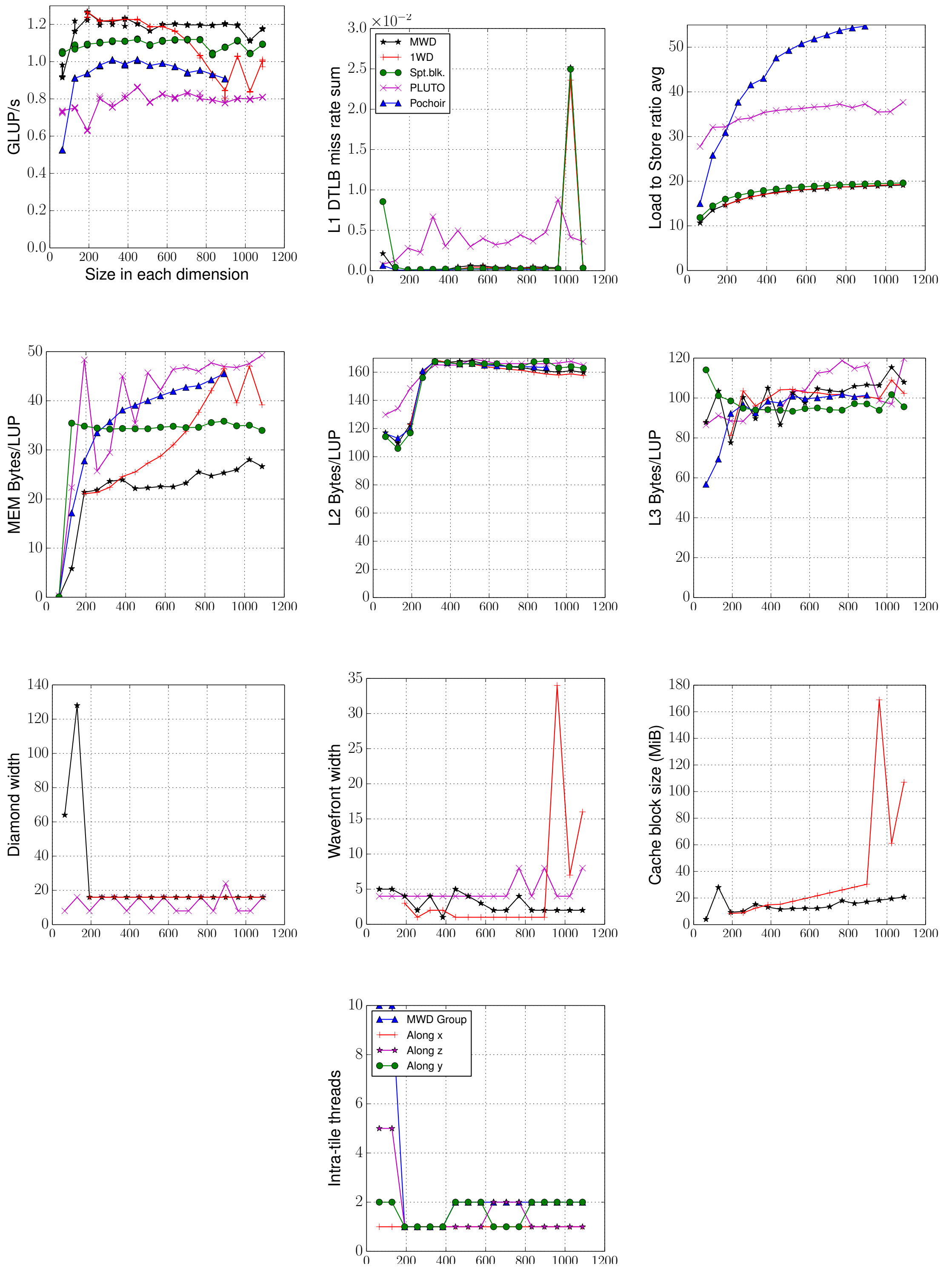


Figure B.3: Ivy Bridge 25-point constant-coefficient stencil results, using increasing cubic grid size. Showing various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

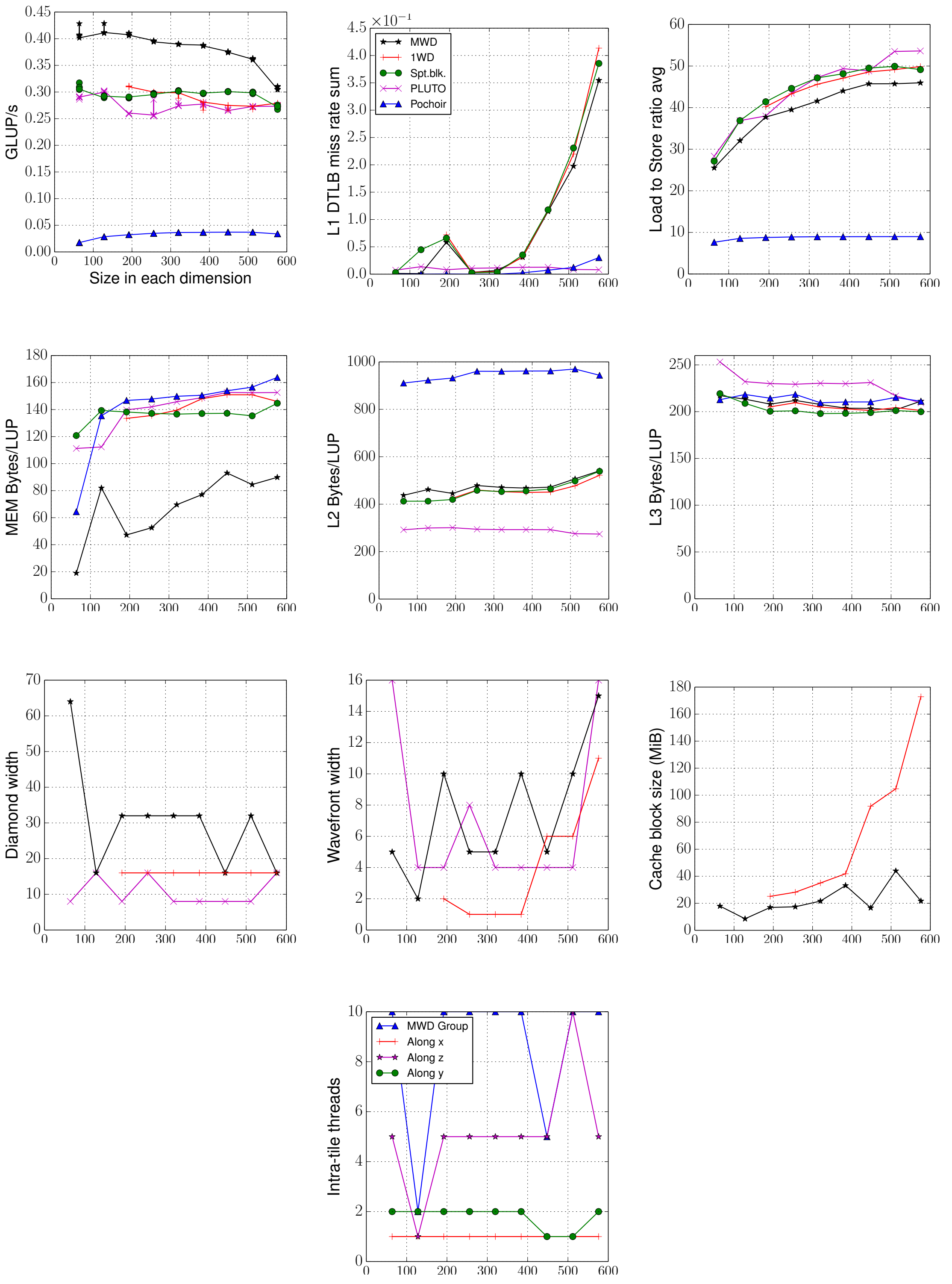


Figure B.4: Ivy Bridge 25-point variable-coefficient stencil results, using increasing cubic grid size. Showing various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

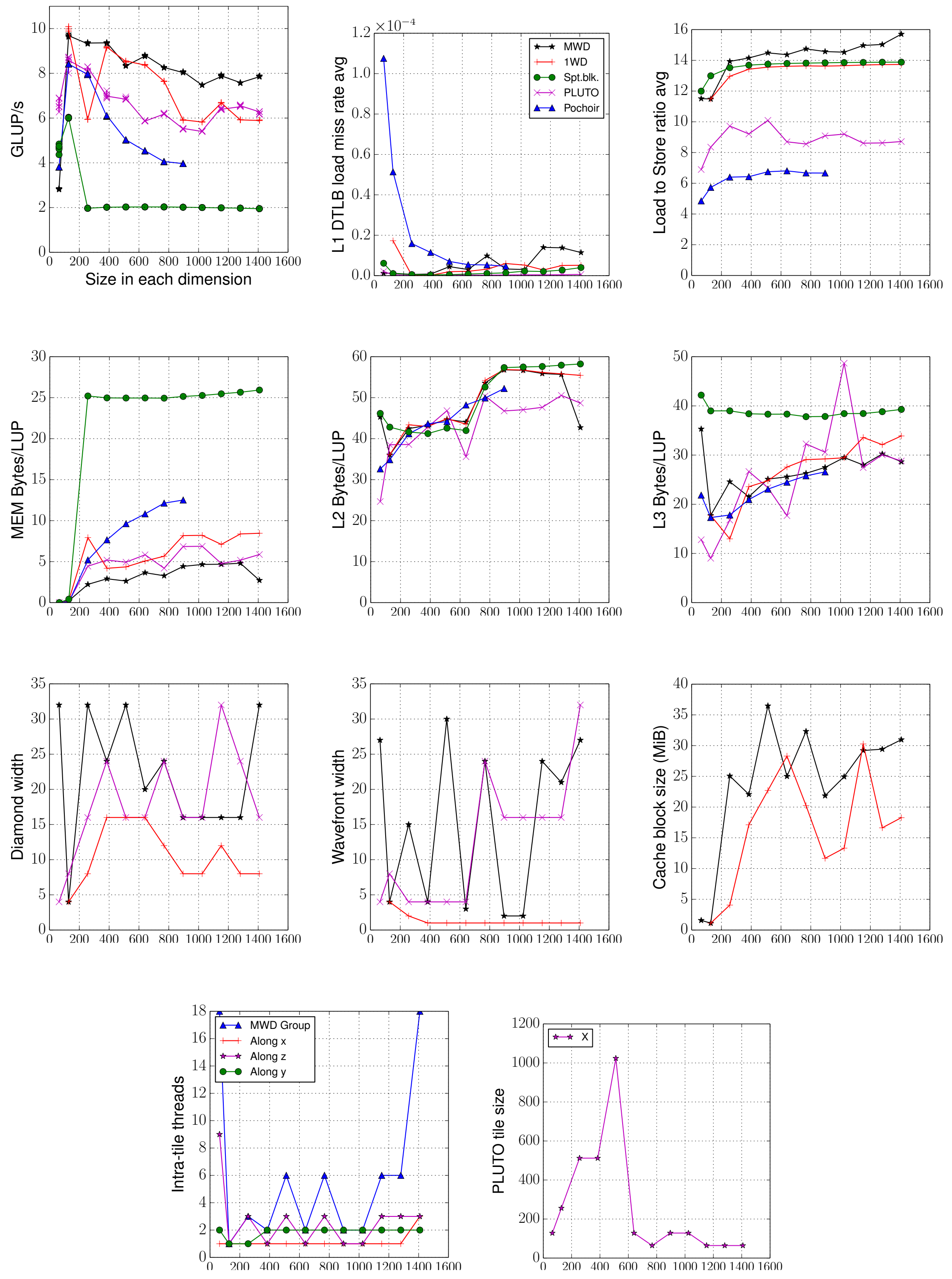


Figure B.5: Haswell 7-point constant-coefficient stencil results, using increasing cubic grid size. Showing various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

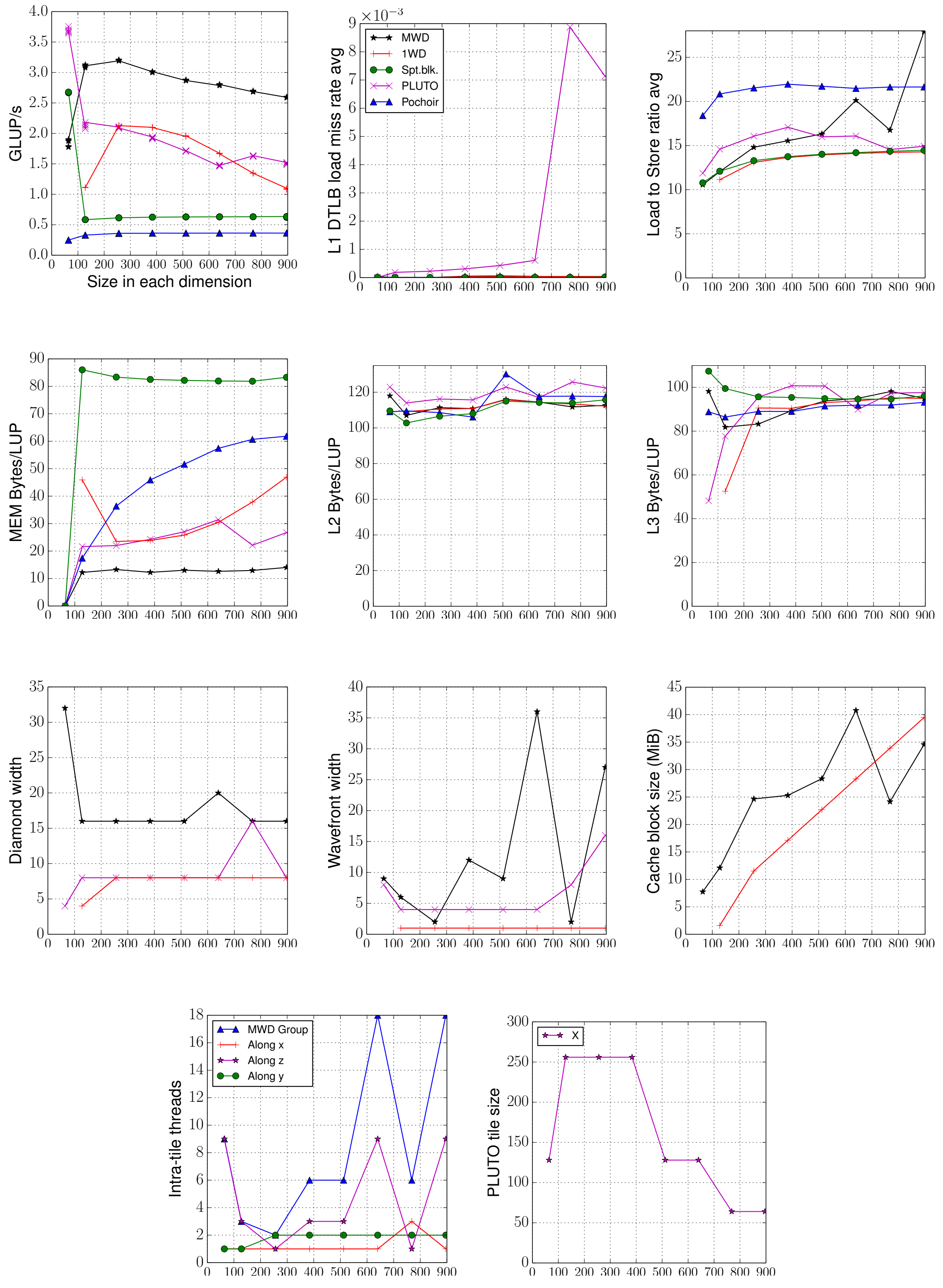


Figure B.6: Haswell 7-point variable-coefficient stencil results, using increasing cubic grid size. Showing various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

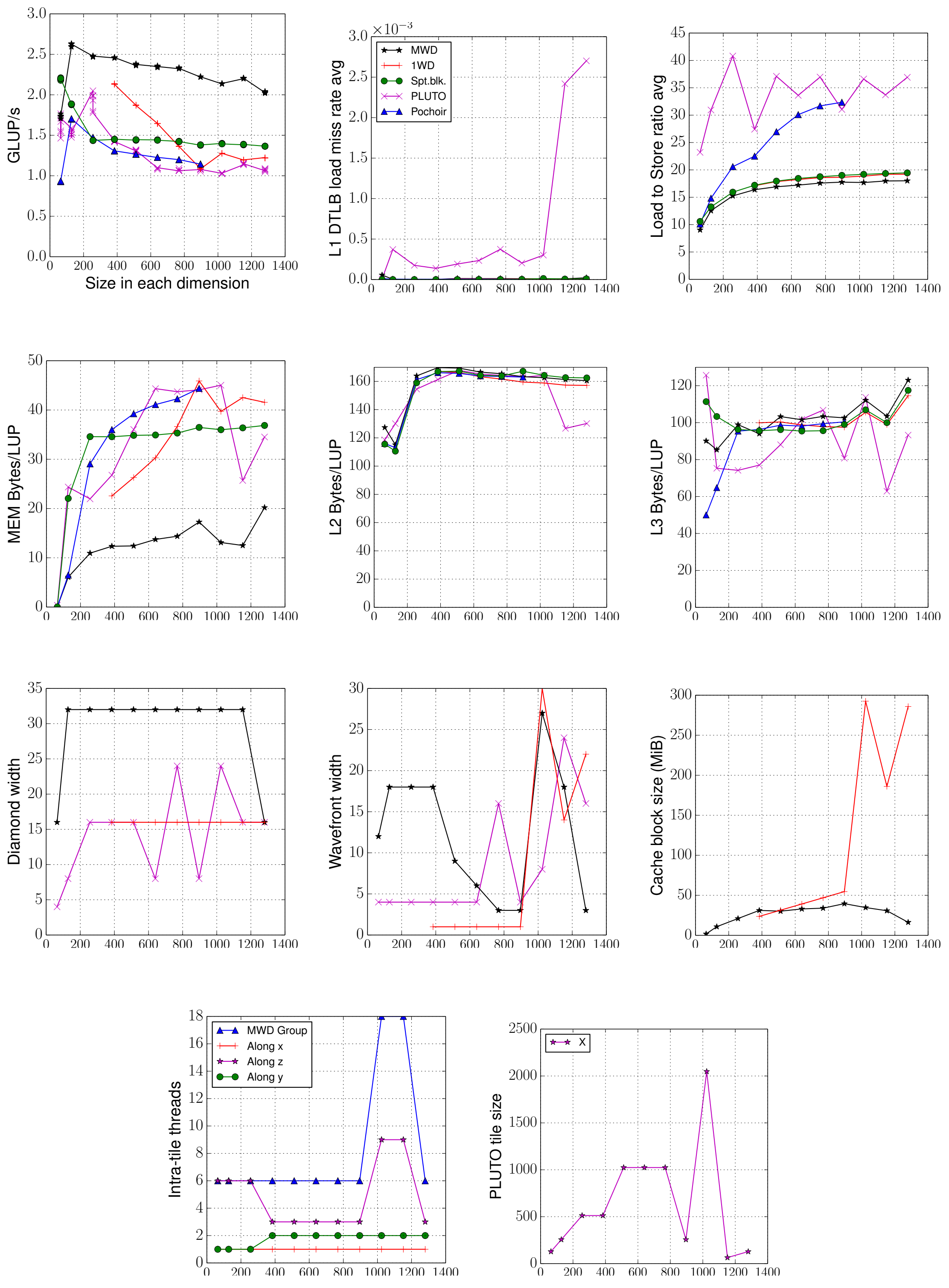


Figure B.7: Haswell 25-point constant-coefficient stencil, using increasing cubic grid size. Showing various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

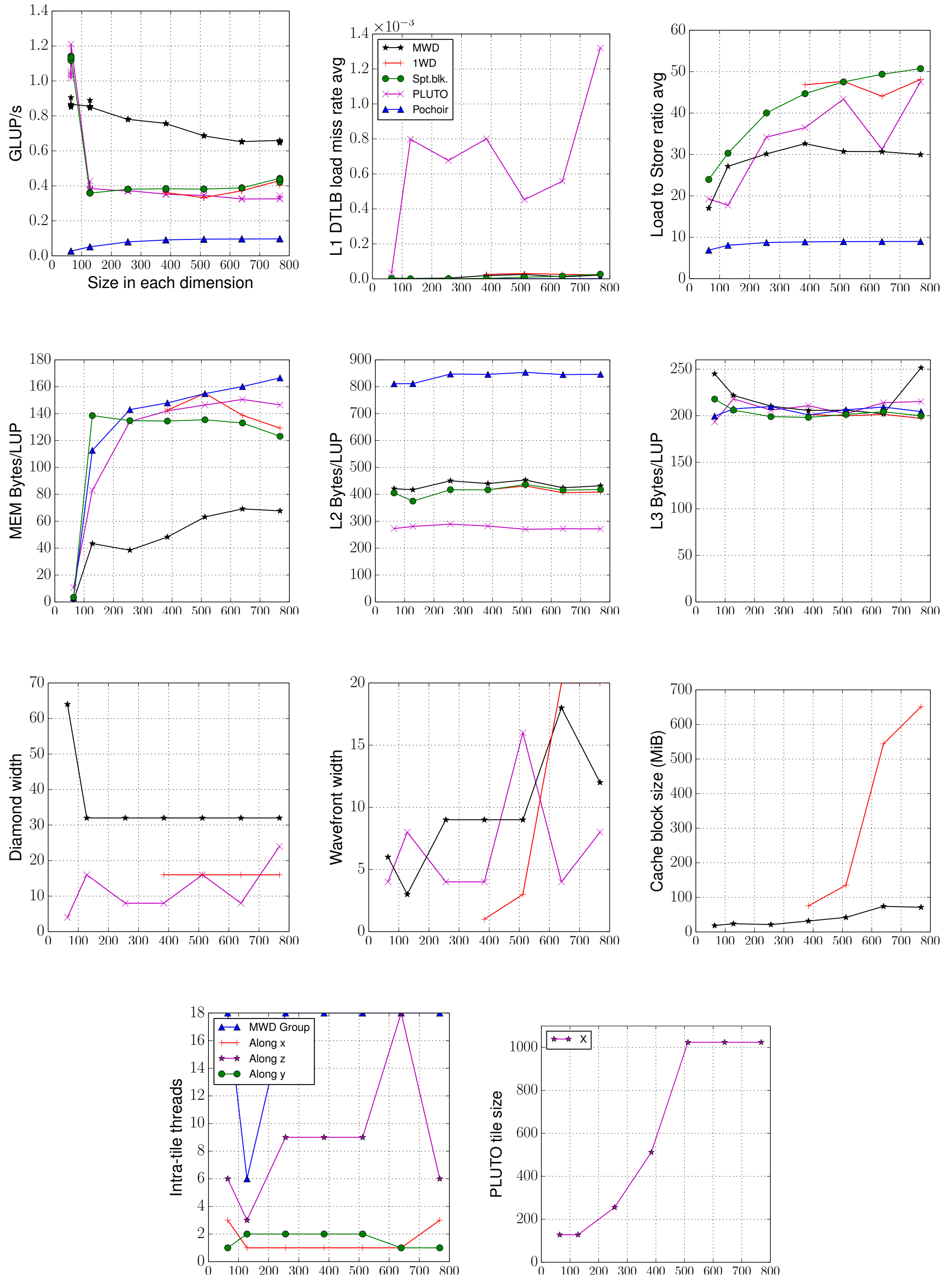


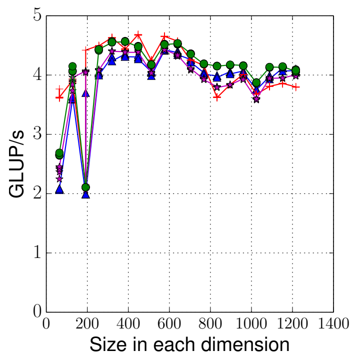
Figure B.8: Haswell 25-point variable-coefficient stencil results, using increasing cubic grid size. Showing various hardware counters measurements and tiling parameters of MWD, PLUTO, Pochoir, 1WD, and spatial blocking.

C Additional results for MWD tile sharing impact on performance, memory transfer, and energy consumption

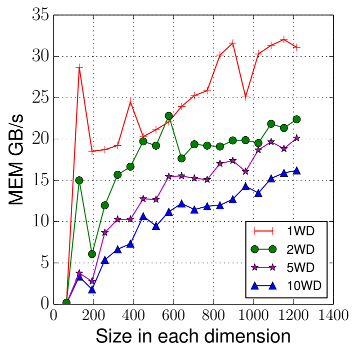
In this appendix, we show complementary results of those presented in Sect. 5.3, where we show the remaining results of the four corner-case stencils in the Intel Ivy Bridge and Haswell processors.

The same conclusions of the detailed analysis in Sect. 5.3 apply to the results in this appendix. For example, larger thread group size leads to less memory bandwidth usage and less memory traffic, as shown in all figures. We also observe negligible synchronization overhead in large thread groups when small thread group size decouple from main memory, as shown in Figs. C.1 and C.2.

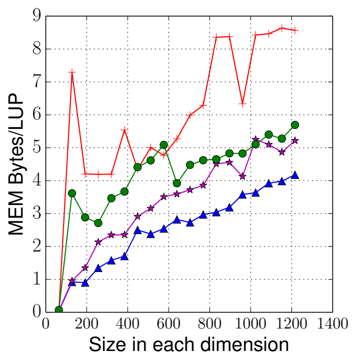
We observe performance degradation in some grid sizes in Figs. C.1a and C.2a. Our measurements show a significant increase in the TLB at these points.



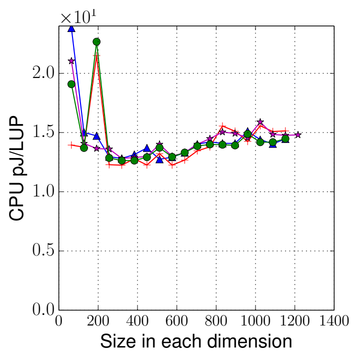
(a) Performance.



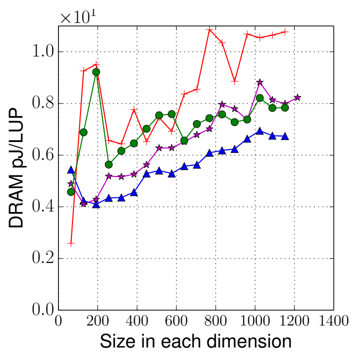
(b) Measured memory bandwidth usage.



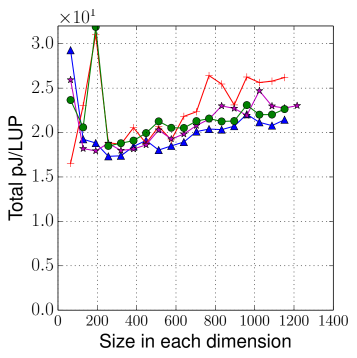
(c) Measured memory transfers per LUP.



(d) CPU energy consumption estimates.

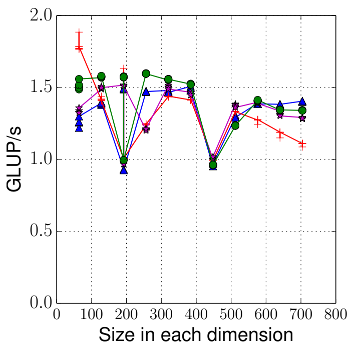


(e) DRAM energy consumption estimates.

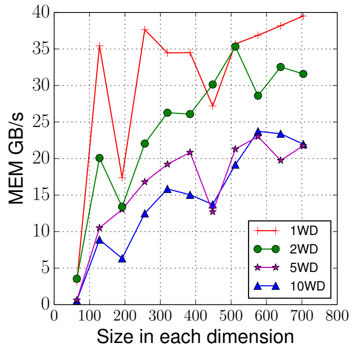


(f) Total energy consumption estimates.

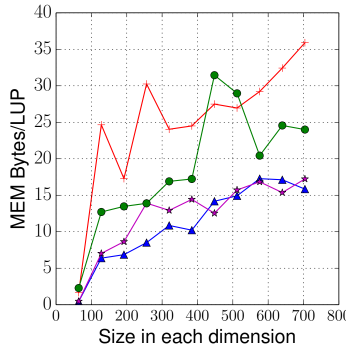
Figure C.1: Ivy Bridge 7-point constant-coefficient performance, memory transfer measurements, and energy consumption estimates using increasing cubic grid size. We compare various thread group sizes in MWD.



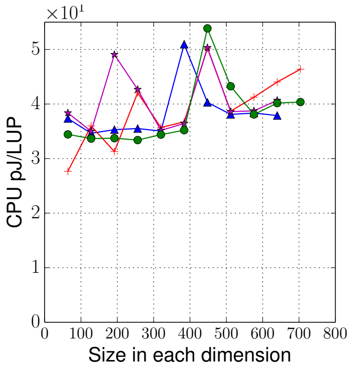
(a) Performance.



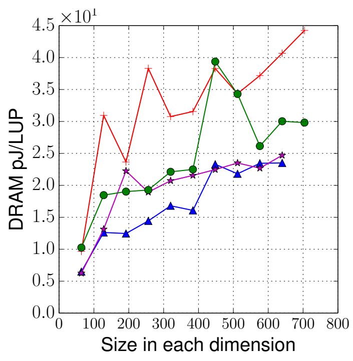
(b) Measured memory bandwidth usage.



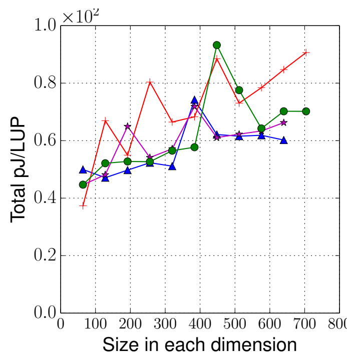
(c) Measured memory transfers per LUP.



(d) CPU energy consumption estimates.

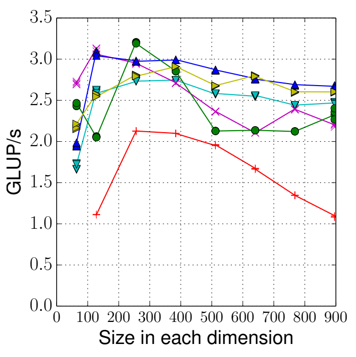


(e) DRAM energy consumption estimates.

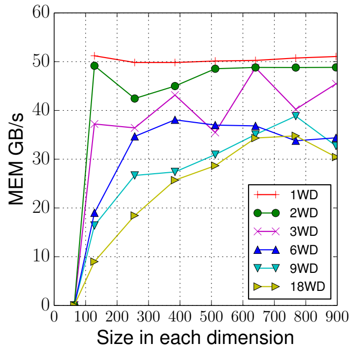


(f) Total energy consumption estimates.

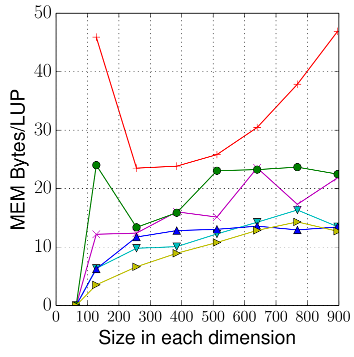
Figure C.2: Ivy Bridge 7-point variable-coefficient stencil performance, memory transfer measurements, and energy consumption estimates using increasing cubic grid size. We compare various thread group sizes in MWD.



(a) Performance.

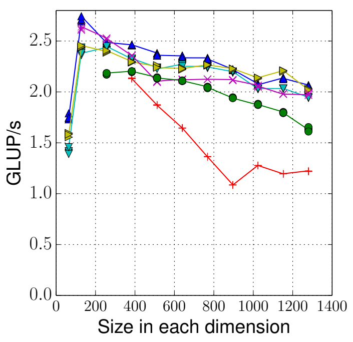


(b) Measured memory bandwidth usage.

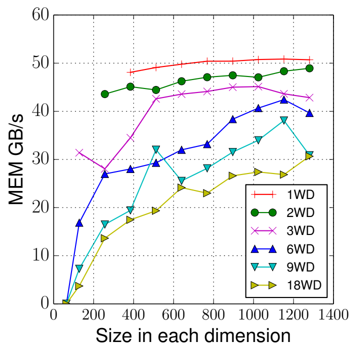


(c) Measured memory transfers per LUP.

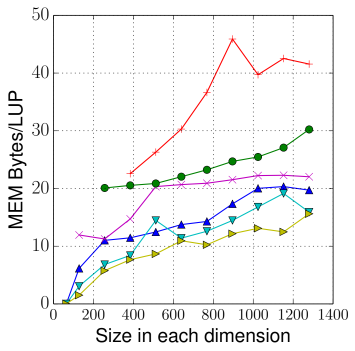
Figure C.3: Haswell 7-point variable-coefficient stencil performance and memory transfer measurements using increasing cubic grid size. We compare various thread group sizes in MWD.



(a) Performance.

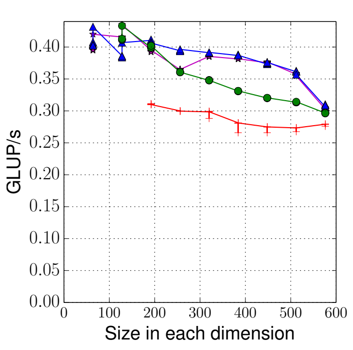


(b) Measured memory bandwidth usage.

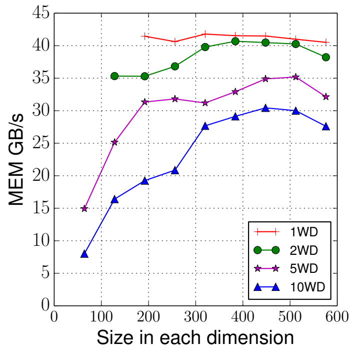


(c) Measured memory transfers per LUP.

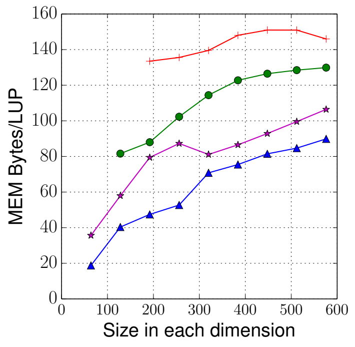
Figure C.4: Haswell 25-point constant-coefficient stencil performance and memory transfer measurements using increasing cubic grid size. We compare various thread group sizes in MWD.



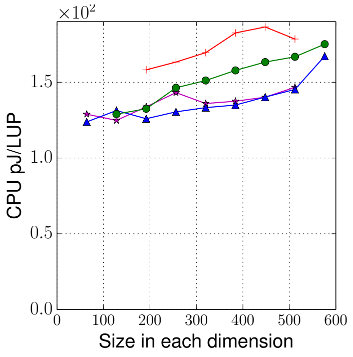
(a) Performance.



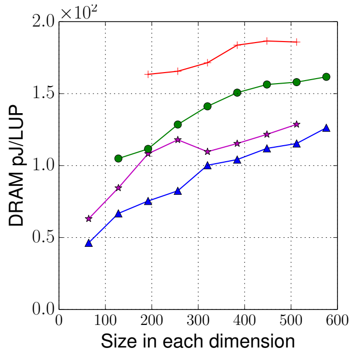
(b) Measured memory bandwidth usage.



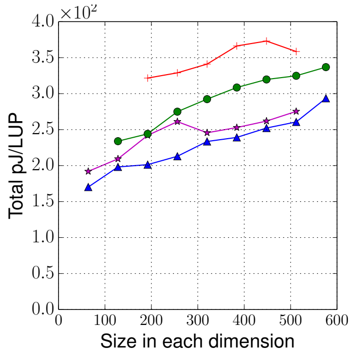
(c) Measured memory transfers per LUP.



(d) CPU energy consumption estimates.



(e) DRAM energy consumption estimates.



(f) Total energy consumption estimates.

Figure C.5: Ivy Bridge 25-point variable-coefficient stencil performance, memory transfer measurements, and energy consumption estimates using increasing cubic grid size. We compare various thread group sizes in MWD.

D Maxwell equations kernels

We present the stencil kernel we use for the solar simulation solver in the application chapter. Each grid point update involves updating six components in the H-field (shown in D.1 and D.2) and six components in the E-field (shown in D.3 and D.4). The updates can be performed concurrently in each field. The data dependency across the fields is shown in Fig. 6.3.

Listing D.1: Magnetic field kernels (1 of 2), showing H_{YX}, H_{ZX}, and H_{XY} updates

```
for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(-nnx*nny);
      stagDiffR = Exyd[i]-Exyd[isub]+Exzd[i]-Exzd[isub];
      stagDiffI = Exyd[i+1]-Exyd[isub+1]+Exzd[i+1]-Exzd[isub+1];
      asgn      = Hyxd[i]*tHyxd[i]-Hyxd[i+1]*tHyxd[i+1]+Hybndd[i]
                 -cHyxd[i]*stagDiffR+cHyxd[i+1]*stagDiffI;
      Hyxd[i+1] = Hyxd[i]*tHyxd[i+1]+Hyxd[i+1]*tHyxd[i]+Hybndd[i+1]
                 -cHyxd[i]*stagDiffI-cHyxd[i+1]*stagDiffR;
      Hyxd[i]   = asgn; }}}

for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(-nnx);
      stagDiffR = Exyd[isub]-Exyd[i]+Exzd[isub]-Exzd[i];
      stagDiffI = Exyd[isub+1]-Exyd[i+1]+Exzd[isub+1]-Exzd[i+1];
      asgn      = Hzxd[i]*tHzxd[i]-Hzxd[i+1]*tHzxd[i+1]
                 -cHzxd[i]*stagDiffR+cHzxd[i+1]*stagDiffI;
      Hzxd[i+1] = Hzxd[i]*tHzxd[i+1]+Hzxd[i+1]*tHzxd[i]
                 -cHzxd[i]*stagDiffI-cHzxd[i+1]*stagDiffR;
      Hzxd[i]   = asgn; }}}

for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(-nnx*nny);
      stagDiffR = Eyxd[isub]-Eyxd[i]+Eyzd[isub]-Eyzd[i];
      stagDiffI = Eyxd[isub+1]-Eyxd[i+1]+Eyzd[isub+1]-Eyzd[i+1];
      asgn      = Hxyd[i]*tHxyd[i]-Hxyd[i+1]*tHxyd[i+1]+Hxbndd[i]
                 -cHxyd[i]*stagDiffR+cHxyd[i+1]*stagDiffI;
      Hxyd[i+1] = Hxyd[i]*tHxyd[i+1]+Hxyd[i+1]*tHxyd[i]+Hxbndd[i+1]
                 -cHxyd[i]*stagDiffI-cHxyd[i+1]*stagDiffR;
      Hxyd[i]   = asgn; }}}
```

Listing D.2: Magnetic field kernels (2 of 2), showing H_{ZY}, H_{XZ}, and H_{YZ} updates

```
for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(-1);
      stagDiffR = Eyxd[i]-Eyxd[isub]+Eyzd[i]-Eyzd[isub];
      stagDiffI = Eyxd[i+1]-Eyxd[isub+1]+Eyzd[i+1]-Eyzd[isub+1];
      asgn      = Hzyd[i]*tHzyd[i]-Hzyd[i+1]*tHzyd[i+1]
                 -cHzyd[i]*stagDiffR+cHzyd[i+1]*stagDiffI;
      Hzyd[i+1] = Hzyd[i]*tHzyd[i+1]+Hzyd[i+1]*tHzyd[i]
                 -cHzyd[i]*stagDiffI-cHzyd[i+1]*stagDiffR;
      Hzyd[i]   = asgn; }}}

for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(-nnx);
      stagDiffR = Ezxd[i]-Ezxd[isub]+Ezyd[i]-Ezyd[isub];
      stagDiffI = Ezxd[i+1]-Ezxd[isub+1]+Ezyd[i+1]-Ezyd[isub+1];
      asgn      = Hxzd[i]*tHxzd[i]-Hxzd[i+1]*tHxzd[i+1]
                 -cHxzd[i]*stagDiffR+cHxzd[i+1]*stagDiffI;
      Hxzd[i+1] = Hxzd[i]*tHxzd[i+1]+Hxzd[i+1]*tHxzd[i]
                 -cHxzd[i]*stagDiffI-cHxzd[i+1]*stagDiffR;
      Hxzd[i]   = asgn; }}}

for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(-1);
      stagDiffR = Ezxd[isub]+Ezyd[isub]-Ezxd[i]-Ezyd[i];
      stagDiffI = Ezxd[isub+1]+Ezyd[isub+1]-Ezxd[i+1]-Ezyd[i+1];
      asgn      = Hyzd[i]*tHyzd[i]-Hyzd[i+1]*tHyzd[i+1]
                 -cHyzd[i]*stagDiffR+cHyzd[i+1]*stagDiffI;
      Hyzd[i+1] = Hyzd[i]*tHyzd[i+1]+Hyzd[i+1]*tHyzd[i]
                 -cHyzd[i]*stagDiffI-cHyzd[i+1]*stagDiffR;
      Hyzd[i]   = asgn; }}}
```

Listing D.3: Electric field kernels (1 of 2), showing E_{XZ}, E_{YZ}, and E_{YX} updates

```
for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(+nnx);
      stagDiffR = Hzxd[isub]-Hzxd[i]+Hzyd[isub]-Hzyd[i];
      stagDiffI = Hzxd[isub+1]-Hzxd[i+1]+Hzyd[isub+1]-Hzyd[i+1];
      asgn      = Exzd[i]*tExzd[i]-Exzd[i+1]*tExzd[i+1]
                 +cExzd[i]*stagDiffR-cExzd[i+1]*stagDiffI;
      Exzd[i+1] = Exzd[i]*tExzd[i+1]+Exzd[i+1]*tExzd[i]
                 +cExzd[i]*stagDiffI+cExzd[i+1]*stagDiffR;
      Exzd[i]   = asgn; }}}

for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(+1);
      stagDiffR = Hzxd[i]+Hzyd[i]-Hzxd[isub]-Hzyd[isub];
      stagDiffI = Hzxd[i+1]+Hzyd[i+1]-Hzxd[isub+1]-Hzyd[isub+1];
      asgn      = Eyzd[i]*tEyzd[i]-Eyzd[i+1]*tEyzd[i+1]
                 +cEyzd[i]*stagDiffR-cEyzd[i+1]*stagDiffI;
      Eyzd[i+1] = Eyzd[i]*tEyzd[i+1]+Eyzd[i+1]*tEyzd[i]
                 +cEyzd[i]*stagDiffI+cEyzd[i+1]*stagDiffR;
      Eyzd[i]   = asgn; }}}

for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(+nnx*nny);
      stagDiffR = Hxyd[isub]-Hxyd[i]+Hxzd[isub]-Hxzd[i];
      stagDiffI = Hxyd[isub+1]-Hxyd[i+1]+Hxzd[isub+1]-Hxzd[i+1];
      asgn      = Eyxd[i]*tEyxd[i]-Eyxd[i+1]*tEyxd[i+1]+Eybndd[i]
                 +cEyxd[i]*stagDiffR-cEyxd[i+1]*stagDiffI;
      Eyxd[i+1] = Eyxd[i]*tEyxd[i+1]+Eyxd[i+1]*tEyxd[i]+Eybndd[i+1]
                 +cEyxd[i]*stagDiffI+cEyxd[i+1]*stagDiffR;
      Eyxd[i]   = asgn; }}}
```

Listing D.4: Electric field kernels (2 of 2), showing E_{ZX}, E_{XY}, and E_{ZY} updates

```
for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(+nnx);
      stagDiffR = Hxyd[i]+Hxzd[i]-Hxyd[isub]-Hxzd[isub];
      stagDiffI = Hxyd[i+1]+Hxzd[i+1]-Hxyd[isub+1]-Hxzd[isub+1];
      asgn      = Ezxd[i]*tEzxd[i]-Ezxd[i+1]*tEzxd[i+1]
                 +cEzxd[i]*stagDiffR-cEzxd[i+1]*stagDiffI;
      Ezxd[i+1] = Ezxd[i]*tEzxd[i+1]+Ezxd[i+1]*tEzxd[i]
                 +cEzxd[i]*stagDiffI+cEzxd[i+1]*stagDiffR;
      Ezxd[i]   = asgn; }}}

for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(+nnx*nny);
      stagDiffR = Hyxd[i]-Hyxd[isub]+Hyzd[i]-Hyzd[isub];
      stagDiffI = Hyxd[i+1]-Hyxd[isub+1]+Hyzd[i+1]-Hyzd[isub+1];
      asgn      = Exyd[i]*tExyd[i]-Exyd[i+1]*tExyd[i+1]+Exbndd[i]
                 +cExyd[i]*stagDiffR-cExyd[i+1]*stagDiffI;
      Exyd[i+1] = Exyd[i]*tExyd[i+1]+Exyd[i+1]*tExyd[i]+Exbndd[i+1]
                 +cExyd[i]*stagDiffI+cExyd[i+1]*stagDiffR;
      Exyd[i]   = asgn; }}}

for(k=zb; k<ze; k++) {
  for(j=yb; j<ye; j++) {
    for(i=2*((k*nny+j)*nnx+xb); i<2*((k*nny+j)*nnx+xe); i+=2) {
      isub      = i+2*(+1);
      stagDiffR = Hyxd[isub]-Hyxd[i]+Hyzd[isub]-Hyzd[i];
      stagDiffI = Hyxd[isub+1]-Hyxd[i+1]+Hyzd[isub+1]-Hyzd[i+1];
      asgn      = Ezyd[i]*tEzyd[i]-Ezyd[i+1]*tEzyd[i+1]
                 +cEzyd[i]*stagDiffR-cEzyd[i+1]*stagDiffI;
      Ezyd[i+1] = Ezyd[i]*tEzyd[i+1]+Ezyd[i+1]*tEzyd[i]
                 +cEzyd[i]*stagDiffI+cEzyd[i+1]*stagDiffR;
      Ezyd[i]   = asgn; }}}
```

E Case study: code analysis of acoustics solvers for wave equations

We perform Roofline model analysis of important PDE solvers to evaluate the expected performance gain of our temporal blocking techniques.

We analyze part of the Clawpack code [75], which uses finite volume Godunov-type methods. In particular, we inspect a code in SharpClaw algorithm [76], which uses strong stability preserving Runge–Kutta time integration, slope limiters, and Riemann solvers. We analyze a single Runge–Kutta stage, based on using the minmod limiter and solving the variable-coefficient acoustic wave equation on a curvilinear mapped grid. This is a three-dimensional problem with multiple solution components and multiple material coefficients per grid cell[1].

The computation requires one TVD reconstruction[2] and two Riemann solves[3] in each cell. These computational kernels are called from a subroutine that is applied to one one-dimension slice of the grid at a time[4].

The Clawpack code is written with a priority on flexibility rather than efficiency. The original code performs the updates dimension by dimension, where the strides are copied to separate buffer and a one-dimensional solver update is performed. In order

[1]The solver is available at: `https://github.com/clawpack/pyclaw/blob/master/examples/acoustics_3d_variable/acoustics_3d_interface.py`

[2]The subroutine `tvd2` in `https://github.com/clawpack/pyclaw/blob/master/src/pyclaw/sharpclaw/reconstruct.f90`

[3]The subroutine in `https://github.com/ketch/riemann/blob/acoustics_mapped_3d/src/rpn3_acoustics_mapped.f90`

[4]`https://github.com/clawpack/pyclaw/blob/master/src/pyclaw/sharpclaw/flux1.f90`

to improve base efficiency, we inlined all the subroutines into a single function and removed the unnecessary array copy and redundant computations. This optimization made the code less flexible and less readable, but this is an important step to get closer to the best attainable performance.

We calculate a total of 483 flops per grid cell update. The Second order TVD reconstruction costs 222 flops and the Riemann solves cost 261 flops. The minimum traffic to main memory is 27 numbers/cell: 15 auxiliary, 4 solution domain reads (4 equations), 8 solution domain read/write (assuming no non-temporal stores). Thus, the arithmetic intensity is $483/27 = 18$ Flops/number. In double precision, we get 2.25 flops/byte. This calculation assumes perfect spatial blocking, where each number is brought once to the cache memory and reused completely before evicting it to the main memory.

The arithmetic intensity of this code is much higher compared to that of the 7-point constant-coefficient stencil (0.54 flop/byte). Moreover, the complexity of this code adds bottlenecks at the core, which results in less memory bandwidth usage pressure. Because this code is not memory bound on typical current architectures, we believe that temporal blocking techniques would not improve its performance.