Optimization of Stencil Computations on GPUs

DISSERTATION

Presented in Partial Fulfillment of the Requirements
for the Degree Doctor of Philosophy
in the Graduate School of The Ohio State University

By

Prashant Singh Rawat, M.Tech.

Graduate Program in Computer Science and Engineering

The Ohio State University

2018

Dissertation Committee:

Prof. P. Sadayappan, Advisor

Prof. Atanas Rountev

Prof. Gagan Agrawal

ABSTRACT

Stencil computations form the compute-intensive core of many scientific application domains, such as image processing of CT and MRI imaging, computational electromagnetics, seismic processing, and climate modeling. A stencil computation involves element-wise update of an output domain based on a fixed set of neighboring points from the input domain. Such stencil computations are either time iterated, or require successive application of multiple stencil operators on the input domains.

Stencil optimization on multi- and many-core architectures has been an active research topic for the past two decades. Stencil computations traditionally have low arithmetic intensity with only a few floating-point operations performed relative to the data transferred per output point, and are therefore memory bandwidth-bound. Since the data movement cost consistently dominates the computational cost in modern architectures, most of these research efforts focus on reducing the data movement in stencils to tackle the bandwidth bottleneck. Consequently, several tiling techniques have been proposed over the years to exploit spatial and temporal reuse across a sequence of stencils or across multiple time steps for time iterated stencil.

With the ever-increasing use of GPUs for general purpose computing, application developers have started exploring the acceleration of data-parallel stencils on GPUs. GPUs have lower data movement costs than the multi-core CPU architectures, and hence are an attractive target for accelerating memory bandwidth-bound stencil computations. At the same time, GPUs are compute-intensive with significantly higher number of registers per thread, and therefore suitable for accelerating stencil computations with high arithmetic intensity as well. The arithmetic intensity of a stencil is proportional to its *order*, which is the number of input elements read from the center along each dimension. In many scientific applications, high-order stencils provide better computational accuracy with lesser data movement than their low-order counterparts. However, the main performance bottleneck for high-order stencils on GPUs is the high register pressure, which causes excessive register spills or a steep drop in achieved parallelism, resulting in a subsequent performance loss.

This dissertation proposes novel GPU-centric optimization strategies that address the performance bottlenecks for stencils with different arithmetic intensities: tiling and fusion heuristics for bandwidth-bound stencils with low arithmetic intensity, and register optimizations for high-order stencils with high arithmetic intensity. The proposed optimizations have been implemented into a DSL based stencil optimization framework, StencilGen, that can automatically generate high-performance CUDA code from an input DSL specification of the stencil computation. The efficacy of the proposed optimizations is demonstrated via empirical evaluation on a variety of 2D and 3D stencil kernels extracted from PDE solvers, image processing pipelines, and proxy DOE applications.

To my mother and my sisters, for their unbounded love and unconditional support.

ACKNOWLEDGMENTS

Now that I look back, this part of life was, in measures, exciting, overwhelming and exhausting. I experienced everything, from first-year jitters, second-year blues, and home sickness, to working tirelessly in burst mode, frustration over rejected papers, and the thrill of publishing; I will cherish these experiences for life. Thankfully, I was not alone in this journey. There were many people who treaded along, making things easier, and they deserve a shout-out.

I could have not found a better advisor than Prof. P. Sadayappan. I have improved on many aspects by listening to his discussions, critiques, and invaluable advice over the past five years. He steered my research when I needed it, and gave me free reins at other times to hone the independent researcher in me. He provided me with internship opportunities, supported me through tough times, and I am certain, helped in many more behind-the-scene situations that I would not even know about. He is an exceptional mentor to all his students.

I am grateful to Prof. Atanas Rountev for his constant help during the past five years, and especially during the first two years of the grad school. Most of my papers made it to publication because of his insightful advice, careful edits, and excellent suggestions on how to translate ideas into words.

I am thankful to Louis-Noël Pouchet and Fabrice Rastello for their contributions to various parts of the dissertation. Louis-Noël has especially been instrumental in shaping up the experimental sections in all my papers. Fabrice provided several insights on the reordering framework and the related work on register allocation in general.

The years I spent at IIT Bombay gave me a new direction. I had the most patient M.Tech advisor, Prof. Supratim Biswas. I am eternally grateful to Prof. Uday Khedker for introducing me to compilers during my Bachelors, letting me be a part of GCC Resource Center (GRC) during my Masters, introducing me to Prof. Biswas and Saday, and bridging my path from IITB to OSU. I spent five years working with him, and I could spend another five talking about how amazing the days with him in GRC were.

I want to express my gratitude to all the present and past fellow lab mates at OSU: Venmugil, Israt, Aravind, Mahesh, Sanket, Shashank, Ankur, Martin, Tom, Kevin, Naznin, Naser, Samyam, Changwan, Jinsung, Wenlei, Miheer, Vineeth, Rohit, Kunal, Emre, Gordon. Also, my friends at IITB: Nisha, Akshar, Salil, Subhajit, Harbaksh, Prajakta, Vini, Jubi, Ishita, Ashish, Prachi, Pritam, Swati, Swaroop, Shubhangi. Thanks for the interesting conversations! I am grateful to my close friends Bharat, Prashal, Santosh, and Ravi, who knew me when I was a confused teen, and helped me transition into what I am today.

And finally, my family back in India – my mother, sisters, brother-in-laws, the whole entourage of nieces and nephews – and my family here, Umesh. Their unconditional love and support is what keeps me going on. Without their encouragement, I could have not reached this far. People rightly say that distance makes the heart grow fonder. I wish I had the vocabulary to do justice to my love for them. This work is as much theirs as it is mine.

VITA

June 2003 – May 2007	Bachelor of Engineering, CSE Mumbai University Mumbai, India.
June 2007 – May 2009	Research Assistant IIT Bombay Mumbai, India.
June 2009 – July 2012	Master of Technology, CSE IIT Bombay Mumbai, India.
August 2012 – Present	Graduate Research Assistant The Ohio State University Columbus, Ohio.
May 2015 – August 2015	Intern NVidia Corporation Redmond, Washington.
May 2016 – June 2016	Intern Lawrence Livermore National Lab. Livermore, California.
May 2017 – August 2017	Intern NVidia Corporation Redmond, Washington.

PUBLICATIONS

Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, P. Sadayappan
GPU Code Optimization via Abstract Kernel Emulation and Sensitivity Analysis.
To appear in *ACM SIGPLAN conference on Pogramming Language Design and Implementation (PLDI)*, June 2018.

Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, Fabrice Rastello, P. Sadayappan
Register Optimizations for Stencils on GPUs.
In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2018.

Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, P. Sadayappan
POSTER: Performance Modeling for GPUs using Abstract Kernel Emulation.
In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2018.

Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, Fabrice Rastello, P. Sadayappan
POSTER: Statement Reordering to Alleviate Register Pressure for Stencils on GPUs.
In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2017.

Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, Atanas Rountev, P. Sadayappan
Resource Conscious Reuse-Driven Tiling for GPUs.
In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2016.

Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, Atanas Rountev, P. Sadayappan
Effective resource management for enhancing performance of 2D and 3D stencils on GPUs.
In *9th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU)*, March 2016.

Prashant Singh Rawat, Martin Kong, Thomas Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, P. Sadayappan
SDSLc: a multi-target domain-specific compiler for stencil computations.
In *5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, November 2015.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF LISTINGS

CHAPTER 1

Introduction

Stencil computations on structured grids are ubiquitous in high-performance computing. A stencil computation involves updating the elements of the output grid by performing computations on the elements belonging to a fixed neighborhood of the input grid. Listing 1.1 shows an example of a 3D 7-point stencil computation extracted from the HPGMG benchmark [45]. The output grid B is updated using the input grid A along with some coefficients (a, b, and *h2inv*). Lines 4-6 represent the core computation that is performed to obtain the updated value at each point of B. This is an *order-1* stencil, where the order refers to the extent of elements read along each dimension.

Stencils similar to the example in Listing 1.1 are a commonplace computational motif in applications ranging from partial differential equation (PDE) solvers [95, 45], computational electromagnetics [105], image processing [22, 23], and scientific simulation for climate modeling and seismic imaging [65, 104]. Such applications spend a significant fraction of the execution time performing the stencil computation. For example, on a Pascal Titan X device, the 3D seismic wave modeling application [104] spends more time executing a single stencil kernel than the time it takes to copy data from host to device. Given that stencils can be a compute-intensive core

Listing 1.1: 3d 7-point stencil operator extracted from HPGMG

```
for (int k=1; k<N-1; k++) {
   for (int j=1; j<N-1; j++) {
      for (int i=1; i<N-1; i++) {
         B[k][j][i] = a*A[k][j][i] + b*h2inv*(A[k-1][j][i] +
                  (A[k][j+1][i] + A[k][j][i+1] + A[k][j][i] +
                   A[k][j-1][i] + A[k][j][i-1] + A[k+1][j][i]);
      }
   }
}
```

of scientific applications, optimizing them for performance on different architectures is critical.

Accelerating Stencils on GPUs Since stencils have regular access pattern with a parallel outermost loop and the innermost loop amenable to vectorization, there have been several research efforts targeting stencil optimization on multi- and many-core CPU architectures [18, 106, 42, 7, 85, 72, 84]. Over the recent years, GPUs are being increasingly used for general-purpose computing. With higher computational power, massive degree of parallelism, higher bandwidth, and better power efficiency than the multi-core architectures, GPUs have steadily emerged as a better alternative for offloading and accelerating regular, and some irregular applications, that were traditionally executed on CPUs.

Data-parallel stencil computations are well-suited for the GPU paradigm due to a number of reasons: (a) different GPU threads can operate on different values of the output grid simultaneously to exploit the massive parallelism provided by GPUs; (b) the regular data accesses in stencils are naturally amenable to coalescence on GPUs; (c) the higher bandwidth of GPUs helps reduce the memory latency; (d) the number of registers per thread in GPUs is large, and therefore one can exploit better reuse in registers; and (e) the programmer has explicit control over data placement in scratch-pad (shared) memory, unlike the L1 cache in multi-core architectures. A number of research efforts in the last decade have focussed on offloading and optimizing stencil computations on GPUs [113, 67, 127, 64, 38, 43, 37, 36, 88, 87, 89, 9, 108, 63]

Memory Bandwidth-Bound Stencils Arithmetic intensity is defined as the number of floating-point operations performed relative to the memory accesses to perform the operations. For example, the 7-point stencil from Listing 1.1 performs 10 floating-point operations per output point at the amortized cost of 2 memory accesses (one read for A, one write for B; the three reads for coefficients a, b, and *h2inv* can come from constant memory). For a double-precision computation, its arithmetic intensity will be 0.625 FLOPs/byte. For the current generation GPU devices with around 300 GBytes/sec global memory bandwidth and a peak double-precision performance of around 1.5 TFLOPS, the required operational intensity to be compute-bound and not memory bandwidth-bound will be around 5 FLOPs/byte. The 7-point stencil will therefore be memory bandwidth-bound on such architectures, effectively under-utilizing the compute resources. In fact, stencil computations traditionally have low arithmetic intensities, and therefore achieve only a fraction of the peak computational power of the underlying GPU architecture.

Tiling of Time Iterated Bandwidth-Bound Stencils Tiling is a key transformation to enhance the temporal reuse of data, and thus reduce the amount of data transfers from/to global memory on a GPU. In a time-tiled execution, operations from several consecutive time steps of the stencil computation are combined to form a tile, which is executed atomically to exploit data reuse in scratchpad memory. With time tiling, simply slicing the input domain and assigning disjoint slices (data tiles) to different thread blocks is not feasible. For every tile, computing the boundary values at time step t requires some data computed at time step $t-1$ from neighboring tiles. This data is referred to as the *halo region* or the *ghost zone*. Different tiling schemes have been proposed to achieve concurrent execution of time-tiled blocks: overlapped tiling [43, 85, 55, 72], split tiling [37, 42], hexagonal tiling [36], etc..

Many previous efforts towards optimizing bandwidth-bound 2D stencils on GPUs have demonstrated the effectiveness of time-tiling, along with the utilization of shared memory [43, 85, 37]. However, tiling 3D stencils has proved to be a challenge for all previously reported code generators. The difficulty arises from the cubic versus quadratic increase in shared memory requirement for the 3D case, and the very limited (usually under 100 KBytes) shared memory available per streaming multiprocessor (SM) on GPUs. Consequently, the use of overlapped tiles [55] with 3D stencils results in very rapid increase in redundant computations, even for very small values of the time-tile size.

Operator Fusion in Bandwidth-Bound Stencil DAGs Multi-statement stencils arising in many scientific simulations and image processing pipelines apply a sequence of bandwidth-bound stencil operators on a set of input domains [71, 85]. Such stencil computations can be represented by a directed acyclic graph (DAG) in which the nodes represent stencil operators and the incoming edges represent inputs to these operators. Time-iterated stencils can also be expressed as a DAG by explicit unrolling of the time loop. Temporal reuse can be exploited in these stencil DAGs via operator fusion and spatial tiling. Apart from the constraints imposed by the GPU hardware resources, stencil DAGs present an additional optimization challenge: determination of tile shape/size and the amount of fusion across multiple stencils, while making effective use of a combination of shared memory and registers to enable a high degree of reuse. For example, one may prefer to fuse stencil operators with true dependence between them in order to exploit the temporal reuse, but the fusion may result in an increase in the halo region. Therefore, one needs to be cognizant of the ratio between data movement reduction and the increase in recomputations when fusing stencil operators. Similarly, while fusing stencil operators with no data dependence might not increase the recomputations, it may lead to higher register pressure and shared memory usage in the fused node.

For stencil DAGs with dozens of stencil operators, manually exploring the space of possible fusion candidates may be difficult. Even for time-iterated stencils which translate to a linear chain DAGs and where fusion is restricted to successive stencil operators, many previously proposed code generators [85, 43] circumvent the navigation of the fusion space by expecting the user to specify the degree of fusion. Frameworks like MODESTO [38] navigate the fusion search space for stencil DAGs by exploring all possible fusion candidates, which may be time consuming for stencils like *hypterm* [29] that accumulate the results in multiple output arrays, and hence have statements with no ordering constraints.

Register Pressure in Compute-Bound Stencils Unlike bandwidth-bound stencils that have low arithmetic intensity, compute-bound stencils lie at the other end of the spectrum with high arithmetic intensity. For example, consider an order-2

3D 125-point stencil with constant coefficients [8]. The stencil performs 134 floating-point operations with 3 memory accesses per output point. Thus, for double-precision computation, its arithmetic intensity is 5.58 FLOPS/byte, making it compute-bound on most GPU devices.

For box stencils, the arithmetic intensity increases with the increase in the stencil order (lower-order time-iterated stencils can be converted to high-order stencils by unrolling the time loop). In many mathematical stencil operators as those arising in numerical solution to PDE solvers, the solution accuracy increases with an increase in the stencil order. With the advent of compute-intensive GPU architectures, mathematicians and application developers have started exploring high-order stencils to achieve greater accuracy at the same data movement cost.

Theoretically, the performance of such high-order stencils must be limited by the compute resources on the GPU, and therefore, applying bandwidth optimizations like operator fusion will not improve the performance. One might simply perform spatial tiling followed by loop unrolling to exploit reuse in registers and obtain a performance close to the computational peak. However, in practice, the performance of such high-order stencil codes is impacted by the instruction scheduling and register allocation passes of the underlying compiler. These two are the most crucial backend passes in a compiler, and their objectives are often antagonistic – instruction scheduling prefers independent instructions scheduled in proximity to increase ILP, whereas register allocation prefers data-dependent instructions to be scheduled in proximity to shorten the live range. Register allocation is generally considered a practically solved problem. For most applications, the register allocation strategies in production compilers are very effective in controlling the number of loads/stores and register spills. However, existing register allocation strategies are not effective for computation patterns with a high degree of many-to-many data reuse, which is the case with high-order stencils. The excessive register pressure in such cases manifests as either register spills, or a performance loss due to a drop in occupancy, and consequently the inability to hide memory latency. For example, the *rhs4th3fort* kernel in SW4 [104] performs 687 floating-point operations relative to 64 bytes of data transfer, which gives it an arithmetic intensity of 10.73 FLOPs/byte. However, spills are only avoided by allocating the hardware maximum of 255 registers per thread on Tesla K40c device, and the achieved performance is merely 182 GFLOPS – 13% of the computational peak.

The inability of the register allocators in production compilers can be attributed to two factors: (a) Stock et al. [99] have demonstrated that associative reordering in the context of multi-core CPUs can help alleviate register pressure. The order of contributions from inputs points to an output point in a stencil computation is semantically irrelevant. However, the compilers do not fully exploit associative reordering to reduce register pressure; and (b) the compilers do not have a global perspective of the stencil DAG, and the access patterns to effectively reduce the live range interferences.

1.1 Key Contributions

In this dissertation, we develop optimization strategies for both bandwidth- and compute-bound stencil computations on GPUs. These optimizations are integrated into a domain specific language (DSL) based code generator named StencilGen. We advance the state-of-the-art in stencil optimizations on GPUs with the following contributions.

- Our first contribution is the development of efficient tiling schemes for memory bandwidth-bound 2D and 3D stencil computations on GPU. Unlike several previously proposed GPU code generators [85, 43, 113] that use symmetric 3D tiles for time tiling, we use streaming along one spatial dimension and symmetric overlapped 2D tiling in the other two spatial dimensions to enable significantly greater data reuse than 3D overlapped tiling. The effectiveness of streaming with spatial tiling was demonstrated by Micikevicius [67], and adapted by Nguyen et al. [72] for temporal tiling. However, their efforts were manual, and their strategy was only applicable to stencils with a specific access pattern. We exploit operator associativity to apply streaming optimization to a wider class of stencil computations.

- Our second contribution is the development of a model-driven approach to determine which stencil operators to fuse in a stencil DAG, since excessive fusion leads to increased redundant computation, increased data traffic to/from global memory, and increase in overhead from register spilling. Unlike the few models for fusion for stencil DAGs on GPUs [38, 116] that tend to explore a vast search space of fusion candidates and model the performance and register pressure of the fused kernel, our heuristic quickly prunes the search space by making greedy fusion choices, and recompiles some fusion candidates to get an accurate measure of register pressure.

- Our third contribution is the development of an effective pattern-driven global optimization strategy for instruction reordering to address the problem of register pressure in high-order stencils. The key idea behind the proposed instruction reordering approach is to model reuse in high-order stencil computations by using an abstraction of a DAG of trees with shared nodes/leaves, and exploit the fact that optimal scheduling to minimize registers for a single tree with distinct operands at the leaves is well known [93]. The DAG replaces the syntactical chain of contributions with an n-ary accumulation node. We thus devise a global statement reordering strategy for a DAG of trees with shared nodes that enables reduction of register pressure to improve performance.

- Our fourth contribution is to generalize the instruction reordering to generate efficient reordering schedules for multi-core CPUs. A significant difference between register allocation for CPU and GPU stems from the fact that the register pressure is reconfigurable at compile time on GPU. GPUs have massive thread level parallelism (TLP), to trade-off instruction level parallelism (ILP). However, the TLP in CPUs is severely restricted, so improving ILP is crucial to performance. We develop a list-based reordering strategy that uses multiple criteria, including affinities of non-live variables to those live in registers, the potentials of variables to fire operations and to release registers, as well as the potential of operations to increase ILP in the reordered schedule. We demonstrate the effectiveness of the reordering strategy for multi-core CPUs and Xeon Phi architecture.

While the techniques and optimizations discussed in this dissertation use CUDA [73] terminology and use NVIDIA GPUs as reference hardware, they are not limited to CUDA or NVIDIA GPUs. These optimizations can be implemented using any language that allows mapping computations to GPU threads, e.g., OpenCL [100], and are applicable to GPUs from other vendors, e.g., AMD.

1.2 Dissertation Outline

The dissertation is organized as follows. Chapter 2 gives a brief overview of the StencilGen language for specifying the stencil computations. Chapter 3 describes the tiling strategies and fusion heuristics to optimize a memory bandwidth-bound stencil DAG. Chapter 4 describes the DAG-based reordering framework that leverages operator associativity to perform register optimizations for high-order stencils on GPUs. Chapter 5 generalizes some of the ideas from Chapter 3 and Chapter 4 to develop a list-based instruction reordering framework named LARS. LARS can be applied to straight-line scientific codes to generate CPU codes with appropriate SIMD intrinsics. The reordering heuristics can be adapted for different stencil types and different optimization objectives. Finally, we discuss some of the future research directions in Chapter 7.

CHAPTER 2

Overview of StencilGen Language

This chapter describes the domain-specific language (DSL) for StencilGen that specifies the stencil computation. The use of a DSL enables the easy identification of the stencil patterns so that stencil-specific optimizations may be performed. Implementing high-performance GPU code for stencil computations requires both domain-specific and target-specific optimization strategies. For a compiler to automatically apply these strategies, the computation of interest must be analyzable to identify parts that represent stencil computations. Limitations in precise compiler analysis for programs in a general-purpose language, such as memory aliasing, loop trip count computation, induction variable analysis, etc. can prevent aggressive optimizations from being be performed for safety/correctness reasons. The use of a domain-specific language helps avoid these problems. The StencilGen language ensures that:

- the entire stencil computation (this includes time iterations, boundary conditions, and the operations applied on each data element) can be modeled;
- the data space accessed by stencils is fully described at the StencilGen level;
- general-purpose programming is allowed outside the stencil regions, i.e., one can use StencilGen both as a stand-alone, or embedded DSL.

Listing 2.1: A 2D Jacobi written in C language

```
double out[M][N], in[M][N], ONE_FIFTH = 0.2;
for (int i=1; i<=M-2; i++) {
  for (int j=1; j<=N-2; j++) {
    out[i][j] = ONE_FIFTH*(in[i-1][j]
       + in[i][j-1]+in[i][j]+in[i][j+1]+in[i+1][j]);
  }
}
for (int j=0; j<=N-1; j++)
  out[0][j] = a * in[0][j];
for (int j=0; j<=N-1; j++)
  out[M-1][j] = a * in[M-1][j];
for (int i=0; i<=M-1; i++)
  out[i][0] = a * in[i][0];
for (int i=0; i<=M-1; i++)
  out[i][N-1] = a * in[i][N-1];
```

An illustrative example We illustrate the main concepts of the StencilGen language using the example in Listing 2.2; the example expresses the 2D Jacobi stencil of Listing 2.1 in StencilGen language. The DSL program begins with a declaration of two integer parameters, `M` and `N` on line 1. These parameters are used to define the dimensions of the input and output arrays (line 3). Line 2 declares iterators, each of which will be mapped to a unique dimension of the computational loop nest. Note that the loops are explicit in the C code of Listing 2.1, but not explicitly defined in the DSL. This is because some of the loop nests will become implicitly data-parallel in the generated CUDA code. However, we assume that the iterators are only incremented by unit stride in the increment expression of the loops in the equivalent C program. The iterators must also be immutable within the body of the computational loop nest. All the arrays and scalars declared (e.g., `in` and `out` in line 3) will be passed as arguments to the host function. Line 5 specifies the arrays and scalars

Listing 2.2: Expressing the Jacobi stencil of Listing 2.1 in StencilGen language

```
 1  parameter M, N;
 2  iterator i, j;
 3  double in[M,N], out[M,N], a;
 4
 5  copyin in;
 6
 7  stencil five_point_avg (A, B) {
 8    double ONE_FIFTH = 0.2;
 9    B[i][j] = ONE_FIFTH*(A[i-1][j]
10      + A[i][j-1]+A[i][j]+A[i][j+1]+A[i+1][j]);
11  }
12
13  stencil boundary (A, B, a) {
14    B[i][j] = a * A[i][j];
15  }
16
17  [0   : 0  ][0   : N-1] : boundary (in, out, a);
18  [M-1 : M-1][0   : N-1] : boundary (in, out, a);
19  [0   : M-1][0   : 0  ] : boundary (in, out, a);
20  [0   : M-1][N-1 : N-1] : boundary (in, out, a);
21  [0   : M-1][0   : N-1] : five_point_avg (in, out);
22
23  copyout out;
```

that need to be copied from host to device. Lines 7–11 define a 5-point stencil named `five_pt_avg`. This stencil takes as arguments the input and output arrays/scalars used in the computation. `five_pt_avg` averages the 5 neighboring data elements from array `A` and writes the result to an element of array `B`. Lines 13–15 define a stencil named `boundary` that does a point-wise copy of the elements from array `A` to array `B` at the boundary. Lines 17–21 invoke the stencil functions over subsets of the problem domain. Finally, line 23 defines the arrays and scalars that need to be copied back from device to host.

Embedding StencilGen in C/C++ Every embedded section of StencilGen begins with `#pragma dsl begin` and is terminated by `#pragma dsl end`. Additional arguments can be supplied to the `begin` pragma to control the grid and block shape in the generated code. The main arguments are:

- `block:<comma separated list of integers>`: A comma separated list of integers specifying the GPU thread block shape/size for generated code. There must be one integer in the list for each spatial dimension of the grid in the StencilGen code.
- `unroll:<comma separated list of integers>`: A comma separated list of integers specifying the coarsening along each dimension. There must be one integer in the list for each spatial dimension of the grid in the StencilGen code.
- `stream:<string>`: A single string value specifying the dimension along which spatial streaming is to be performed in the generated code. In the absence of this argument, streaming is not performed.
- `shmem:<integer>`: A single literal value specifying the size of shared memory in `KB` available in the GPU architecture. In the absence of this argument, shared memory is not used in the generated code.

In order to allow easy integration of StencilGen into C/C++ programs, restrictions are imposed on how the data structures passed between the rest of the program and the StencilGen segment are declared. Contiguous memory regions of scalar types (e.g., `float`, `double`, etc.) are required for the fields storing the data; and integer types are required for parameters such as the grid size or the number of time iterations. It is also expected that the right-hand-side expressions of the assignment statements in the stencil body are side-effect-free. This implies that the right-hand-side of an assignment statement can contain standard side-effect-free math functions like sine, sqrt, etc.

More details about the grammar of StencilGen language are provided in Appendix A. Appendix B describes the complilation flags, and provides examples of the code generated using StencilGen.

CHAPTER 3

Bandwidth Optimizations for Stencils on GPU

3.1 Introduction

Stencil computations form the compute-intensive core of scientific applications in many domains, making their performance critical to the scientific community. Many recent efforts target optimization of stencils, including several domain-specific languages and frameworks [30, 18, 85, 43, 37, 108, 64, 38, 106, 42, 7, 36, 116, 87, 88]. Stencil computations exhibit a high degree of data parallelism, thereby making them attractive for execution on GPUs. However, many stencil computations are memory-bandwidth limited, unless temporal reuse is exploited across a sequence of stencils or across multiple time steps for time iterated stencils.

Tiling is a key transformation to enhance temporal reuse of data and thus reduce the amount of data transfer from/to global memory on a GPU. Time-tiled execution of stencil code on GPUs has been pursued by several efforts [72, 67, 86, 113, 84, 127, 108, 64, 38, 18, 85, 43, 37, 36, 87, 116, 24, 66, 19, 79, 126, 9, 112, 114, 118, 117, 63], to advance the state-of-the-art so that very high performance can currently be realized using stencil-specific and GPU-specific optimization techniques. In this chapter, we present a detailed analysis and description of domain-specific and GPU-target-specific optimization strategies that enable high performance with stencil computations.

Several considerations are important in achieving high performance on GPUs: movement of contiguous chunks of data (coalesced accesses) from/to global memory; reuse of data in registers and shared-memory; sufficient concurrency, orders of magnitude larger than the number of physical cores; and minimization of control-flow divergence among threads. The total amount of on-chip shared memory in each SM (Streaming Multiprocessor) of a GPU is very limited, typically under 100 Kbytes. As elaborated later, the low shared-memory capacity limits the maximum number of time steps in a time-tile and also limits the number of concurrently active thread blocks in each SM. Low occupancy can leads to low performance due to inability to effectively overlap memory access latency with sufficient operations to execute across the active warps. However, judicious exploitation of the data access pattern of stencil computations, combined with associative reordering of the stencil operations where appropriate, can enable the use of GPU registers to offload data from shared memory, thereby enhancing data reuse and/or occupancy. Since the register file capacity per SM is larger than shared memory capacity on most modern GPUs (a trend that is expected to continue), it alleviates the constraints on occupancy imposed by shared memory usage. The low access latency of registers further improves the performance of the code. In this chapter, we present details on the optimization of register and shared-memory resources for stencil computations. In combination with streamed execution (elaborated later), along an arbitrarily long tile along one of the spatial dimensions, it is possible to achieve higher GPU performance than the use of uniform tile sizes along all spatial dimensions.

The rest of the chapter is organized as follows. Section 3.2 details the impact of the GPU hardware constraints on the stencil optimization strategy. An overview of the overlapped-tiling strategy and GPU code generation algorithms for overlapped tiling of stencil computations are described in Sections 3.3 and 3.4. Section 3.5 describes fusion across multiple stencils. Section 3.7 presents an experimental evaluation of the described stencil code generation approach, comparing it with several current general-purpose and special-purpose code generators and optimizers for multicore processors, manycore processors, and GPUs. We summarize in Section 3.8.

3.2 GPU Constraints for Stencil Computation

This section discusses architectural constraints to be considered when optimizing stencil computations on GPUs. The basic computational unit of a GPU is a thread. Going up the hierarchy, threads are grouped into thread blocks, and thread blocks are grouped into a grid. A program explicitly specifies this hierarchy of grids and thread blocks as kernel launch parameters. Threads within a thread block are executed on the same *streaming multiprocessor* (SM); they can exchange data via shared memory, and synchronize among themselves.

In order to be efficient, tiling for GPUs must address the following:

- Access global memory in a coalesced manner,
- Achieve sufficient concurrency to tolerate memory access latency, and
- Judiciously use faster storage, like shared memory and registers, for caching data.

Shared memory has lower access latency than global memory, and is well suited to cache data that is accessed by multiple threads in a thread block. Registers are local to a thread, and therefore well suited to cache data that is accessed by a single thread. Registers have lower access latency than shared/global memory, and are more plentiful than shared memory in most GPU architectures. Therefore, it is often beneficial to offload some storage from shared memory to registers. However, excessive register usage may result in either lower occupancy or expensive spills.

Since these aspects are tightly coupled to the underlying hardware, the kernel launch parameters need to be tuned for the specific GPU architecture on which the program is executed. In this section, we present some general constraints on thread block and grid size for spatial tiling of a d dimensional stencil computation, over an N^d input domain. Typically, each point of the iteration space is executed by a thread in the GPU. We assume that the stencil is of order k along each dimension.

3.2.1 Constraints on block and grid size

Every GPU has a hardware limit on the maximum number of threads per SM (T_{sm}), maximum number of threads per thread block (T_b), maximum shared memory per SM (M_{sm}), maximum number of concurrently active thread blocks per SM (B_{sm}), and register file size per SM (R_{sm}). For instance, $T_{sm} = 2048$, $T_b = 1024$, $M_{sm} =$ 48KB, and $B_{sm} = 16$, and $R_{sm} = 65536$ for the Nvidia Kepler K20c. The threads in an SM can be grouped in various ways (e.g., 2 blocks of 1024 threads, 16 blocks of 128 threads, etc.). The threads in a block are scheduled in groups of 32, called a *warp*.

When a thread in a warp accesses global memory, the warp stalls for hundreds of clock cycles due to high memory latency. In order to hide the latency, the SM can switch to another warp in the ready state. There must be a sufficient number of instructions across all active warps to keep the SM busy until the memory request of the stalled warp is served. One extreme is to ensure that the maximum possible number of threads in an SM are active. If we reach the hardware limit of $\frac{T_{sm}}{32}$ warps per SM, then we achieve maximum *occupancy*, the ratio of active warps to maximum theoretical active warps per SM.

Stencil computations can benefit from spatial reuse if the data is cached in shared memory. The access latency of shared memory is orders of magnitude lower than global memory, but it puts an additional constraint on the number of blocks that can be concurrently active on an SM. If each block of size sz_b uses $c.sz_b$ bytes of shared memory, then we can have no more than $\frac{M_{sm}}{c.sz_b}$ active blocks per SM. If c is large, then fewer blocks can be active per SM, which can potentially reduce occupancy.

Registers represent the fastest storage resource available to the thread. The maximum number of registers usable by a thread is constrained by the hardware. If all the threads in an SM are to be active, the number of registers must be bounded by $\frac{R_{sm}}{T_{sm}}$. In general, if a thread uses t_{reg} registers, then the maximum number of active threads per SM is upper-bounded by $min\left(T_{sm}, \frac{R_{sm}}{t_{reg}}\right)$.

Taking all these factors into account, the maximum number of active thread blocks per SM, max_b is upper-bounded by $max_b \leq min\left(B_{sm}, \frac{T_{sm}}{sz_b}, \frac{M_{sm}}{c.sz_b}, \frac{R_{sm}}{t_{reg}.sz_b}\right)$. Unless thread coarsening is utilized to increase instruction level parallelism (ILP), the block size sz_b must be chosen to maximize occupancy, and consequently the thread level parallelism (TLP), i.e., $sz_b \times max_b$ must be as close to T_{sm} as possible. If there

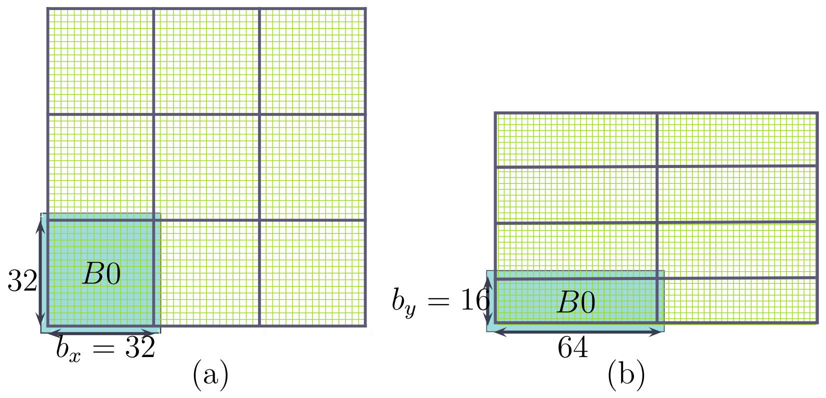


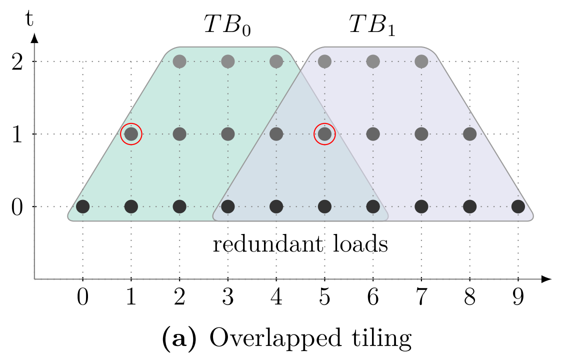
Figure 3.1: Different thread block configurations for a fixed thread block size

are K SMs in the GPU, then the grid size must be at least $K \times max_b$ to maximize concurrency.

Coalescing of global memory accesses is important since it reduces the total number of memory transactions, thereby minimizing required DRAM bandwidth. To benefit from coalescing, the fastest varying dimension of the thread block is aligned to the fastest varying dimension of the input domain, and is usually chosen to be a multiple (or factor) of warp size, i.e. $mod(sz_b, 32) = 0$, or $mod(32, sz_b) = 0$.

3.2.2 Partitioning threads across thread block dimension

Given a 2D thread block of size sz_b, we can vary the number of threads along x and y dimensions to get different thread block configurations. The performance of a stencil computation can vary depending on the chosen configuration. To illustrate, let $sz_b = b_x \times b_y$. Two possible configurations for $sz_b = 1024$ are shown in Figure 3.1. For configuration *(a)*, $b_x = b_y = 32$. For configuration *(b)*, $b_x = 64, b_y = 16$. For both the configurations, b_x is a multiple of warp size to benefit from global memory coalescing.



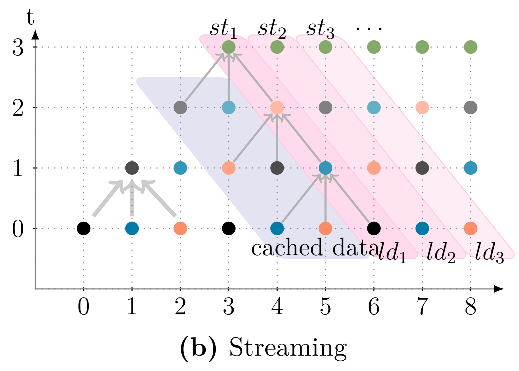


Figure 3.2: Different tiling schemes for 3-point 1D Jacobi computation

If an order-k stencil over an N^2 domain uses configuration *(a)*, the number of global reads is $N^2(32 + 2k)^2/1024$. For configuration *(b)*, the number of global reads is $N^2(64 + 2k)(16 + 2k)/1024$. For any value of k, we see that configuration *(a)* requires fewer global read transactions for performing the same number of arithmetic operations.

3.3 Overlapped Tiling with Streaming

With stencil computations, both input values and values computed at each time step are used multiple times due to the stencil access pattern. In order to achieve high performance, it is critical that these values are retained in faster shared memory and registers instead of repeated accesses to slower global memory. Tiling is a key transformation to address this issue. However, simply slicing the input domain and assigning disjoint slices (data tiles) to different thread blocks is not feasible for time tiled stencil computations. For every tile, computing the boundary values at time step t requires some data computed at time step $t - 1$ from neighboring tiles. This data is referred to as the *halo region* or the *ghost zone*.

3.3.1 Overlapped Tiling

A tiling technique that has been shown to be effective for GPUs is overlapped tiling [55, 43, 86], illustrated by Figure 3.2a. It eliminates inter-tile dependence by using redundant "overlapped" computation and data loads. The extent of input data read per tile is increased so that no inter-tile synchronization or communication is needed to compute the boundary values at each time step in the time-tile.

The extent of redundant computation depends on the block size, number of overlapping dimensions, the order of the stencil, and the time tile size. For example, blocks TB_0 and TB_1 in Figure 3.2a compute three output values. There are four redundant reads and two redundant computations. The fraction of redundant reads and computations can be decreased by increasing the spatial tile size. Conversely, If the time tile size is increased, the number of redundant reads and computations increase. The redundancy increases with increase in stencil order as well. While the amount of redundant computation is often tolerable for low-order 2D stencils, it can be excessive for 3D stencils. Consider a thread block that computes an $8^3 = 512$ output block for two time steps of a 3D first-order stencil. Each thread block would need to read a $12^3 = 1728$ block of the input domain, out of which almost half are read redundantly by another thread block as well. The number of points at the intermediate time step computed by the block would be $10^3 = 1000$. The amount of redundant computation would be $1000 - 512 = 488$, i.e., half of the computation performed by each block is performed by another block as well. This problem of redundancy for 3D stencils get worse with an increase in stencil order and time tile size.

3.3.2 Streaming

Since overlapped tiling for all the dimensions of a 3D stencil is not an effective approach, an alternative is to tile two of the dimensions, and ***stream*** along one dimension of the computation. The discussion below first describes the idea of streaming along a dimension of the computation and then explains how it is used in combination with overlapped tiling to address the problem of redundancy for 3D stencils.

Figure 3.2b describes streaming through x dimension for a 1D Jacobi stencil with time tile of 3 time steps. $t = 0$ is the initial state from which the input values are read at $t = 1$. At each time step, intermediate output values are computed using the data from the previous time step. The final output is computed at $t = 3$. Every time step requires a different set of buffers. In the figure, different colored dots at time step $0-2$ represent distinct shared memory buffers. The global memory is represented by green dots. For a 1D Jacobi computation, three distinct memory buffers are required at each time step, shown by different shades of black, blue, and orange dots. In general, an order-k stencil needs $2k + 1$ distinct buffers per time step.

After an initial prologue, an iteration of the computation loads a value from global memory at $t = 0$, computes intermediate results at each time step, and finally writes out an output value to global memory. To observe an instance of this computation, assume that the input values at $x = \{4, 5, 6\}$ are cached in shared memory buffers $\{blue_0, orange_0, black_0\}$ respectively before their use (each buffer is labeled as $color_t$). At time step 1, $x = 5$ can be computed by reading from these buffers, and stored in $blue_1$. Assuming that the values computed at $x = \{3, 4\}$ are available in $orange_1$ and $black_1$, $x = 4$ can be computed at time step 2 and the result can be stored the $orange_2$ buffer. Once again, assuming that the values computed at $x = \{2, 3\}$ are available in $black_2$ and $blue_2$, $x = 3$ can be computed at time step 3, and the result can be written out to global memory.

In the subsequent iteration, the input values at $x = \{5, 6\}$ are already buffered due to the data overlap from the stencil point on the left. However, the value $x = 4$ is no longer required in the computation, and its buffer can be reused. Thus, we can store the next input value from $x = 7$ in $blue_0$. This reuse holds for each time step. Two observations can be made from the computation: (1) For each input point x_i read in iteration i, output point x_{i-3} can be computed. (2) Only 3 buffers are needed per time step for the entire computation, irrespective of the memory footprint of the tile size. In general, for an order-k stencil, $2k + 1$ buffers are needed per time step. The buffer storing the oldest value in iteration i can always be reused to store the newest value at iteration $i + 1$. The scheme is called streaming since a sliding window is used to minimize the shared memory footprint, and the consumption of input is moderated at one point per iteration.

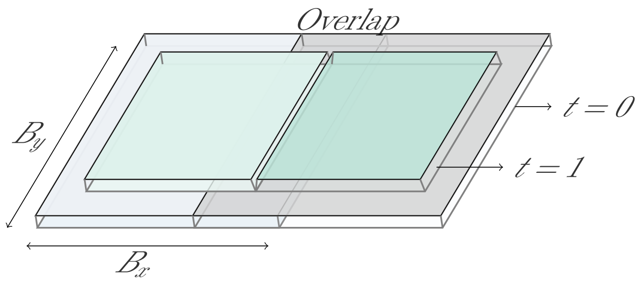
When a thread block computes all the points along x axis serially in a streaming fashion, we have ***serial-streaming***. An alternative would be to block the iteration space along x axis, and assign each partition to different thread blocks. Each thread block will still serially stream through its share of iteration space, but all the blocks can concurrently stream through their share of iteration space. We will refer to such streaming as ***concurrent-streaming***.

Streaming can be extended to 2D or 3D Jacobi stencils by interpreting a point in Figure 3.2b as a line or a plane, respectively. For large problem size, fitting $2k + 1$ lines (or planes) in entirety in shared memory may be infeasible. The line (or plane) then should be blocked, with the block size carefully chosen depending on the time tile size and hardware constraints. If the time tile size is 1, then only spatial reuse is exploited. For an N^3 domain, a point in Figure 3.2b can be interpreted as a blocked plane of size B^2, and serial-streaming can be performed through N blocked planes in the third dimension. Such spatial blocking is known as 2.5-D blocking, and the corresponding time tiled blocking is known as 3.5-D blocking [72].

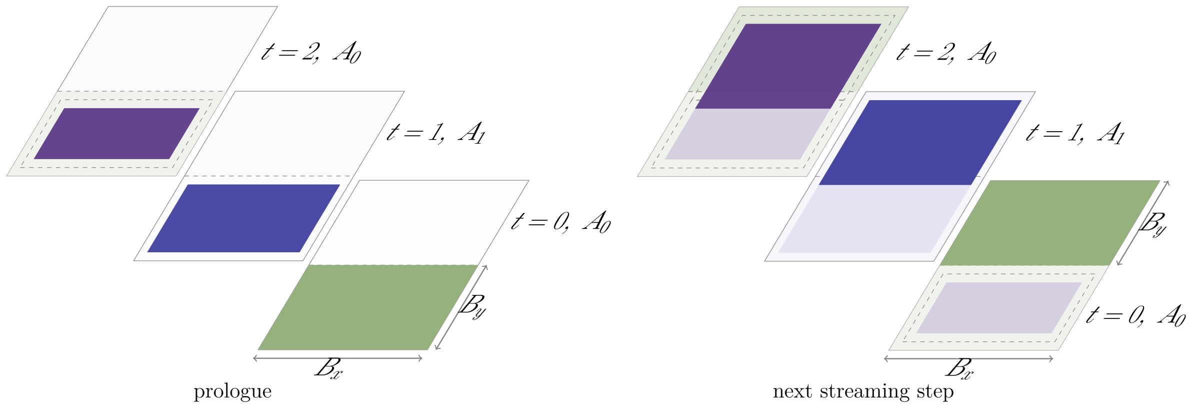
Since streaming does not perform any redundant computation and uses very limited amount of shared memory, it is an attractive option to combine with overlapped tiling. For a 3D computation, using overlapped tiling along two dimensions and streaming along the third would reduce the amount of redundant operations (loads and computations). The downside of streaming is that it serializes the computation along one dimension of the problem. If the problem size is not big enough to keep all the SMs busy, this would result in a performance degradation. In such cases concurrent streaming can be used to increase the degree of concurrency in the problem to keep all the SMs busy. Section 3.4 describes a systematic approach to decide the best tiling strategy for a given problem.

3.4 Resource Optimization Strategies

This section discusses different factors to be considered in determining a tiling strategy that is best suited for a given stencil computation. Section 3.4.1 describes time tiled implementations of overlapped tiling for 2D stencils. Section 3.4.2 characterizes 3D stencils based on their access patterns, and describes optimization techniques for them. There is also a discussion of implementation details that might not be directly related to the tiling scheme, but are important to achieve good performance. Further, some of these techniques use explicit registers for storage, thereby



(a) Overlapped tiling along x for a 2D block



(b) A block's computation in optimized version of streaming along y

Figure 3.3: Overlapped tiling for 2D stencils

increasing the per-thread register pressure and necessitating a trade-off between occupancy and per-thread register usage. The NVCC compiler provides a compile-time flag `-maxrregcount=n` that limits the number of registers per thread to n. This can be used to balance the trade-off between occupancy and register spills. Let us consider a stencil of order k, with time tile size T, and all operations in single precision. An tiled code must be tuned for different architectures; the discussion assumes parameters of an Nvidia Kepler K20c as the target GPU.

3.4.1 Overlapped Tiling for 2D stencils

Overlapped tiling described in Section 3.3.1 can be extended to any d-dimensional domain. Figure 3.3a shows overlapped tiling for a 2D thread block along the x dimension.

Overlapped tiling + serial streaming

Given an N^2 input domain, streaming can eliminate redundant computations along y dimension, and overlapped tiling along x dimension can achieve concurrency in execution. The input is partitioned into overlapping strips of size $B_x \times N$. In the steady state, a $B_x \times B_y$ thread block reads B_y input lines along y dimension per iteration to compute B_y points of the iteration space. Each thread systematically executes the iteration space at a stride of B_y in the y dimension. In this mode, each thread block has to store $T.(B_y + 2k)$ lines in the shared memory. With this information, it is possible make choices for block dimensionality. To simplify the computations, let $k = 1$ and $T = 4$.

- $B_y = 1$, *i.e. 1D block*: To achieve maximum occupancy, the block size must be $\frac{2048}{16} = 128$. In each iteration, a block reads one line from input at $t = 0$ to compute one line of output at $t = 4$. For this, it needs 6144 bytes of shared memory. From Section 3.2.1, it follows that an SM can have at most $min\left(16, \frac{2048}{128}, \frac{48KB}{6144B}\right) = 8$ blocks, a 50% loss of occupancy.
- $B_y = 32$, *i.e. 2D block*: Two blocks of size 32×32 can theoretically be active per SM. Since a block now operates on a chunk of 32 lines instead of 1, it needs $18KB$ of shared memory. Each SM can practically have $min\left(16, \frac{2048}{1024}, \frac{48KB}{18KB}\right) = 2$ active blocks, which implies maximum occupancy.

Clearly, better occupancy is achieved with 2D blocks. For a $B_x \times B_y$ block, the computation proceeds as shown in Figure 3.2b: in each iteration after prologue, B_y lines are computed at time step t by reading $B_y + 2k$ lines from the shared memory buffer at time step $t - 1$. A ***sliding-window*** approach is used where after each iteration, B_y oldest lines in the buffer can be reused to cache the new lines. The buffer itself can be implemented as a circular array, and row performs modulo operations to find the top and bottom k rows.

Modulo operations are costly on GPUs since they compile to multiple instructions in the assembly code. By making $B_y + 2k$ a power of 2, it is possible to replace the modulo operator by a bitwise operator[1] which has a very high throughput.

Ping-Pong approach with 2D block + streaming

A modification of the sliding window approach is the ***ping-pong buffer*** approach, as illustrated in Figure 3.3b. It requires two shared memory buffers A_0 and A_1. All even time steps read from buffer A_0 and write to buffer A_1, and all odd time steps read from A_1 and write to A_0. In the prologue of Figure 3.3b, at time step $t = 0$, a block reads B_y lines to fill the first half of the buffer A_0. The prologue computes $B_y - 2tk$ lines at each time step $t > 0$. In the next streaming step, the same block reads the next B_y lines to fill the second half of A_0, computing $B_y + 2k$ at each time step. This continues, with the even streaming steps using the first half of A_0, and the odd steps using the second half.

This approach not only avoids the use of expensive modulus operation by setting B_y as a power of 2, but also reduces the shared memory requirement for a $B_x \times B_y$ block from $T.(B_y + 2k).B_x$ for sliding-window approach to $4B_yB_x$. Further, the amount of

[1]$a \bmod b \equiv a \,\&\, (b - 1)$ if b is a power of 2

shared memory required is independent of the time tile size. If thread blocks of size (32×16) are created, then each block will use 8192 bytes of shared memory. Each SM can concurrently schedule at most $min\left(16, \frac{2048}{512}, \frac{48KB}{8192B}\right) = 4$ blocks, utilizing all the available threads per SM. Assuming that there is enough concurrency to keep all SMs busy, this optimized version outperforms the traditional 2D overlapped tiling scheme.

Overlapped tiling with 1D block + registers + concurrent streaming

Overlapped tiling + streaming with 1D thread blocks was not optimal because the high shared memory usage lowered the occupancy. A stencil is *diagonal-access free* if the access offset (x_0, y_0, z_0) from one plane along z dimension to other plane is strictly of the form $(0, 0, z_0)$. If the stencil is diagonal-access free, then the shared memory requirement can be lowered by using registers to cache the $2k$ accesses to lines [67]. For a thread block of 128 threads, $k = 1$ and $T = 4$, a block now needs 8 registers and 2048 bytes of shared memory. With this optimization, the number of feasible active blocks per SM is $min\left(16, \frac{2048}{128}, \frac{48KB}{2048B}\right) = 16$ blocks. While the occupancy is maximized, this strategy suffers from low concurrency. To keep all 13 SMs of K20c busy, at least $13 \times 16 = 208$ thread blocks are needed. However, even a large input domain of size 8192^2 can only be partitioned into $\frac{8192}{128} = 64$ blocks. This was not a problem with the ping-pong approach since it needed only $13 \times 4 = 52$ blocks to achieve full concurrency, and the input domain for it was partitioned into 256 blocks.

The simplest way to increase concurrency is to overlap-tile the input domain along y dimension as well. It is only necessary to partition the domain into $\left\lceil \frac{208}{64} \right\rceil = 4$

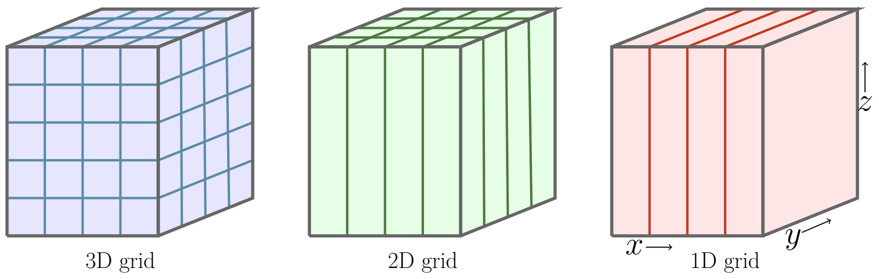


Figure 3.4: Various grid dimensionalities for a 3D input

blocks along the y dimension. Since the thread block is 1D and each tile in the input grid is 2D, streaming is performed within a tile.

This version incurs some volume of redundant computation and bandwidth along y dimension when compared to the ping-pong approach, but the overhead of that is negligible compared to the reduction in redundancy and bandwidth along x dimension due to larger *blockDim.x* (128 vs. 32). This implementation also benefits from the low access latency of the registers.

3.4.2 Overlapped Tiling for 3D stencils

Efficient 3D tiling must strive to reduce the extra data transfers involved in reloading ghost region across tiles. Figure 3.4 shows the possible grid dimensionality for a 3D domain. 3D grid requires extra memory bandwidth for redundantly loading the same halo regions for neighboring blocks along z dimension. 2D grid entails streaming through the non-partitioned dimension. For 1D grid, streaming is done through one non-partitioned dimension, and tiling is used for the other non-partitioned dimension using parallelogram tiling. However, 1D grid might not provide enough concurrency to achieve good performance.

For higher dimensional domain, streaming simplifies the tiling algorithm and code generation. Since the streamed dimension is not tiled, a d-dimensional domain can be tiled using the same algorithm that tiles $d-1$ dimensions. Our tiling implementation for 3D stencils partitions the input domain as a 2D grid, and assigns a 2D thread block to each partition that streams through the unpartitioned dimension. Concurrency is achieved by using 2D overlapped tiling (Section 3.4.1) on the partitioned dimensions. Note that streaming cannot be done through x dimension for 3D stencils, since this would entail non coalesced accesses while loading data from a yz plane. Without loss of generality, consider z to be the streaming dimension in this subsection.

Streaming in a time tiled 3D stencil requires $T.(2k+1)$ planes to be in shared memory. When $T = 4$ and $k = 1$, a thread block of size 32×32 will need $48KB$ shared memory. Since this is the total available shared memory, an SM can have no more than one active block, resulting in 50% occupancy. If $k = 2$, then the required shared memory exceeds the hardware limit. Micikevicius [67] and Nguyen et al. [72] use registers to offload the caching of some planes from shared memory to registers for spatial and time tiling. We discuss the details of implementing a time tiled code with shared memory and registers. The discussion will be independent of the tiling scheme used across thread blocks to achieve concurrency, as it is orthogonal to the streaming optimizations that are local to a block.

Overlapped tiling with 2D blocks + streaming + registers

In the scenario above with $k = 1$ and block size 32×32, if the stencil is diagonal-access free, then the per thread register pressure can be increased by $2Tk$, which reduces the shared memory requirement to 16KB. With this trade off, an SM can have two active blocks, achieving maximum occupancy. If there are register spills due

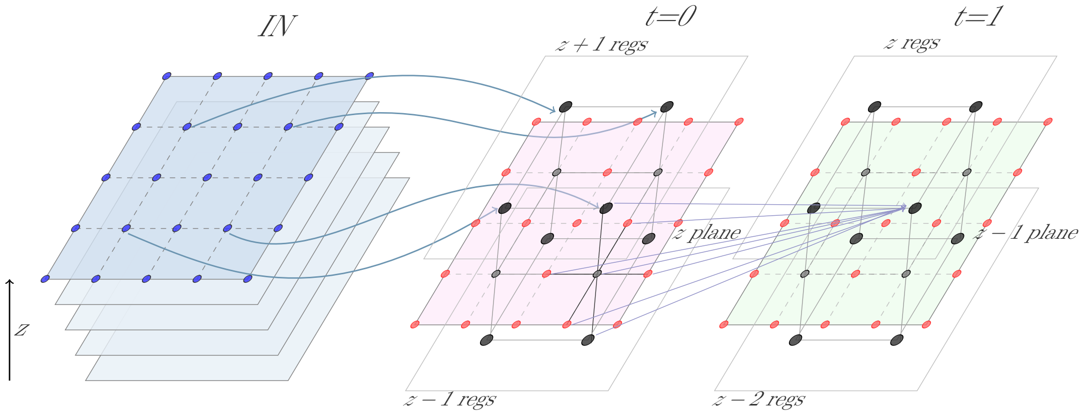


Figure 3.5: Streaming along z dimension for 7-point 3D Jacobi

to the increased register pressure, then the value of T can be reduced to alleviate the register pressure. Figure 3.5 shows one time step of the time tiled 7-point 3D stencil using registers.

For the 7-point stencil, the resources involved in the computation are shared memory buffer $A[T]$, and registers $r_p[T]$, $r_m[T]$. In each iteration i, threads in a block read a point from input plane z_{i+1} into $r_p[0]$. The block computes plane z_i at time step 1 using $A[0], r_p[0]$, and $r_m[0]$, and stores this computed value in $r_p[1]$. Using $A[1], r_p[1]$, and $r_m[1]$, the block computes plane z_{i-1} at time step 2. Proceeding this way, at time step T, plane z_{i-T} of the output array is computed. After each iteration, a data shift $(r_p \rightarrow A \rightarrow r_m)$ is performed at each time step, freeing r_p to store new values.

An implementation sketch of the scheme is presented in Algorithm 3.1. The algorithm is presented independently of the tiling scheme for the other two dimensions. The initializations at Line 1 depend on the tiling schemes for x and y dimensions. Function *compute_stencil()* returns the output of applying stencil computation to a point.

Algorithm 3.1: Stream (IN, T): Streaming along the slowest-varying dimension, and using registers for storage in lieu of shared memory for a 3D order-1 stencil

```
   Input  : $IN$ : input array, $T$ : time tile size
   Output: $OUT$ : output array
 1 $A[T]$ : shared memory buffer for each time step;
 2 $r_p[T]$,$r_m[T]$ : registers storing $z+1$ and $z-1$ planes at each time step;
   // Initialization
 3 $A[0] \leftarrow$ load plane $IN[0][\ldots][\ldots]$;
 4 $r_p[0] \leftarrow$ load plane $IN[1][\ldots][\ldots]$;
 5 for each $z$ from 2 to $N-2$ do
       // Shift data
 6     $r_m[0] = A[0]$;
 7     $A[0] = r_p[0]$;
 8     $r_p[0]$ = load plane $IN[z][\ldots][\ldots]$;
 9     ___syncthreads ();
10     for $t$ from 1 to $T$ do
           // Shift data, put result in $r_p$
11         $r_m[t] = A[t]$;
12         $A[t] = r_p[t]$;
13         $r_p[t]$ = compute_stencil ($A[t-1]$, $r_p[t-1]$, $r_m[t-1]$);
14         ___syncthreads ();
15     $OUT[z-kT][\ldots][\ldots] = r_p[T]$;
```

Algorithm 3.1 can be generalized to all stencils where each plane accesses only one point per plane from other planes along z dimension. If such a stencil accesses a diagonal point (x_0, y_0, z_0), then appropriate shuffle intrinsics will have to be inserted in the code, so that each thread maintains a correct r_p and r_m value.

Optimization for associative stencils

For stencils that access more than one point per plane from other planes along z dimension, the streaming + registers version will incur high register pressure. There is also some redundancy in the values stored in the registers of neighboring threads.

Listing 3.1: An illustrative 7-point 2D associative stencil that accesses 2 points from planes along y dimension

```
for (y=1; y<N-1; y++) {
   for (x=1; x<N-1; x++) {
       B[y][x] = c0*(A[y-1][x-2] + A[y-1][x+2]) +
              c1*(A[y][x-1] + A[y][x] + A[y][x+1]) +
                c2*(A[y+1][x-2] + A[y+1][x+2]);
   }
}
```

Listing 3.2: Rewriting the stencil of Listing 3.1: reading one input plane at a time, and accumulating its contribution at different output points

```
for (y=0; y<N; y++) {
   for (x=0; x<N; x++) {
       B[y+1][x] += c0*(A[y][x-2] + A[y][x+2]);
       B[y][x] += c1*(A[y][x-1] + A[y][x] + A[y][x+1]);
       B[y-1][x] += c2*(A[y][x-2] + A[y][x+2]);
   }
}
```

For a 27-point 3D stencil with $T = 2$, the number of registers needed per thread is 36, which will inadvertently lead to register spills if one intends to maximize occupancy.

If such a stencil is associative, optimization can be applied to reduce the number of registers required at each time step to just 1 for each plane accessed, bringing down the total number of registers per thread to $2Tk+1$. A similar optimization strategy is used by Stock et al. [99] in the context of high-order stencils on multi-core CPUs. An example of such a stencil is shown in Listing 3.1. The associativity of addition and multiplication can be exploited to convert the stencil into an accumulation stencil, as shown in Listing 3.2.

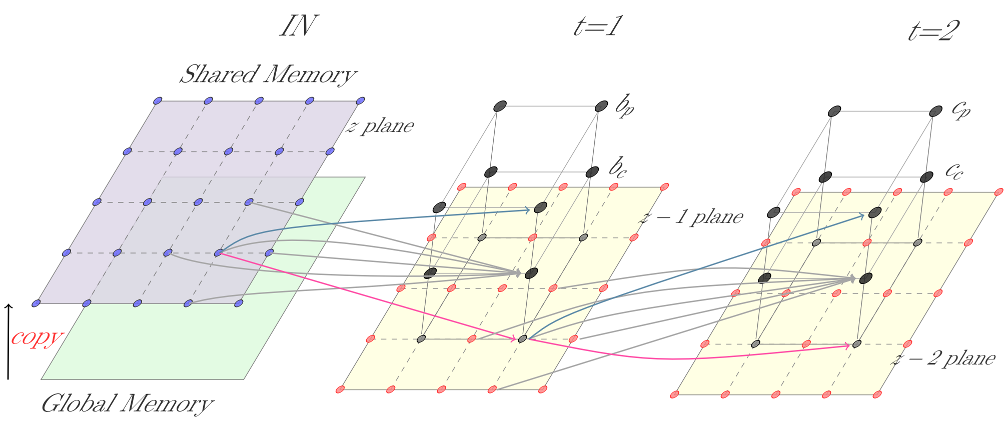


Figure 3.6: Associative reordering to scatter contributions from an input plane to output accumulation registers for 7-point 3D Jacobi

Figure 3.6 shows the time tiling using accumulative registers for associative stencils. Unlike the tiling scheme of 3.5, which used registers to cache input values, the tiling scheme shown in Figure 3.6 uses registers to accumulate the output values.

The output value for planes $z_0 + 1$ and z_0 are accumulated in registers $r_p[T]$ and $r_c[T]$, and the output value for plane $z_0 - 1$ is accumulated in shared memory buffer $A[T]$. In each iteration i, the input plane z_i is read into $A[0]$. From it, the contributions to output plane z_{i-1}, z_i, and z_{i+1} are accumulated in $r_p[1], r_c[1]$, and $A[1]$ at time step 1. All remaining time steps repeat the same process of accumulating output values using $A[t-1]$ as the input plane. After T time steps, all contributions to plane z_{i-T} would have been accumulated in $A[T]$. After each iteration, the data is shifted ($r_p \rightarrow r_c \rightarrow A$), freeing r_p to accumulate the contributions for next output plane.

Algorithm 3.2: Associative-Stream (IN, T): Streaming along the slowest-varying dimension, and scattering the contributions from an input plane to output accumulation registers via associative reordering for a 3D order-1 stencil

```
Input  : $IN$ : input array, $T$ : time tile size
Output: $OUT$ : output array
1  $A[T]$ : shared memory buffer for each time step;
2  $r_p[T]$, $r_c[T]$ : registers storing $z+1$ and $z$ planes at each time step;
3  for each $z$ from 0 to $N-1$ do
4      $A[0] \leftarrow$ load plane $IN[z][\ldots][\ldots]$;
5      ___syncthreads ();
6      for $t$ from 1 to $T$ do
           // Scatter contributions from $A[t-1]$
7          $r_p[t] \leftarrow$ bottom_plane_contrib ($A[t$-1]);
8          $r_c[t]$ += mid_plane_contrib ($A[t$-1]);
9          $A[t]$ += top_plane_contrib ($A[t$-1]);
10         ___syncthreads ();
11     $OUT[z-kT][\ldots][\ldots] = A[t]$;
12     ___syncthreads ();
       // Shift data
13     for $t$ from 1 to $T$ do
14         $A[t] \leftarrow r_c[t]$;
15         $r_c[t] \leftarrow r_p[t]$;
```

Algorithm 3.2 presents an implementation sketch for this approach. An optimized implementation of the algorithm uses only $2Tk+1$ registers and T shared memory buffers.

3.5 Fusion Heuristics

Multi-statement stencils and image processing pipelines apply a sequence of stencil operators, identified by the stencil statements, on a set of input domains [71, 85]. Such stencil computations can be represented by a directed acyclic graph (DAG) in which nodes represent stencil operators and incoming edges represent inputs to these operators. Time iterated stencils can also be rewritten as a DAG of stencil operators via explicit unrolling of the time loop. Fusion of stencil operators in a DAG can enhance data reuse. The key idea behind such fusion is to fuse the nodes in the DAG with common predecessors (i.e., stencil operators that access common input arrays), or a chain of nodes in the DAG (i.e., stencil operators that have producer-consumer dependences) so that temporal locality and the per-load data reuse is maximized. This section describes a simple, yet effective greedy fusion heuristic to fuse stencil operators in a DAG.

Fusion of any two operators in the stencil DAG is valid if it does not violate any data dependences. Without violating dependences, different nodes in a DAG can be fused to get different valid schedules. The performance of each schedule can vary depending on the profitability of fusion. The space of all valid execution schedules can be vast and exploring this space presents a significant challenge for any fusion algorithm. To avoid such exploration, a greedy algorithm can be used to fuse nodes in the DAG with the intent to minimize a chosen objective function. From the initial stencil DAG $D_g = (V, E)$, the goal is to create a resultant *fused* directed graph $D_f = (V_f, E_f)$ such that each $v_f \in V_f$ is a *convex partition* of nodes from V. A convex partition is a "macro-node" comprising nodes from V, with no dependence chain leaving and re-entering any macro-node. Just as convex-shaped tiles of an iteration space can execute atomically, the macro-nodes computed by the fusion algorithm can be executed atomically without violating dependences. The convex partitioning therefore guarantees that D_f is acyclic. Only nodes whose fusion is deemed profitable by the objective function are combined together in v_f. A single GPU kernel is generated for each node in v_f. No global memory transactions are needed for domains whose definition and use do not cross partition boundaries.

3.5.1 Overview of a Greedy Fusion Algorithm

The fusion algorithm consists of the following steps:

Step 1: For each stencil operator, compute the amount of shared memory and registers that will be used to cache the data (Section 3.5.2).

Step 2: Identify pair of stencil operators that can be fused without violating dependences (Section 3.5.3).

Step 3: For each stencil operator pair, compute the shared memory and registers used post fusion, based on the individual resource usage computed in step 1 (Section 3.5.4).

Step 4: Based on the resource usage of the fused node computed in step 3, create a profitability metric that encodes the impact of fusion on the GPU resources, and the data movement volume (Section 3.5.5).

Step 5: Define a custom sort to order the profitability metrics of various stencil operator pairs, and choose to fuse the operators of the most profitable pair into a macro-node (Section 3.5.6). Update the stencil DAG and the dependence graph, and repeat again from Step 1, until no more stencil operator pairs can be fused

Once the fusion algorithm creates the fused stencil DAG D_f, an optimized CUDA kernel can be generated for each node of D_f using the algorithms described in Section 3.4.

3.5.2 Computing Resource Requirements of a Stencil Operator

Without loss of generality, assume that streaming is done through the outermost dimension of a 3D domain. As discussed in Section 3.4, to determine the resource requirement of a stencil operator that is order-k along the streaming dimension, it is necessary to reason about the storage type of the $2k + 1$ accessed planes. The following chain of reasoning can be used:

(a) If the computation of an output element at (z_0, y_0, x_0) only uses a single input element at (z_r, y_0, x_0) from an input plane z_r, then that input element is stored in an explicit register (i.e., storage type of z_r is register). Otherwise, z_r is cached in shared memory

(b) An output element is written to an explicit register (i.e., its storage type is register) if it is an accumulation, or it is not the last assignment to a copy-out array. For the last assignment to any copy-out array, the storage type is global memory.

Here, an *explicit* register refers to a scalar temporary variable created to hold an array element instead of using shared memory. It is expected that the compiler will place such scalars in registers. These explicit registers are distinguished from *implicit* registers that are used internally by NVCC, to hold elements of arrays and intermediate results in computing expressions. Note that these terms are used just for the purpose of explanation; in the final code, the explicit registers just appear as thread scalars.

Listing 3.3 shows two time steps of an illustrative 3d 7-point Jacobi stencil defined in the StencilGen language. From rule (a), the statement at line 7 of Listing 3.3 should use a shared memory buffer to store the input plane $A[0, *, *]$. This is because five different values are read from that plane and contribute to different output points.

Listing 3.3: Two time steps of an illustrative 7-point 3D Jacobi stencil. The stencil is written using associativity of + to scatter contributions from an input plane to different output points

```
parameter L,M,N;
iterator k,j,i;
float A[L,M,N], B[L,M,N], C[L,M,N];
...
stencil j3d7pt (A, B, C) {
  B[k+1][j][i] =  (A[k][j][i]);
  B[k][j][i]   += (A[k][j-1][i] + A[k][j][i-1] + A[k][j][i]
                    + A[k][j][i+1] + A[k][j+1][i]);
  B[k-1][j][i] += (A[k][j][i]);

  C[k][j][i]   =  (B[k-1][j][i]);
  C[k-1][j][i] += (B[k-1][j-1][i] + B[k-1][j][i-1] +
                  B[k-1][j][i] + B[k-1][j][i+1] + B[k-1][j+1][i]);
  C[k-2][j][i] += (B[k-1][j][i]);
}
...
```

From rule (b), it follows that three explicit registers must be used to store the output values written to $B[1,*,*]$, $B[0,*,*]$ and $B[-1,*,*]$ at lines 2, 3, and 5 respectively.

The resource requirement for each stencil operator can be represented by a 3-tuple $(N_{reg}, N_{shm}, M_{acc \to res})$, where N_{reg} and N_{shm} are the number of explicit registers and shared memory buffers used by the operator, and $M_{acc \to res}$ is a resource map from each accessed data plane to a handle that represents the storage used for that plane.

3.5.3 Identifying Fusion Candidates

The dependences in the stencil DAG are captured by the dependence graph G_{dep} $= (V, E_{dep})$. From G_{dep}, consider a transitive dependence graph $G_{trans} = (V, E_{trans})$, such that

$$\begin{aligned} s_i \rightarrow s_j \in E_{trans} \text{ iff } \{ \; & \exists s_k \mid s_k \in V \wedge s_k \neq s_i \wedge s_k \neq s_j \wedge \\ & \text{there exists a path from } s_i \text{ to } s_k \text{ (denoted as } s_i \rightsquigarrow s_k \text{) } \wedge \\ & \text{there exists a path from } s_k \text{ to } s_j \text{ i.e., } s_k \rightsquigarrow s_j \; \} \end{aligned}$$

G_{trans} is used to ensure the validity of fusion, i.e., that all macro-nodes generated by fusion are convex. If $s_i \rightarrow s_j \in E_{trans}$, then fusing s_i and s_j will violate convexity unless all nodes along any path $s_i \rightsquigarrow s_j$ are also included in the fused macro-node. The definition above does not preclude fusion of a stencil operator with its immediate predecessor as long as there are no intervening operators along all paths from s_i to s_j.

For each distinct stencil operator pair (s_i, s_j) such that $s_i \rightarrow s_j \notin E_{trans}$, the approach first computes the total GPU resources required should the two stencil operators be fused, and then constructs a *fusion profitability* metric that will quantify the benefit of fusing s_i and s_j into a macro-node, in terms of the GPU resource consumption, and data movement reduction post fusion.

3.5.4 Computing the Post-Fusion Resource Map

To assess the impact of fusing two stencil operators, it is necessary to find the resources that would be needed to execute the fused node. Let M_i and M_j be the maps from accessed array planes to GPU resources ($M_{acc \rightarrow res}$) for the fusion candidates s_i and s_j, respectively. Let M_{fused} be the resource map for the fused node. If there is no dependence between s_i and s_j, then M_{fused} is simply a union of the resource map of the individual nodes, where the rules of union are as defined in this section.

If there is a true dependence between the two stencil operators, it is necessary to first create an interleaving of the instances of those operators such that the dependence between them is preserved post fusion. Once such a schedule is created, the resource map for the target of the dependence may need to be suitably adjusted by the dependence distance along the streaming dimension. For example, in Listing 3.3, after the contribution at line 5 the values of plane $B[-1, *, *]$ can be used as input for the next time step (lines 7–10). There is a true dependence from the write to plane $B[-1, *, *]$ at line 5 to the subsequent reads at lines 7–10. Notice that the accesses along the streaming dimension are shifted by the dependence distance in lines 7–10 to ensure that after fusing lines 2–10, dependences are preserved.

Once the schedules are adjusted to preserve dependences post fusion, there might be a non-empty intersection between the planes accessed by stencil operators s_i and s_j. For example, in Listing 3.3, $B[-1, *, *]$ is mapped to a register in line 5, and to shared memory in lines 8–9. In the presence of such conflicts for a plane, its storage in the fused node is always demoted to the lower memory in the hierarchy, i.e, shared memory in this case.

3.5.5 Computing the Profitability Metric

Once the resource map for the fused node has been computed, it is possible to quantify the profitability of fusing operators s_i and s_j using a 5-tuple $(D_m, T_{reg}, T_{shm}, S_{reg}, S_{shm})$, where

– D_m represents the savings in data movement after fusing s_i and s_j. For every true dependence between s_i and s_j, there is an implicit reduction in data movement for the array carrying the dependence. To capture this, D_m is incremented by 2, indicating a saving in a store and subsequent load of the array. There is no explicit data movement saving for arrays carrying WAR dependence. If there is WAW (RAR) dependence between s_i and s_j but no RAW dependence, D_m is incremented by 1 to capture the reduction in multiple writes (reads) to (from) the same location in global memory.

– T_{reg} represents the total number of explicit registers in the fused node

– T_{shm} represents the total amount of shared memory used in the fused node

– S_{reg} represents the total number of accesses that are mapped to the same explicit registers in s_i and s_j post fusion. $S_{reg} = \mathit{num\_registers}(M_i) + \mathit{num\_registers}(M_j) - \mathit{num\_registers}(M_{fused})$.

– S_{shm} represents the total number of accesses that are mapped to the same shared memory buffer in s_i and s_j post fusion. $S_{shm} = \mathit{num\_shared}(M_i) + \mathit{num\_shared}(M_j) - \mathit{num\_shared}(M_{fused})$.

3.5.6 Constructing the Objective Function

Once the 5-tuple has been computed for all stencil operator pairs, they should be ordered based on their fusion profitability. This can be done by defining a custom-sort relation $\prec$ as a total order over the list of 5-tuples (L_{tuple}). The set of sorting rules that can be applied to order two 5-tuples $c_i, c_j \in L_{tuple}$ is as follows:

a. $(D_m)_{c_i} < (D_m)_{c_j} \Rightarrow c_i \prec c_j$

b. $(T_{reg} + T_{shm})_{c_j} < (T_{reg} + T_{shm})_{c_i} \Rightarrow c_i \prec c_j$

c. $(T_{shm})_{c_j} < (T_{shm})_{c_i} \Rightarrow c_i \prec c_j$

d. $(T_{reg})_{c_j} < (T_{reg})_{c_i} \Rightarrow c_i \prec c_j$

e. $(S_{reg} + S_{shm})_{c_i} < (S_{reg} + S_{shm})_{c_j} \Rightarrow c_i \prec c_j$

f. $(S_{shm})_{c_i} < (S_{shm})_{c_j} \Rightarrow c_i \prec c_j$

g. $(S_{reg})_{c_i} < (S_{reg})_{c_j} \Rightarrow c_i \prec c_j$

h. $i < j \Rightarrow c_i \prec c_j$

where rule h is only used to ensure that a total order can be found in case of tuples with strictly identical metrics.

The order in which the sorting rules are applied to break a tie depends on the minimization objective. For example, if the objective is to minimize the data movement, $\prec$ follows the rule sequence $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h$, where a is the primary sorting rule, and the ties are settled by following the remaining rules in the sequence. Algorithm 3.3 describes the process of creating the 5-tuple for each stencil operator pair, and then sorting them based on the chosen objective (data movement minimization in the algorithm).

Once sorted, the topmost tuple of L_{tuple} represents the most profitable candidate for fusion. If the fusion (1) does not exceed the hardware limit on shared memory, and (2) does not result in a prohibitively large halo region (and consequently volume of redundant computations) in case of a true dependence between s_i and s_j, then the nodes corresponding to it are fused. The stencil DAG is updated to include the fused node, and the original operators are removed after updating the dependence graph appropriately. The whole process is repeated for this new DAG, till no further fusion is feasible.

Algorithm 3.3: Create-Sorted-Tuple-List (D): From the input DAG D, create the 5-tuple for each fusion candidate, and sort them based on the objective function

```
   Input  : IN: input DAG $D$ with $n$ statements
   Output: OUT: A sorted list of 5-tuple, $L_{tuple}$
1  $L_{tuple} \leftarrow \varnothing$;
2  compute_resource_3-tuple ($D$);
3  $\{G_{raw}, G_{war}, G_{waw}\} \leftarrow$ create_dependence_graphs ($D$);
4  $G_{dep} \leftarrow G_{raw} \cup G_{war} \cup G_{waw}$;
5  $G_{trans} \leftarrow$ transitive_closure ($G_{dep}$) $\backslash G_{dep}$;
6  for each distinct stmt pair $(i, j)$, $i, j \leq n$ do
7      $D_m \leftarrow 0$;
8      if ($s_i \rightarrow s_j \notin G_{trans}$) then
9          $M_i \leftarrow$ resource_usage ($s_i$);
10         $M_j \leftarrow$ resource_usage ($s_j$);
11         $M_{fused} \leftarrow$ fused_resource_map ($s_i, s_j, M_i, M_j$);
12         $D_m$ += 2*RAW_dependence_count ($s_i, s_j$);
13         $D_m$ += WAW_dependence_count ($s_i, s_j$);
14         $D_m$ += RAR_dependence_count ($s_i, s_j$);
15         $S_{reg} \leftarrow$ num_registers ($M_i$) + num_registers ($M_j$) - num_registers
            ($M_{fused}$);
16         $S_{shm} \leftarrow$ num_shared ($M_i$) + num_shared ($M_j$) - num_shared ($M_{fused}$);
17         $\{T_{shm}, T_{reg}\} \leftarrow$ compute_total_resources ($M_i, M_j$);
18         $L_{tuple}$.append (make_tuple ($D_m, T_{reg}, T_{shm}, S_{reg}, S_{shm}$));
19 sort ($L_{tuple}$, minimize_data_movement);
20 return $L_{tuple}$;
```

3.5.7 Imposing Hardware Constraints

In this section, we describe how the fusion algorithm accounts for the constraints of shared memory available per SM on the device (K_{max}), as well as the number of registers available for each thread (R_{max}).

The amount of shared memory used by each node can be estimated from the resource map ($M_{acc \rightarrow res}$) associated with each node, the size of the thread block used, and the data type of each buffer mapped to shared memory. The number of registers

Scheme	Load Inst.	Load Transactions	Store Inst.	Store Transactions
Traditional 3D Tiling	B^3	$B^2\left\lceil\frac{B}{W}\right\rceil$	$(B-2Tk)^3$	$(B-2Tk)^2\left\lceil\frac{B-2Tk}{W}\right\rceil$
Sliding Window with 2D Tiling	NB^2	$NB\left\lceil\frac{B}{W}\right\rceil$	$(N-2Tk)$ $(B-2Tk)^2$	$(N-2Tk)(B-2Tk)\left\lceil\frac{B-2Tk}{W}\right\rceil$

Scheme	Total Computation	Redundant computation
Traditional 3D Tiling	$\sum_{i=1}^{T}(B-2ik)^3$	$\sum_{i=1}^{T-1}((B-2ik)^3-(B-2Tk)^3)$
Sliding Window with 2D Tiling	$\sum_{i=1}^{T}(N-2ik)(B-2ik)^2$	$\sum_{i=1}^{T-1}((N-2ik)(B-2ik)^2 \ -(N-2Tk)(B-2Tk)^2)$

Table 3.1: Equations for per-block data transfers and computational volume for different overlapped tiling schemes. W is the warp size.

per thread is usually supplied at compile time in GPUs, and we assume that the limit is known to the fusion algorithm. Given these limits, the tuples in the list where the fused node would require (a) more than R_{max} registers during compliation, and (b) more than (K_{max}/N_B) bytes of shared memory per block, are ignored (N_B indicates the number of active blocks per SM). The amount of register spills can be determined by using *"-Xptxas -v"* flag during compilation. Setting the threshold of register spills to 0 might seem too conservative. However, modeling the relationship between spills, occupancy, and performance is currently beyond the scope of our framework. We allow the user to set the threshold on number of spills that can be tolerated.

3.5.8 Modeling Computation and Communication Redundancy

Table 3.1 lists the per-block data transfer and the computation volume for traditional 3D tiling, as well as the streaming strategy, assuming that all global memory transactions are perfectly aligned. Multiplying them by the number of blocks launched gives us a rough estimate of the total data transfers and computation volume. Based on these equations, we stop fusing nodes when the volume of recomputation in the fused node starts to dominate the data transfer volume. For illustration, assume that there are four fusion-amenable stencil operators. We can explore several fusion choices – fuse all the four to get a single operator, fuse two to get two resultant operators, or fuse none. For sliding-window tiling, Table 3.2 explores the profitability of different fusion choices for stencils of various orders.

k	fusion degree	Load Transactions	Store Transactions	Redundant Computation
1	1	18939904	17686800	0
	2	10786022	9364037	39978854
	4	7225344	5334336	144831456
2	1	21572044	18728073	0
	2	14450688	10668672	95227776
	4	15745024	7626496	487849728
4	1	28901376	21337344	0
	2	31490048	15252992	313916416

Table 3.2: Choosing degree of fusion for varying k

We seek to minimize global memory transactions via fusion, while accounting for the increase in redundant computation as more statements are fused together. The hardware constraints used to eliminate tuples of the sorted list L_{tuple} in Section 3.5.7 usually eliminate high degrees of fusion. The profitability of fusion for the configurations in Table 3.2 is determined by computing the estimated time for performing global memory transactions (T_{mem}) and the estimated time for performing the computation (T_{comp}). To compute these estimates, we use the peak memory bandwidth and the peak performance achieved by a set of synthetic stencil benchmarks on the

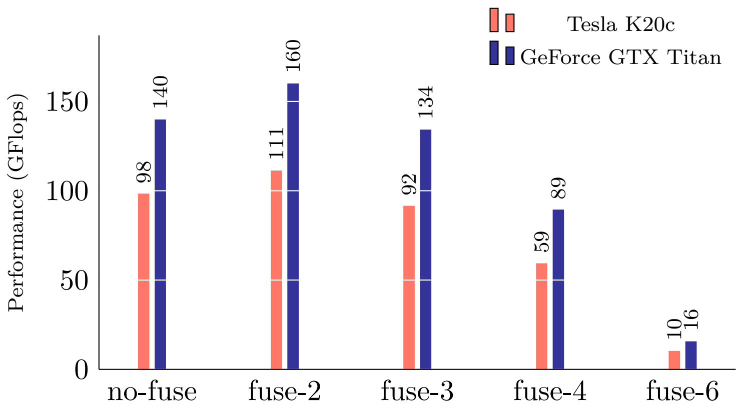


Figure 3.7: Performance of 12 time steps of a 13-point Jacobi stencil (k=2) for varied degree of fusion

underlying device. The comparison metric $T_{max} = max\ (T_{comp}, T_{mem})$, inspired by the roofline model [121], is used to quantify the efficiency of the generated sliding-window overlap-tiled code. If a tuple of the list L_{tuple} has a value of T_{max} for the fused node that is greater than the sum of the values of T_{max} for the unfused nodes, that fusion choice is eliminated from consideration. The configurations with lowest T_{max} for different values of k are highlighted in Table 3.2. For experimental validation, we plot the performance for 12 time steps of a 13-point order-2 3D Jacobi stencil sequence with varying degree of fusion. The results are shown in Figure 3.7. The best performance is achieved when the degree of fusion is 2, as predicted by the model.

3.5.9 Controlling recompilation

Since we need to compile the code generated for each fusion candidate to determine register usage/spills, excessive recompilation can add significant overhead to the code generation time for complex stencils. However, this overhead is mitigated in part due to the choice of a greedy algorithm over one that explores all valid schedules. The number of recompilations is controlled by the following choices:

- Fusion always results in an increase in the resources used by the fused cluster. We only recompile the code of the fused cluster when the fusibility constraints are not already violated by the constraints on shared memory, explicit registers, and redundant computation.
- If fusing any two nodes results in register spills, then we rule out, without recompilation, fusion of any clusters involving either of these two nodes. For example, if fusing nodes A and B results in spills, then we will not consider fusing clusters such as $\{A, C\}$, $\{B, D\}$, etc.
- We provide the user an option to specify a recompilation-bypass size, and only recompile for clusters exceeding that size. For example, if the recompilation-bypass size is 2, then we only recompile when the fusion results in a cluster of size ≥ 3.

With these choices, we prune away a large number of fusion candidates requiring recompilation, thus controlling the total code generation time.

3.6 Case Study: Hypterm

We evaluate the impact of applying the fusion heuristics on the hypterm routine of the ExpCNS mini-application from DoE that integrates the Compressible Navier-Stokes (CNS) equations [29]. For the discussion, we set the fusion objective to be minimization of data movement, and assume that the compilation restrics the registers per thread to 32. The hypterm routine reads from a set of input arrays (*cons* and *q*) to update a set of *flux* arrays. In the stencil DAG of hypterm shown in Figure 3.8,

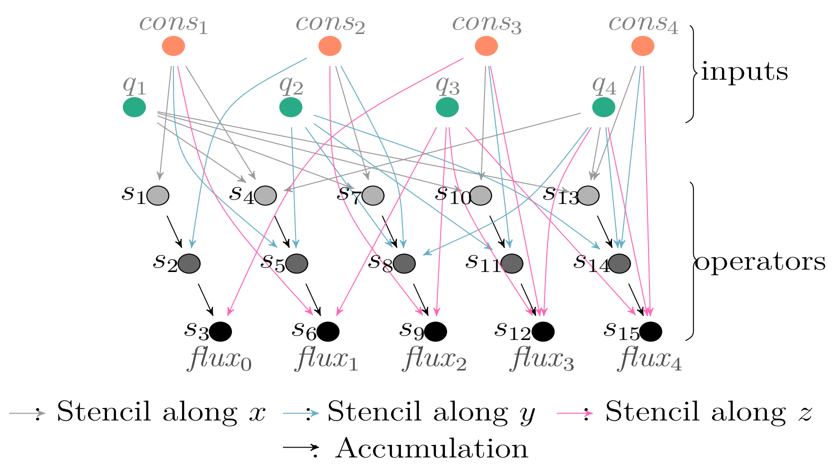


Figure 3.8: The stencil DAG of hypterm routine. *cons* and q are the input arrays, *flux* is the output array.

the intermediate and output nodes (in shades of black) are order-4 stencil operators. There are 15 accumulation statements in the stencil computation, which are labeled in the figure. Different colored edges in Figure 3.8 represent 1D stencils along different dimensions. The hypterm routine is not time-iterated, and thus efficient data reuse across multiple stencils is crucial for performance. An untiled implementation of hypterm which does not use shared memory, and where each stencil operator is in a separate kernel, achieves only 35.65 GFlops on a Tesla K20c device.

The fusion approach is applied to this DAG as follows:

– We begin by noting that s_{12} and s_{15} (s_6 and s_9) apply an order-4 1D stencil along z to three (two) input arrays. The number of registers used for spatial tiling would be ≥ 24 (≥ 16), which exceeds M_{reg}. Therefore, these statements with never be fused with any other statements.

– Statements s_{13} and s_{14} accumulate data to the same array, and have two common input arrays. Since their fusion would result in maximum data movement saving among all nodes, the fusion algorithm merges them.

– The next candidates for fusion are clusters $\{s_{10}$,$s_{11}\}$, and $\{s_4$,$s_5\}$. Both clusters result in a data movement saving of 2. However, the algorithm chooses to fuse s_{10} and s_{11} first, since the resulting node requires fewer total shared memory buffers. Statements s_4 and s_5 are fused in the next step.

– Fusing clusters $\{s_4$,$s_5\}$ and $\{s_{13}$,$s_{14}\}$ will result in a data movement saving of 3. However, the fusion results in register spills in the generated code, violating the fusibility constraints. Hence this cluster is not fused.

– Statements s_7 and s_8 have a common input, and contribute to the same output. Hence they are fused next.

– Clusters $\{s_4$,$s_5\}$ and $\{s_7$,$s_8\}$ are fused next, since fusing them saves on data movement without register spills. Statements s_1 and s_2 are fused as they contribute to the same array.

– Statement s_3 and cluster $\{s_{10}, s_{11}\}$ are fused next, since they have a common input. Cluster s$\{s_1$,$s_2\}$ and $\{s_{13}$,$s_{14}\}$ are fused to minimize the number of kernel launches.

Any further fusion violates the fusibility constraints, since the shared memory usage of the fused cluster would exceed M_{shm}. The final nodes in the optimized DAG are: $\{s_1, s_2, s_{13}, s_{14}\}$, $\{s_3, s_{10}, s_{11}\}$, $\{s_4, s_5, s_7, s_8\}$, s_6, s_9, s_{12}, and s_{14}. As mentioned earlier, statements s_6, s_9, s_{12}, and s_{14} apply an order-4 stencil along the streaming dimension z on two or more input arrays.

Due to the high register pressure with our tiling scheme, we opt for naïve tiling with no use of shared memory or explicit registers for s_{13} and s_{14} in the generated code. The optimized code for the post-fusion DAG achieves 128.98 GFlops on a K20c, a 3.6× speedup over the base unfused version.

Recompilations If we set the recompilation-bypass size to 1, then we perform nine recompilations to detect any violation of the fusibility constraints – five for clusters of size 2, two for clusters of size 3, and two for clusters of size 4. If the recompilation-bypass size is set to 2, then only four recompilations are needed.

Effect of implicit registers on degree of fusion Since all the statements in the hypterm stencil are accumulations, a fusion model that does not account for implicit registers will fuse too many statements together. This will create high register pressure in the generated kernel, leading to register spills. In Figure 3.9, we plot the performance of the generated code for various degrees of fusion. *max-fuse* refers to the version with maximum fusion. It requires a 264 byte stack frame, generating 648 bytes of spill stores and 488 bytes of spill loads. *fuse-8* and *fuse-6* also generate spills. *fuse-8* requires 72 bytes of stack frame (76 bytes spill stores, 92 bytes spill loads), and *fuse-6* requires 16 bytes of stack frame (20 bytes spill stores, 20 bytes spill load). There are no spills generated for *fuse-4* and *fuse-2*. Any advantages from reduction in data movement for a degree of fusion higher than 4 is offset by the increase in global memory traffic from register spills. Since our fusion approach checks for register spills in the fusibility constraints, we generate a code with degree of fusion 4, thereby achieving the highest performance amongst these variants.

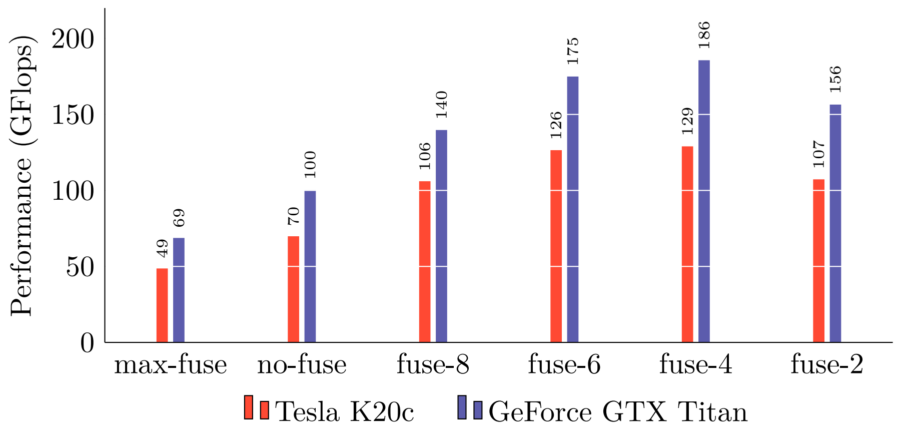


Figure 3.9: Performance of hypterm for varied degree of fusion

3.7 Experimental Evaluation

3.7.1 Evaluation on GPUs

Experimental setup We have implemented the tiling schemes and fusion heuristics described in Sections 3.4 and 3.5 into the StencilGen code generator, and we compare below the performance of the StencilGen-generated code against PPCG-0.07 [113], OpenACC-17.4 [78], and the auto-scheduler branch of Halide [70]. The benchmarks used in evaluation are listed in Table 5.3. All the benchmarks are single precision, and the results GPU are obtained on Tesla K20c and Pascal Titan X devices; the hardware is detailed in Table 5.1. For PPCG and StencilGen, the generated code was compiled using NVCC 8.0 [75]. The Halide generated code was compiled with LLVM 3.7 [57]. The compilation flags are listed in Table 3.5.

Code generation PPCG performs classical time tiling along with default thread coarsening. Mapping multiple iterations to a thread exposes instruction level parallelism. Coarsening within the sustainable per thread register pressure aids register

Benchmark	N	T	k	FPP
j2d5pt	8192^2	4	1	10
j2d9pt-gol	8192^2	4	1	18
j2d9pt	8192^2	4	2	18
gaussian	8192^2	4	2	50
gradient	8192^2	4	1	18
j3d7pt	512^3	4	1	13
j3d13pt	512^3	4	2	25
j3d17pt	512^3	4	1	28

Benchmark	N	T	k	FPP
j3d27pt	512^3	4	1	54
curl	450^3	1	1	36
heat	512^3	4	1	15
poisson	512^3	4	1	21
cheby	512^3	4	1	39
denoise	512^3	4	2	62
hypterm	300^3	1	4	358

N: Domain Size, T: Time Tile Size, k: Stencil Order, FPP: FLOPs per Point

Table 3.3: Characteristics of bandwidth-bound stencil benchmarks

Resource	**Details**
Tesla K20c GPU	13 MPs, 192 cores/MP, 208 GB/s peak global memory bandwidth, 3.52 TFLOPS peak single-precision performance, 5 GB global memory, 48K shared memory, 64K registers/SM
Pascal Titan X GPU	28 MPs, 128 cores/MP, 480 GB/s peak global memory bandwidth, 10.9 TFLOPS peak single-precision performance, 12 GB global memory, 48K shared memory, 64K registers/SM
Intel Skylake CPU	Intel core i7-6700K (4 cores, 2 threads/core 4.00 GHz, 512 GFLOPS peak single-precision performance)
Intel Xeon CPU	Intel Xeon E5-2680 (14 cores, 2 threads/core 2.40 GHz, 1.05 TFLOPS peak single-precision performance)
KNL phi	Intel Xeon Phi 7250 (68 cores, 4 threads/core 1.40GHz, 6.092 TFLOPS peak single-precision performance)

Table 3.4: Benchmarking hardware for bandwidth-bound stencils

level reuse, and helps hide memory access latency by exposing instruction-level parallelism [115]. We tuned the tile sizes for PPCG, and report results for the version with best performance. StencilGen does not have the support for thread coarsening at present. For each benchmark, we leverage operator associativity to alleviate pressure on GPU resources. The fusion heuristic restricts the degree of fusion to two time steps for the time iterated benchmarks. K20c and Titan X GPU devices can load the

Compiler	**Flags**
nvcc	-O3 -maxrregcount=n -ccbin=g++ -std=c++11 -Xcompiler "-fPIC -fopenmp -O3 -fno-strict-aliasing" –use_fast_math -Xptxas "-v -dlcm=ca" -arch=sm_35/60
gcc	-Ofast -fopenmp -ffast-math -fstrict-aliasing -march=core-avx2/{knl -mavx512f -mavx512er -mavx512cd -mavx512pf} -ffp-contract=fast -mfma
llvm	-Ofast -fopenmp=libomp -mfma -ffast-math -fstrict-aliasing -march=core-avx2/knl -mllvm -ffp-contract=fast
icc	-Ofast -qopenmp -no-prec-div -march=native -fma -xMIC-AVX512 -fp-model fast=2 -complex-limited-range -override-limits

Flags for GPU, multi-core CPU, and Xeon Phi

Table 3.5: Compilation flags for different compliers and different benchmarking hardware

read-only data through the cache used by texture pipeline. To enable this feature, the read-only data in StencilGen-generated code is automatically annotated with the `__restrict__` keyword.

GPU performance results Figures 3.10 and 3.11 plot the performance of the StencilGen-generated code against different code generators on the two GPU devices. StencilGen systematically outperforms the other code generators in our experiments, by a factor up to 7×. Our optimization scheme for associative stencils helps achieve 868 GFLOPS (3.81 TFLOPS) for gaussian stencil, and 730 GFLOPS (2.94 TFLOPS) for j3d27pt stencil on K20c (Titan X) device. For the 2D stencils, the best performance is achieved by overlapped tiling along both x and y dimensions, and concurrent-streaming along y dimension. For the 3D stencils, overlapped tiling along x and y dimensions, and serial-streaming along z dimension gives best performance.

After careful analysis of this experimental data, we obtained the following conclusions. (1) One has to choose a combination of varying optimizations that overcome

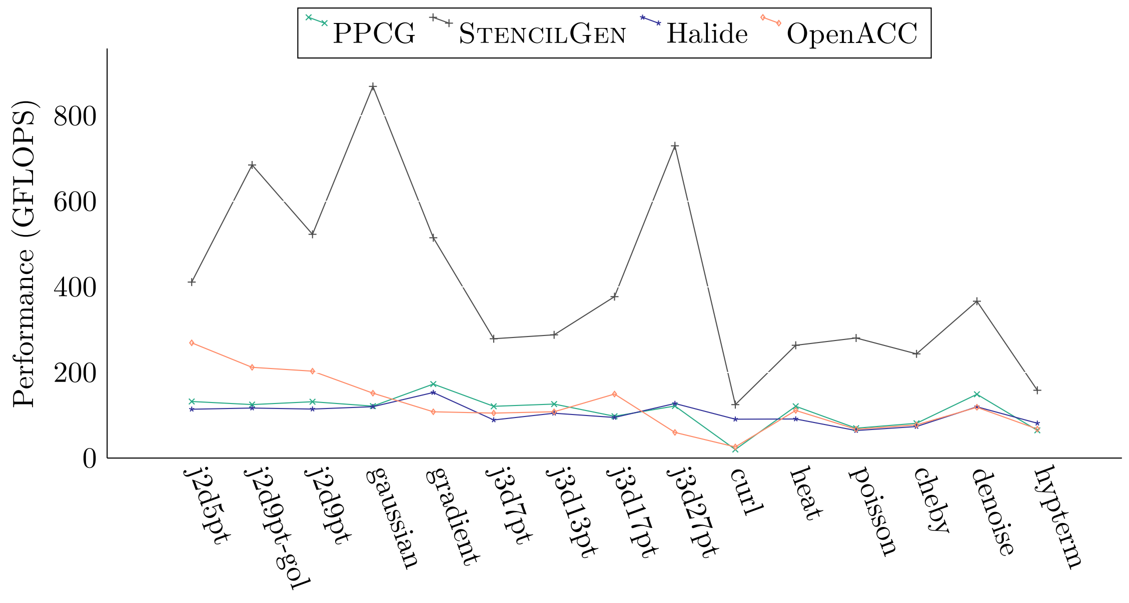


Figure 3.10: Performance of benchmarks on Tesla K20c device

the performance-limiting hardware constraints depending on the dimensionality of the problem (2) Using registers along with shared memory as buffers is crucial to performance irrespective of the dimensionality. (3) Serial-streaming for 3D stencils and concurrent-streaming for 2D stencils reduces the extra bandwidth consumption without constraining concurrency. (4) Fusion of time steps for time iterated stencils, and stencil operators for a stencil DAG reduces the global memory transactions, which is crucial in optimizing bandwidth-bound stencils.

Table 3.6 lists the percentage of the peak single-precision FLOPs achieved by the code generated by different code generators on the two GPU devices. Despite applying a plethora of optimizations, none of the codes achieve more than 25% (35%) of the peak performance on K20c (Titan X). This serves to highlight the bandwidth-bound nature of stencil computations.

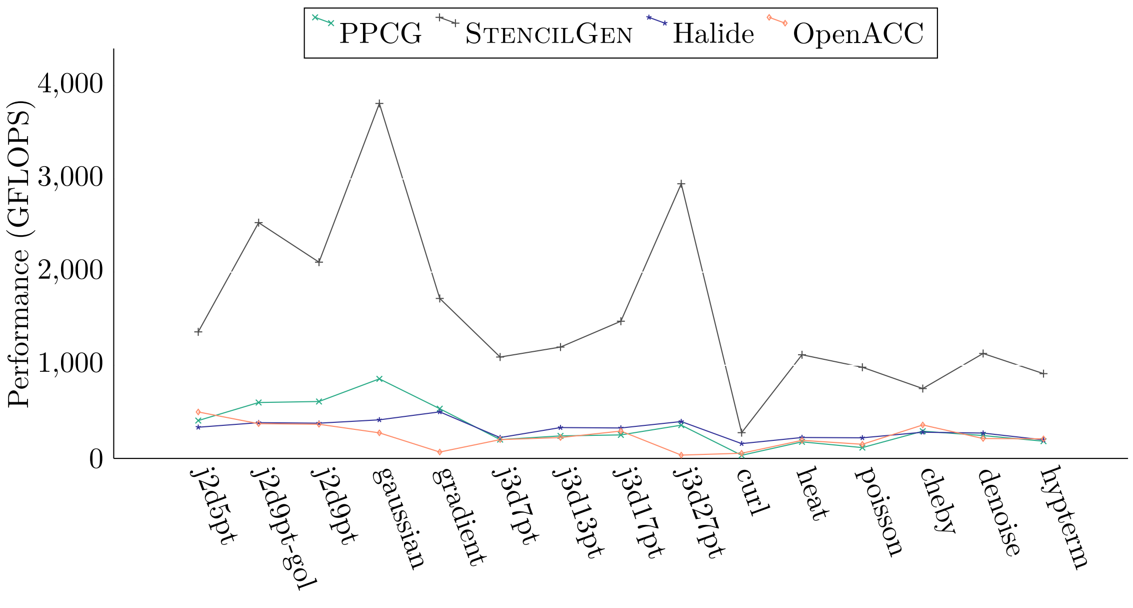


Figure 3.11: Performance of benchmarks on Pascal Titan X GPU device

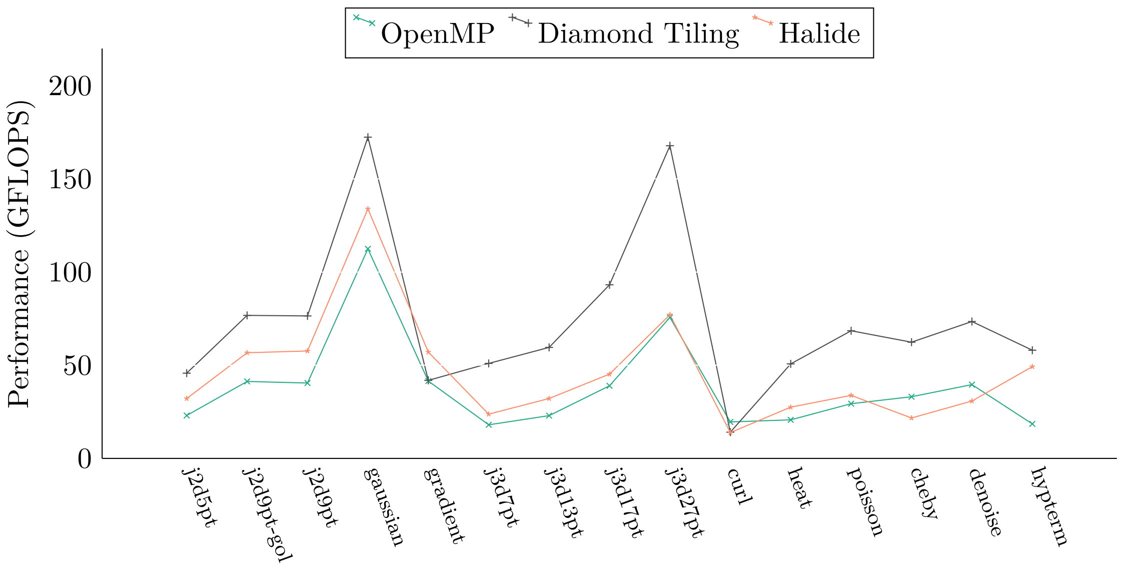


Figure 3.12: Performance of benchmarks on Skylake processor

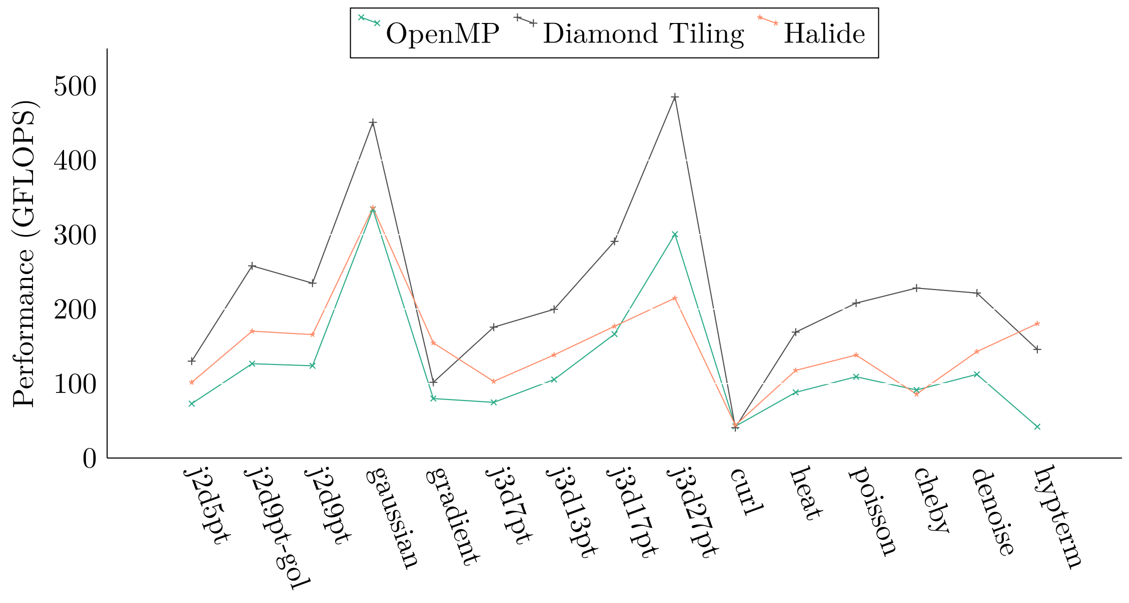


Figure 3.13: Performance of benchmarks on Xeon processor

3.7.2 Stencil Computations on CPUs

Experimental setup Many frameworks explicitly target stencil optimizations on CPU [7, 84, 125, 18, 42]. We evaluate the CPU code generated by diamond tiling [7] and halide with auto-scheduling support [70] against a baseline OpenMP code on two different multi-core CPU processors, and a Xeon Phi processor. We also evaluate Intel's latest stencil optimization framework, YASK [125], on the Xeon Phi processor. Due to the limitations of the framework, we could only generate YASK code for 3D stencils. We use three different compilers, namely ICC-17.0 [46], LLVM-5.0 [57], and GCC-7.2 [98] to compile the generated C/C++ stencil code, and report the highest performance.

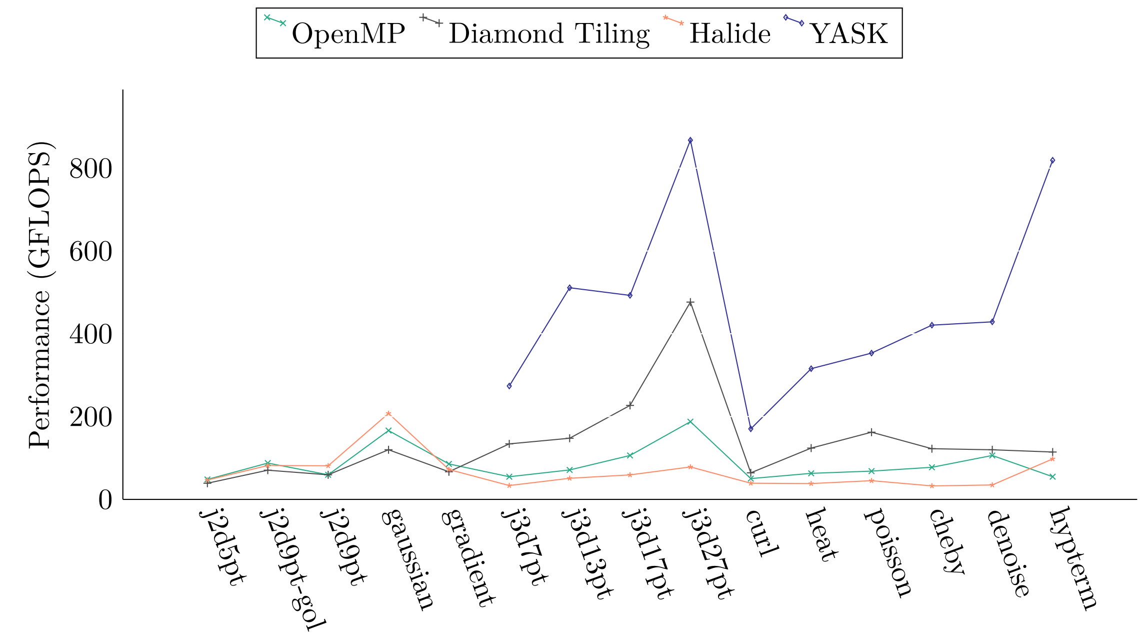


Figure 3.14: Performance of benchmarks on Xeon Phi 7250

CPU performance results The performance is plotted in Figures 3.12, 3.13 and 3.14. One can observe that diamond tiling outperforms both Halide and the baseline code with its time tiling and concurrent start optimizations. On Xeon Phi, the performance of YASK is far superior to that of other frameworks. YASK performs optimizations like vector folding, cache blocking, and temporal wave-front tiling, that are specifically tuned for the Xeon Phi architecture.

For each benchmark, Tables 3.7 and 3.8 report the compiler that was used in compilation of the best-performing version, the percentage of the machine peak achieved by that version. For the multi-core CPUs, only diamond tiling is able to achieve nearly 30%-45% of the machine peak for j3d27pt stencil. Despite the high performance, YASK is able to achieve only 15% of the machine peak.

3.7.3 Energy Efficiency

Tables 3.6, 3.7, and 3.8 also tabulate the energy expended in a single floating-point operation for GPU, multi-core CPUs, and Xeon Phi respectively. For the stencil codes compiled with NVCC, we use the NVidia Management Library (NVML[2]) to measure the Joules/FLOP ratio. For multi-core CPUs and Xeon Phi, we use RAPL[3] to obtain the measurement. Although typically measurements indicate that faster code versions have a lower Joules per FLOP ratio, data shows situations where a faster code (higher PP) has a lower energy efficiency (JPF) as when comparing Halide and OpenMP versions of j2d9pt-gol. We conjecture the use of more power-hungry instructions (e.g., FMA) and additional in-processor data traffic as possible causes for this slight decrease in energy efficiency, as unfortunately obtaining fine-grain power measurements to determine the root cause is not feasible with our energy measurement setup.

Note these benchmarks running on GPU devices have a much lower ($1\times - 0.1\times$) Joules/FLOP ratio that the multi-core CPUs and Xeon Phi, while also achieving slightly higher peak performance. It indicates that the current generation of GPU devices appear more energy efficient than these CPU processors in executing such bandwidth-bound stencil computations.

3.8 Summary

Several factors are important in enhancing the performance of a 3D stencil computation on a GPU: (a) maximizing use of the processing resources; (b) minimizing

[2]`https://developer.nvidia.com/nvidia-management-library-nvml`

[3]`http://web.eece.maine.edu/~vweaver/projects/rapl/`

global memory accesses; (c) effective use of shared memory and registers; (d) accounting for the computational redundancy introduced by the tiling scheme. In this chapter, we described algorithms to generate optimized code for stencil computations by managing the allocation of GPU resources, using fusion along with 2D overlapped tiling, and streaming through one unpartitioned spatial dimension. We described a greedy fusion heuristic seeks to optimize the use of shared memory and registers. Experimental results on two GPU devices on a set of resource-intensive benchmarks demonstrated significant performance improvement over other available code generators.

Bench.	Tesla K20c						Pascal Titan X					
	PPCG		StencilGen		Halide	OpenACC	PPCG		StencilGen		Halide	OpenACC
	PP	JPF	PP	JPF	PP	PP	PP	JPF	PP	JPF	PP	PP
j2d5pt	3.75	6.17E-10	11.66	2.60E-10	3.23	7.6	3.70	3.28E-10	12.37	1.37E-10	3.05	4.54
j2d9pt-gol	3.54	5.47E-10	19.43	1.57E-10	3.31	6.01	5.46	2.41E-10	23.06	7.86E-11	3.49	3.39
j2d9pt	3.73	5.20E-10	14.83	2.10E-10	3.24	5.76	5.56	2.40E-10	19.19	8.47E-11	3.44	3.34
gaussian	3.44	4.58E-10	24.64	1.34E-10	3.40	4.30	7.78	1.90E-10	34.71	5.17E-11	3.77	2.50
gradient	4.90	4.94E-10	14.60	2.24E-10	4.34	3.06	4.86	2.71E-10	15.64	1.16E-10	4.55	0.62
j3d7pt	3.43	6.67E-10	7.89	3.56E-10	2.52	2.98	1.84	5.18E-10	9.91	1.78E-10	2.03	1.83
j3d13pt	3.57	5.68E-10	8.17	2.71E-10	2.97	3.06	2.20	3.98E-10	10.89	1.58E-10	3.01	2.04
j3d17pt	2.77	7.08E-10	10.70	2.34E-10	2.69	4.24	2.29	4.08E-10	13.42	1.29E-10	2.97	2.67
j3d27pt	3.45	4.02E-10	20.70	1.29E-10	3.61	1.69	3.25	2.27E-10	26.86	5.34E-11	3.60	0.32
curl	0.56	2.69E-09	3.54	6.95E-10	2.57	0.73	0.30	2.33E-09	2.52	4.96E-10	1.44	0.50
heat	3.43	5.27E-10	7.47	3.69E-10	2.59	3.15	1.62	4.35E-10	10.13	1.67E-10	2.04	1.76
poisson	1.98	7.74E-10	7.96	3.24E-10	1.82	1.91	1.05	6.03E-10	8.91	2.04E-10	2.01	1.38
cheby	2.30	8.83E-10	6.90	3.61E-10	2.09	2.20	2.66	4.14E-10	6.83	1.63E-10	2.56	3.27
denoise	4.22	5.58E-10	10.39	2.74E-10	3.39	3.36	2.23	3.85E-10	10.26	1.38E-10	2.48	1.94
hypterm	1.83	1.06E-09	4.49	4.24E-10	2.31	1.92	1.68	8.01E-10	8.29	1.73E-10	1.81	1.91

PP:Percentage of peak GFLOPS, JPF: Joules per FLOP

Table 3.6: Energy and performance characteristics of benchmarks on GPU devices

Bench.	i7-6700K									Xeon E5-2680								
	OpenMP			Diamond Tile			Halide			OpenMP			Diamond Tile			Halide		
	C	PP	JPF	C	PP	JPF	C	PP	JPF	C	PP	JPF	C	PP	JPF	C	PP	JPF
j2d5pt	gcc	4.49	1.6E-09	icc	8.93	1.0E-09	icc	6.27	4.5E-09	icc	6.79	3.9E-09	icc	12.1	4.8E-09	llvm	9.45	1.9E-08
j2d9pt-gol	icc	8.06	9.8E-10	icc	14.9	8.0E-10	icc	11.1	2.6E-09	llvm	11.8	2.8E-09	icc	24.0	1.3E-08	gcc	15.8	1.1E-08
j2d9pt	icc	7.90	1.0E-09	icc	14.9	7.0E-10	gcc	11.3	2.6E-09	icc	11.5	2.1E-09	icc	21.8	1.1E-08	icc	15.4	1.1E-08
gaussian	gcc	21.9	4.8E-10	icc	33.7	3.4E-10	icc	26.1	1.1E-09	icc	31.1	9.3E-10	icc	42.0	4.4E-09	llvm	31.3	4.8E-09
gradient	llvm	8.10	1.1E-09	icc	8.17	1.2E-08	icc	11.1	2.6E-09	llvm	7.42	3.8E-09	icc	9.45	1.5E-08	icc	14.4	1.1E-08
j3d7pt	gcc	3.52	2.1E-09	icc	9.95	8.0E-10	icc	4.63	3.3E-09	llvm	6.95	4.7E-09	gcc	16.3	3.4E-09	llvm	9.57	1.1E-08
j3d13pt	llvm	4.48	1.6E-09	icc	11.6	7.1E-10	gcc	6.28	2.2E-09	icc	9.80	3.5E-09	llvm	18.6	2.7E-09	icc	12.9	6.6E-09
j3d17pt	llvm	7.61	1.0E-09	icc	18.2	5.2E-10	gcc	8.83	1.9E-09	icc	15.5	2.2E-09	gcc	27.1	1.8E-09	llvm	16.4	6.3E-09
j3d27pt	llvm	14.7	6.0E-10	icc	32.8	3.5E-10	llvm	15.2	1.1E-09	llvm	27.9	1.3E-09	icc	45.1	3.7E-09	gcc	20.0	3.7E-09
curl	gcc	3.82	4.2E-09	icc	2.74	1.7E-09	icc	2.70	9.0E-09	icc	3.96	5.7E-09	llvm	3.8	1.6E-08	icc	4.1	3.6E-08
heat	llvm	4.04	1.7E-09	icc	9.90	8.2E-10	icc	5.37	2.9E-09	icc	8.20	4.1E-09	gcc	15.7	5.3E-09	gcc	10.9	9.3E-09
poisson	gcc	5.73	1.3E-09	icc	13.4	7.1E-10	llvm	6.60	2.5E-09	icc	10.1	2.8E-09	gcc	19.3	4.0E-09	gcc	12.9	7.9E-09
cheby	icc	6.46	9.1E-10	icc	12.2	6.1E-10	icc	4.23	2.7E-09	icc	8.50	3.6E-09	icc	21.2	7.0E-09	llvm	8.0	8.0E-09
denoise	llvm	7.73	9.4E-10	gcc	14.3	6.2E-10	icc	6.01	2.0E-09	icc	10.4	3.0E-09	icc	20.6	4.0E-09	gcc	13.3	5.4E-09
hypterm	icc	3.61	2.2E-09	icc	11.3	9.8E-10	icc	9.22	4.2E-09	llvm	3.91	6.6E-09	icc	13.6	6.6E-09	gcc	16.8	2.0E-08

C: Compiler with highest GFLOPS, PP: Percentage of peak GFLOPS, JPF: Joules per FLOP

Table 3.7: Energy and performance characteristics of benchmarks on multi-core CPU

Bench	OpenMP			Diamond Tile			Halide			YASK		
	C	PP	JPF	C	PP	JPF	C	PP	JPF	C	PP	JPF
j2d5pt	icc	0.79	4.15E-09	icc	0.64	5.77E-09	llvm	0.76	1.03E-07	-	-	-
j2d9pt-gol	icc	1.44	2.47E-09	icc	1.15	9.40E-08	icc	1.34	6.12E-08	-	-	-
j2d9pt	icc	0.97	3.26E-09	icc	0.97	9.36E-08	icc	1.33	6.43E-08	-	-	-
gaussian	llvm	2.72	1.16E-09	icc	1.96	1.78E-09	icc	3.40	2.72E-08	-	-	-
gradient	icc	1.40	2.27E-09	icc	1.09	9.22E-08	llvm	1.18	5.98E-08	-	-	-
j3d7pt	llvm	0.89	3.61E-09	icc	2.20	1.69E-09	icc	0.55	5.65E-08	icc	4.49	8.95E-10
j3d13pt	icc	1.16	3.18E-09	icc	2.42	1.48E-09	icc	0.83	3.41E-08	icc	8.39	5.32E-10
j3d17pt	icc	1.74	1.97E-09	icc	3.72	9.54E-10	llvm	0.97	3.15E-08	icc	8.08	5.95E-10
j3d27pt	icc	3.08	1.19E-09	icc	7.82	4.48E-10	llvm	1.28	1.88E-08	icc	14.23	3.31E-10
curl	icc	0.82	4.98E-09	icc	1.05	1.20E-08	llvm	0.64	1.42E-07	icc	2.79	8.31E-09
heat	icc	1.03	3.15E-09	icc	2.03	1.52E-09	icc	0.62	4.87E-08	icc	5.18	7.78E-10
poisson	icc	1.12	2.87E-09	icc	2.66	1.32E-09	llvm	0.74	3.97E-08	icc	5.79	8.64E-10
cheby	icc	1.27	2.22E-09	icc	2.00	1.79E-09	icc	0.53	3.31E-08	icc	6.90	6.85E-10
denoise	icc	1.74	2.06E-09	icc	1.96	2.08E-09	icc	0.57	2.35E-08	icc	7.03	5.39E-10
hypterm	llvm	0.90	3.27E-09	icc	1.87	3.57E-10	icc	1.60	9.70E-08	icc	13.44	6.89E-09

C: Compiler with highest GFLOPS, PP: Percentage of peak GFLOPS, JPF: Joules per FLOP

Table 3.8: Energy and performance characteristics of benchmarks on Xeon Phi

CHAPTER 4

Register Optimizations for Stencils on GPU

4.1 Introduction

The footprint of a stencil is determined by its *order*, which is the number of input elements from the center read along each dimension. Stencils with high (low) order have high (low) arithmetic intensity. In many scientific applications, the stencil order determines the computational accuracy. For this reason, high-order stencils have been gaining popularity. However, the inherent data reuse within or across statements in such high-order stencils exposes performance challenges that are not addressed by current stencil optimizers.

A significant focus in optimizing stencil computations has been to fuse operations across time steps or across a sequence of stencils in a pipeline [7, 36, 38, 71, 87, 116, 124]. With high-order stencils, the operational intensity is sufficiently high so that even with just a simple spatial tiling, the computation should theoretically not be memory-bandwidth bound. Consider a GPU with around 300 GBytes/sec global memory bandwidth and a peak double-precision performance of around 1.5 TFLOPS. The required operational intensity to be compute-bound and not memory-bandwidth bound is around 5 FLOPs/byte or 40 FLOPs per double-word. Many high-order stencil computations have much higher arithmetic intensities than 40. For such stencils, achieving a high degree of reuse in cache is very feasible, but high performance is not realized on GPUs. *The main hindrance to performance is the high register pressure with such codes, resulting in excessive register spilling and a subsequent loss of performance.* As we elaborate in the next section, existing register management techniques in production compilers are not well equipped to address the problem with register pressure for high-order stencils. Addressing this problem in context of GPUs is even more challenging, since most of the widely used GPU compilers like NVCC [75] are closed-source. Even the recent effort by Google (gpucc [122]) has only exposes the front-end to the user, and uses the NVCC backend as a black box to perform instruction scheduling and register allocation.

In this chapter, we develop an effective pattern-driven global optimization strategy for instruction reordering to address this problem. The key idea behind the instruction reordering approach is to model reuse in high-order stencil computations by using an abstraction of a DAG of trees with shared nodes/leaves, and exploit the fact that optimal scheduling to minimize registers for a single tree with distinct operands at the leaves is well known [93]. We thus devise a statement reordering strategy for a DAG of trees with shared nodes that enables reduction of register pressure to improve performance. The chapter makes the following contributions:

- It proposes a framework for multi-statement stencils that reduces register pressure by reordering instructions across statements.
- It describes novel heuristics to schedule a DAG of trees that reuse data using a minimal number of registers.

- It demonstrates the effectiveness of the proposed framework on a number of register-constrained stencil kernels.

4.2 Background and Motivation

Register Allocation and Instruction Scheduling A compiler has several optimization passes, register allocation and instruction scheduling being two of them. Passes before register allocation manipulate an intermediate representation with an unbounded number of *temporary* variables. The goal of register allocation is to assign those temporaries to physical storage locations, favoring the few but fast registers to the slower but larger memory.

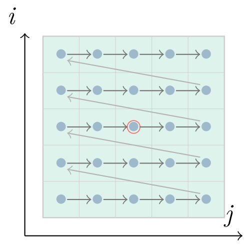
For a fixed schedule, a common approach to perform register allocation is to build an interference graph of the program, which captures the intersection of the live-ranges of temporaries at any program point. Register assignment is then reduced to coloring the interference graph, where each color represents a distinct register [15, 16]. Interfering nodes in the interference graph are assigned different colors due to their adjacency. The number of registers needed by the coloring algorithm is lower-bounded by the maximum number of intersecting live-ranges at any program point (MAXLIVE). If MAXLIVE is more than the number of physical registers, *spilling* of registers and the consequent load/store operations from/to memory are unavoidable.

Register pressure can sometimes be alleviated by reordering the schedule of dependent instructions to reduce the MAXLIVE. Reordering independent instructions is often used to enhance the amount of instruction-level parallelism (ILP), for hiding memory access latency. Thus, there is a complex interplay between instruction scheduling and register allocation, affecting instruction-level parallelism and register pressure, and the associated optimization problem is highly combinatorial. Production compilers generally use heuristics for increasing ILP, with a best-effort greedy control on register pressure. For typical application codes, the negative effect on register pressure is not very significant. However, for high-order stencil codes with a large number of operations and a lot of potential register-level reuse, the impact can be very high, as illustrated by an example below.

Illustrative Example Consider an unrolled version of the double-precision 2D Jacobi stencil computation (Figure 4.1a) from [99]. NVCC interleaves the contribution from each input point to different output points to increase instruction level parallelism (ILP). The interleaving performed to increase ILP also has the serendipitous effect of reducing the live range of the register data, and a consequent reduction in register pressure. Nvprof [76] profiling data on a Tesla K40c device shows that under maximum occupancy, this version performs 3.73E+06 spill transactions, achieving 467 GFLOPS.

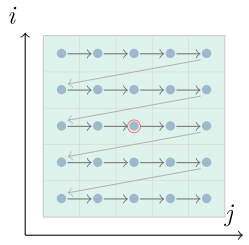
Figure 4.1b shows the same stencil computation after changing the order of accumulation. Exactly the same contributions are made to each result array element, but the order of the contributions has been reversed. With this access pattern for the code in Figure 4.1b, NVCC fails to perform the same interleaving despite allowing reassociation via appropriate compilation flags. In fact, the register pressure is now exacerbated by the consecutive scheduling of independent operations to increase ILP. For this version, 1.58E+08 spill transactions were measured, with performance dropping to 51 GFLOPS.

```
for (i=2; i<N-2; i++)
    for (j=2; j<N-2; j++) {
         out[i][j] = 0;
         for (ii=-2; ii<=2; ii++)
              for (jj=-2; jj<=2; jj++)
                   out[i][j] += in[i+ii][j+jj] * w[ii+2][jj+2];
    }
```



(a) Stencil with lexicographical sweeps

```
for (i=2; i<N-2; i++)
    for (j=2; j<N-2; j++) {
         out[i][j] = 0;
         for (ii=2; ii>=-2; ii--)
              for (jj=-2; jj<=2; jj++)
                   out[i][j] += in[i+ii][j+jj] * w[ii+2][jj+2];
    }
```



(b) Stencil with reverse-lexicographical sweeps

Figure 4.1: Comparing the performance of the same stencil computation with different sweeping order

This example illustrates a problem with register allocation when the computation has a specific reuse pattern, characteristic of high-order stencil computations. The problem stems from the fact that for most compliers the register allocation and instruction scheduling algorithms that operate at a basic-block level have a peephole view of the computation – they make scheduling/allocation decisions without a global perspective, and thus sometimes work antagonistically. Meanwhile, stencil computations typically have a very regular access pattern. With a better understanding of the pattern, and a global perspective on the computation, it is feasible to devise an instruction reordering strategy to alleviate register pressure.

Solution Approach In this chapter, we circumvent the complexity of the general optimization problem of instruction reordering and register allocation by devising a pattern-specific optimization strategy. Stencil computations involve accumulation of contributions from array data elements in a small neighborhood around each element. The additive contributions to a data element may be viewed as an expression tree. Thus, for multi-statement stencils, we have a DAG of expression trees. Due to the fact that an element may contribute to several result elements, the trees within the DAG can have many shared leaves.

Given a single tree without any shared leaves, it is well known [93] how to schedule its operations in order to minimize the number of registers needed. We use this as the basis for developing heuristics to schedule the operations from the DAG of trees with shared leaves. In contrast to the problem of reordering an arbitrary sequence of instructions to minimize register pressure, a structured approach of adapting the optimal schedule for isolated trees to the case of DAG of trees with shared leaves results in an efficient and effective algorithm that we develop in the next two sections.

4.3 Scheduling DAG of Expression Trees

Stencil computations are often succinctly represented using a domain-specific language (DSL). Listing 4.1 shows an illustrative 7-point Jacobi stencil expressed in the StencilGen DSL. The core computation is shown in lines 7–9. As with similar DSLs, the user can specify unroll factors for loop iterators (line 11). Loop unrolling, or thread coarsening on GPUs, is often used to exploit register-level reuse in the code. The computation is automatically unrolled as a preprocessing step, before the code is generated and optimized.

Listing 4.1: The input representation in the StencilGen DSL

```
1   parameter L, M, N;
2   iterator k, j, i;
3   double in[L][M][N], out[L][M][N], a, b, c;
4   copyin in;
5
6   function j3d7pt (out, in, a, b, c) {
7    out[k][j][i] = a*(in[k+1][j][i]) + b*(in[k][j-1][i] +
8            in[k][j][i-1] + in[k][j][i] + in[k][j][i+1] +
9                  in[k][j+1][i]) + c*(in[k-1][j][i]);
10  }
11  unroll k=2, j=2;
12  j3d7pt (out, in, a, b, c);
13  copyout out;
```

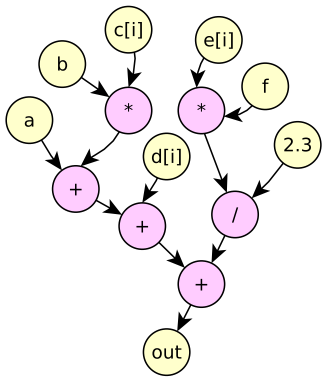
It is important to note that using a DSL is not a prerequisite for using the scheduling techniques proposed in this work. As described shortly, our approach works on a DAG of expression trees. This DAG can be automatically extracted either from the DSL representation or from C/Fortran code.

A stencil statement can be defined by the stencil shape (as in lines 7–9) and the input/output data (as in line 3). Each such stencil statement can be represented by a labeled expression tree. For example, the tree corresponding to the computation in Listing 4.1 has array element $out[k, j, i]$ as its root, scalars a, b, c and accesses to elements of array *in* as its leaves, and arithmetic operators $*$ and $+$ as inner nodes.

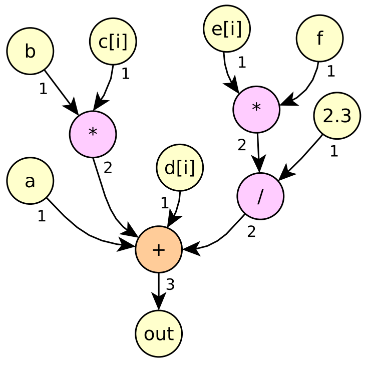
An expression tree for a stencil computation has three types of nodes: (1) nodes $n \in N_{mem}$ representing accesses to memory locations, (2) nodes $n \in N_{op}$ representing binary/unary arithmetic operators, and (3) leaf nodes representing constants. All leaf nodes in N_{mem} correspond to reads of array elements (e.g., $in[k+1, j, i]$) or scalars. The root of the expression tree is also in N_{mem} and corresponds to a write to an

```
out = a + (b * c[i]) + d[i] + ((e[i] * f) / 2.3);
```

(a) Illustrative stencil statement



(b) Expression tree



(c) Expression tree with accumulations

Figure 4.2: Expression tree example

array element (e.g., $out[k, j, i]$) or a scalar. We associate a unique label with each read/written memory location, and assign to each node in N_{mem} the corresponding label. The remaining tree nodes are in N_{op}. Figure 4.2b shows the expression tree for an illustrative expression.

In a preprocessing step, we introduce k-ary nodes for associative operators. For example, for the tree in Figure 4.2b, the chain of + nodes is replaced with a single "accumulation" + node. Figure 4.2c shows the resulting expression tree; the numbers on the nodes will be described shortly. The semantics of an accumulation node is as expected: the value is initialized as appropriate (e.g., 0 for +, 1 for $*$) and the contributions of the children are accumulated in arbitrary order.

We often consider a sequence of stencil computations—for example, in image processing pipelines [85]. Each computation in the sequence will be represented by a separate expression tree. Similarly, unrolling will result in distinct expression trees for each unrolled instance. For example, after unrolling along dimensions k and j in Listing 4.1, there will be a sequence of four expression trees. In some cases the output of a tree is used as an input to a later tree in the sequence. In such a case, there is a flow dependence: the root of the producer tree has the same label as some leaf node of the consumer tree (without an in-between tree that writes to that label). In the input to our analysis, this flow is represented by a dependence edge from the root node to the leaf node. Thus, the entire computation is represented as a DAG of expression trees.

Throughout the chapter, we make the following two assumptions: (1) the assembly instructions generated for the DAG of trees after register allocation are of the form $r_1 \leftarrow r_2 \; op \; r_3$, where r_1 and r_2 can be the same; (2) each operand/result requires exactly one register. This condition is only enforced to simplify the presentation of the next two sections, and can be very easily relaxed [5]. Our objective is to schedule the computations in the DAG so that the register pressure is reduced.

4.3.1 Sethi-Ullman Scheduling

We will use "data sharing" to refer to cases where the same memory location is accessed at multiple places. There are two types of data sharing: (1) within a tree: several nodes from N_{mem} have the same label; and (2) across trees: in a DAG of trees, nodes from distinct trees have the same label.

A classic result, due to Sethi and Ullman [93], applies to a *single* expression tree *without* data sharing (i.e., each $n \in N_{mem}$ has a unique label), and with binary/unary operators. They present a scheduling algorithm that minimizes the number of registers needed to evaluate such an expression tree under a spill-free model.[4] Each tree node n is assigned an Ershov number [1]; we will refer to them as "Sethi-Ullman numbers" and denote them by $su(n)$. They are defined as

$$su(n) = \begin{cases} 1 & n \text{ is a leaf} \\ su(n_1) & n \text{ has one child } n_1 \\ max(su(n_1), su(n_2)) & su(n_1) \neq su(n_2) \\ 1 + su(n_1) & su(n_1) = su(n_2) \end{cases} \tag{4.1}$$

The last two cases apply to a binary op node n with children n_1 and n_2. Intuitively, $su(n)$ is the smallest possible number of registers used for the evaluation of the subtree rooted at n. The first two cases are self-explanatory. For a binary op node n, if one child n' has a higher register requirement (case 3), this "big" child should be evaluated first. The result of n' will be stored in a register, which will be alive while the second ("small") child is being evaluated. The remaining $su(n') - 1$ registers used for n' are available (and enough) to evaluate the small child. Finally, the register of n' can be used to store the result for n, meaning that $su(n)$ is equal to $su(n')$. If the order of evaluation were reversed, the result of the small child would have to be kept in a register while n' is being evaluated, which would lead to sub-optimal $su(n) = 1 + su(n')$. In the last case in equation (4.1), both children have the same register needs; thus, their relative order of evaluation is irrelevant and one extra register is needed for n. Of course, under the definitions in equation (4.1), $su(n)$ is the same as MAXLIVE for the tree rooted at n.

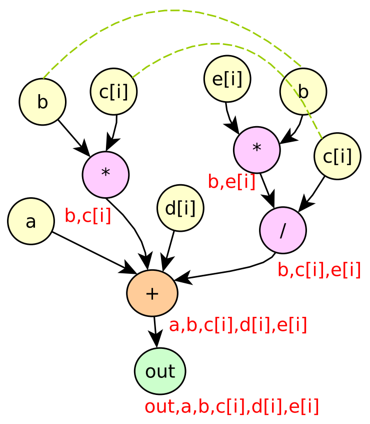
It is straightforward to generalize Sethi-Ullman numbering to trees containing accumulation nodes (as in Figure 4.2c). Each such accumulation node n has children n_i for $1 \leq i \leq k$. Let $mx = max_i\{su(n_i)\}$. If there is a single child n_j with $su(n_j) = mx$,

[4]In a spill-free model of the computation, a data element is loaded only once into a register for all its uses/defs.

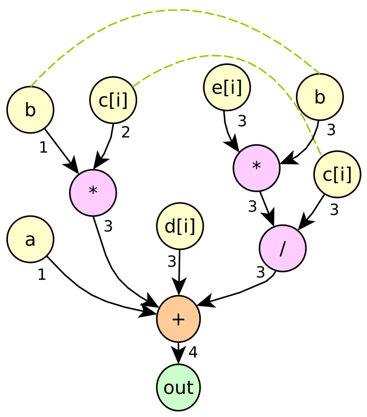
this child is scheduled for evaluation first, and therefore $su(n) = mx$. If two or more children n_j have $su(n_j) = mx$, one of them is scheduled first; however, in this case $su(n) = 1 + mx$. In both cases, the order of evaluation of the remaining children is irrelevant. Figure 4.2c shows the Sethi-Ullman numbers for the sample expression tree.

Note that the schedules produced by this approach perform *atomic evaluation* of subexpressions: one of the children is evaluated completely before the other ones are considered. For a tree without data sharing, this restriction does not affect the optimality of the result. In the presence of data sharing, atomic evaluation may not be optimal.

Since stencils read values from a limited spatial neighborhood, data sharing often manifests in the DAG of expression trees. For example, in Listing 4.1, $in[k][j][i]$ will be an input to all four expression trees corresponding to the unrolled stencil statements. One can also find other nodes in Listing 4.1 that will be shared across multiple expression trees. For such DAGs, the Sethi-Ullman algorithm cannot be directly applied to obtain an optimal schedule. In Section 4.3.2, we present an approach to compute an optimal atomic schedule for a DAG of expression trees *with* data sharing. In cases when finding an optimal evaluation can be prohibitively expensive, Section 4.3.3 presents heuristics to trade off optimality in favor of pruning the exploration space. Finally, restricting the evaluation to be atomic can generate suboptimal schedules. Section 4.4 presents a remedial slice-and-interleave algorithm that performs interleaving on the output schedule generated by the approach presented in Section 4.3.2.



(a) Tree with data sharing



(b) Scheduling cost

Figure 4.3: Scheduling a tree with data sharing

4.3.2 Scheduling a Tree with Data Sharing

Figure 4.3a shows an expression tree with data sharing. For illustration, nodes with the same label are connected in the figure. Recall that we assume a spill-free model, therefore a shared label loaded once into a register will remain live for all its uses. With data sharing, there is a possibility that (1) a label is already live before we begin the recursive evaluation of a subtree that has its subsequent use, and/or (2) a label must remain live even after the evaluation of the subtree in which it is used. The optimal schedule of a subtree is affected by the labels that are live before and after the evaluation of the subtree. Therefore, we need to add live-in/out states as parameters to the computation of the optimal schedule of a subtree. In this section, we present an approach to optimally schedule a tree with data sharing, under the model of atomic evaluation of children; we defer the interleaving of computation across subtrees to Section 4.4.

For a node n, let $uses(n)$ be the set of labels used in the subtree rooted at n. Figure 4.3a shows $uses(n)$ for each internal node n. The live-in set for a node n, denoted by $in(n)$, contains all labels that are live before the subtree rooted at n is evaluated. The live-out set is

$$out(n) \;=\; (in(n) \,\cup\, uses(n)) \,\setminus\, kill(n) \tag{4.2}$$

where $kill(n)$ is the set of labels that have their last uses in the subtree rooted at n. Note that $kill(n)$ is context-dependent, i.e., the set will vary depending on the order in which the node is evaluated. The kill sets can be computed on the fly by maintaining the number of occurrences of each label l in the current schedule, and comparing it with the total number of occurrences of l in the entire DAG.

We now show how to compute a modified Sethi-Ullman number, su' for each node n, when provided with an "evaluation context" in terms of live-in and live-out labels[5]. Consider a node n with some in and out state. Just before the evaluation of n begins, $|in(n)|$ registers are live. Similarly, just after the evaluation of n finishes, $|out(n)|$ registers will be live. During the evaluation of n, additional registers may become live, while some of the other live registers may be released. Now $su'(n, in, out)$ represents the maximum number of registers that were simultaneously live at any point during the evaluation of n. We also define $su'(\pi, in, out)$, where π is a sequence of the children nodes of n. This value will represent the maximum number of registers that were simultaneously live at any point during the evaluation of n with its children ordered in the sequence described by π.

For simplicity we will use $su'(n)$ instead of $su'(n, in, out)$, but the definitions will use the live-in/out sets $in(n)$ and $out(n)$.

[5]In [51], the context is expressed in terms of import and export nodes

For a leaf node $n \in N_{mem}$ with $|in(n)| = \alpha$,

$$su'(n) = \begin{cases} \alpha + 1 & label(n) \notin in(n) \\ \alpha & label(n) \in in(n) \end{cases}$$

The first case implies that a new register must be reserved for the label of n if it is not already live before the evaluation of n. The second case is self-explanatory.

To compute su' for a k-ary (binary or accumulation) node n with children $n_1 \ldots n_k$, we need to explore all $k!$ evaluation orders of the children. Let π be any permutation of the children of n representing their evaluation order. Then $su'(n) = \min_{\pi} su'(\pi)$.

For the purpose of explanation, suppose the permutation $\pi = \langle n_1, n_2 \rangle$ is one particular evaluation order for a binary node n with children n_1, n_2. To compute $su'(\pi)$, first we determine the live-in and live-out sets for nodes in π as follows: $in(n_1) = in(n)$, $in(n_2) = out(n_1)$, and $out(n_2) = out(n)$; here $out(n_1)$ is as defined in equation 4.2. This provides the required context to compute $su'(n_1)$ and $su'(n_2)$. Let $mx = max_i\{su'(n_i)\}$, so that mx equals the maximum number of simultaneously live registers at any time during the evaluation of π. Then,

$$su'(\pi) = \begin{cases} 1 + mx & n_i \in N_{mem} \;\&\; label(n_i) \in out(n) \\ mx & \text{otherwise} \end{cases}$$

In case 2, if $n_1 \in N_{op}$, or if $n_1 \in N_{mem}$ but $label(n_1) \notin out(n)$, then the result of the computation, identified by the label of the node n, can reuse the register of n_1 (similarly for n_2). However, in case 1, where both n_1 and n_2 are leaf nodes in N_{mem} and both must be live after evaluating n, we need an additional register to hold the result.

For an accumulation node with k children, consider permutation $\pi = \langle n_1, n_2 \ldots, n_k \rangle$. Suppose we have computed all $su'(n_i)$ and let $mx = max_i\{su'(n_i)\}$. Then,

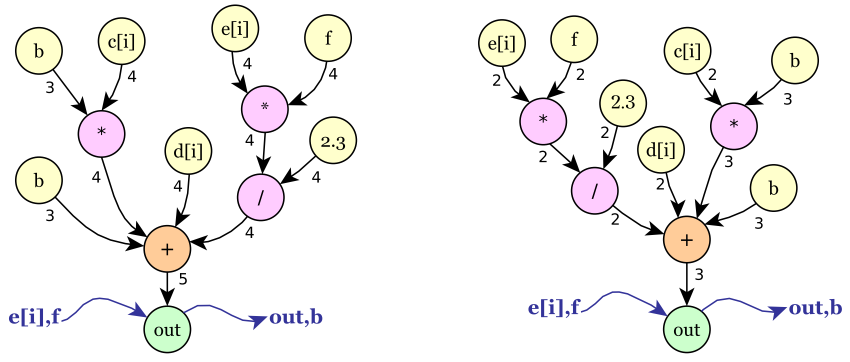
$$su'(\pi) = \begin{cases} mx & su'(n_1) = mx \,\&\, n_1 \in N_{mem} \,\&\, label(n_1) \\ & \notin out(n_1) \,\&\, su'(n_j) \neq mx, 2 \leq j \leq k \\ mx & su'(n_1) = mx \,\&\, n_1 \in N_{op} \,\& \\ & su'(n_j) \neq mx, 2 \leq j \leq k \\ 1 + mx & \text{otherwise} \end{cases}$$

Just like the generalization of $su(n)$ for accumulation nodes in Section 4.3.1, $su' = mx$ when the following two conditions hold: (1) n_1 requires the maximum number of simultaneously live registers, and the rest of the nodes in π can be completely evaluated using the registers released by n_1, and (2) the register holding n_1 can be reused by n, i.e., either $n_1 \in N_{op}$ (case 2), or n_1 is a leaf node that is not live beyond this point (case 1). In all other scenarios, we need $mx + 1$ registers (case 3).

The computation of $su'(n)$ for a tree without an evaluation context is shown in Figure 4.3b, and with an evaluation context is shown in Figure 4.4a. For the same tree, Figure 4.4b shows the permutation with minimum su'. In all three figures, the children of a node are ordered left-to-right, which defines the corresponding permutation.

In some cases, exhaustively exploring all permutations of the children may be unnecessary. In the tree of Figure 4.4a, there are two subtree operands of the accumulation node that share no data (the first and the third subtree operands in the left-to-right order). Therefore, even though the scheduling within those two subtrees may be influenced by the evaluation context, they do not influence each other's evaluation. Let *passthrough* denote the labels that are live both before and after the evaluation of node n: $passthrough(n) = in(n) \cap out(n)$. Then, for a k-ary node n, any two of its children n_i and n_j do not influence each other's evaluation if

$$(uses(n_i) \cap uses(n_j)) \setminus passthrough(n) = \varnothing \tag{4.3}$$



(a) Tree with context at root **(b)** Optimal schedule

Figure 4.4: Example: computing $su'(\pi)$

In such a scenario, for the node n, we can create maximal clusters of its children that share data, but the only data shared among clusters is the passthrough labels of n. For example, if children t_1 and t_2 share label l_1, and children t_2 and t_3 share label l_2, where l_1, l_2 are non-passthrough labels of the parent node, then $\{t_1, t_2, t_3, t_4\}$ must belong to the same cluster. We extend the intuition behind Sethi-Ullman scheduling algorithm to establish that different clusters cannot influence each other's evaluation. Then, for each cluster c_i, we can independently compute $su'(c_i)$ with $in(c_i) = in(n)$. We only need to explore all permutations within the non-singleton clusters. We propose the following theorems to establish an evaluation order for the clusters.

Theorem 1. *For k clusters c_i $1 \leq i \leq k$ such that $|in(c_i)| \leq |out(c_i)|$, the one with larger $su'(c_i) - |out(c_i)|$ will be prioritized for evaluation over others in the optimal schedule. In the special case where all the clusters have the same $su'(c_i) - |out(c_i)|$, they can be evaluated in any order without affecting MAXLIVE.*

This result is a direct consequence of the Sethi-Ullman algorithm. The cluster with larger $su'(c_i) - |out(c_i)|$ will release more registers, which can be reused by the next cluster. The special case too is a direct consequence of the Sethi-Ullman algorithm, where two sibling nodes with the same su can be evaluated in any order (case 4 of equation 4.1).

Theorem 2. *For two clusters c_1 and c_2 such that $|in(c_1)| > |out(c_1)|$ and $|in(c_2)| \leq |out(c_2)|$, c_1 must be evaluated before c_2 in the optimal schedule.*

We prove the result by contradiction. Suppose that c_2 is evaluated before c_1 in the optimal schedule. Since the schedule is optimal, $su'(c_2) \geq su'(c_1)$. Now we change this optimal schedule by moving the evaluation of c_1 before c_2. Evaluating c_1 earlier will release $|in(c_1)| - |out(c_1)|$ (i.e., ≥ 1) registers, which can then be used in the evaluation of c_2. Based on the previous equations, the $su'(c_2)$ will either decrease or remain the same, depending on whether the number of registers released by c_1 is greater than, or equal to 1. This modified schedule therefore either has the same, or has lower su' than the optimal schedule, making it an optimal schedule.

Theorem 3. *For two clusters c_1 and c_2 such that $|in(c_1)| > |out(c_1)|$ and $|in(c_2)| > |out(c_2)|$, the one with smaller su' must be prioritized for evaluation in the optimal schedule.*

Again, we prove the result by contradiction. Suppose that $su'(c_1) < su'(c_2)$, and c_2 is scheduled before c_1 in the optimal schedule. We change this schedule by moving the evaluation of c_1 before that of c_2. From Theorem 2, $su'(c_2)$ after this change will either remain the same, or decrease. Thus, su' for the new schedule will either be the same, or reduce if $su'(c_2)$ was the maximum, making it an optimal schedule.

Algorithm 4.1: Schedule-Tree (n, in, out): Optimal evaluation of an expression tree with data sharing under atomic evaluation constraint

```
   Input  : A tree rooted at $n$ with live-in/out contexts $in$ and $out$
   Output: An optimal schedule $S$ for the tree
1  $sched\_cost \leftarrow \varnothing$, $S \leftarrow \varnothing$;
2  $C \leftarrow create\_maximal\_clusters(n)$; (Sec. 4.3.2)
3  for each cluster $c$ in $C$ do
4      if $|c|$ is 1 then
5          $in(c) \leftarrow in(n)$;
6          $out(c) \leftarrow$ computed using equation 4.2;
7          $sched\_cost[c] \leftarrow su'(c)$;
8      else
9          compute $in$ and $out$ for each tree in $c$; (Sec. 4.3.2)
10         $\pi \leftarrow$ all permutations of the trees in $c$;
11         $sched\_cost[c] \leftarrow su'(\pi)$;
12 $P \leftarrow$ sequence clusters using $sched\_cost$ and Thms. 1, 2, 3;
13 for each subtree $t_s$ in the sequence described by $P$ do
14     append the schedule for $t_s$ in $S$;
15 return $S$;
```

Based on these theorems, Algorithm 4.1 summarizes the evaluation of an optimal schedule for a tree with data sharing.

4.3.3 Heuristics for Tractability

For a non-singleton cluster c, the algorithm presented in Section 4.3.2 can become prohibitively expensive if $|c|$ is large. For example, when $|c|$ changes from 7 to 8, the permutations explored increase from 5040 to 40320. We now present some heuristics that trade off optimality for tractability, and a caching technique to further speed up the algorithm.

Pruning Heuristics We begin by establishing, for any node n, the bounds on $su'(n)$. When n is evaluated with non-empty contexts in and out, the bounds are:

$$su'(n, \varnothing, \varnothing) \leq su'(n, in, out) \leq su'(n, \varnothing, \varnothing) + |in \cup out|$$

Simply put, the bounds imply that $su'(n)$ with context is always greater than $su'(n)$ without context, but only by, at most, the number of registers required to maintain the context. We prove both the lower and upper bounds by contradiction.

$su'(n, \varnothing, \varnothing) \leq su'(n, in, out)$: Assume to the contrary that $su'(n, in, out) < su'(n, \varnothing, \varnothing)$. We will modify the schedule S corresponding to $su'(n, in, out)$ as follows: prepend a stage to S that loads the labels $\in in(n)$ into $|in|$ registers, and make $in(n) = \varnothing$. Append a state to S that stores all the labels $\in out(n)$ from the respective registers into memory, and make $out(n) = \varnothing$. This modified schedule corresponds to $su'(n, \varnothing, \varnothing)$, and hence, $su'(n, \varnothing, \varnothing) = su'(n, in, out)$.

$su'(n, in, out) \leq su'(n, \varnothing, \varnothing) + |in \cup out|$: Assume to the contrary that $su'(n, \varnothing, \varnothing) + |in \cup out| < su'(n, in, out)$. Let S be a schedule corresponding to $su'(n, \varnothing, \varnothing)$. We modify S as follows: we assign a unique register r_k to each label $l_k \in (in \cup out)$. Now we prepend a stage to S that loads each label $l_i \in in$ into r_i. After every use of a label $l_j \in out$ in S, we add an assignment statement $r_j \leftarrow value(l_j)$. This modified schedule uses exactly $|in \cup out|$ extra registers, and is a schedule corresponding to $su'(n, in, out)$. Hence, $su'(n, \varnothing, \varnothing) + |in \cup out| = su'(n, in, out)$.

With the bounds established, instead of exploring all permutations, we can sacrifice optimality and stop further exploration when we are *close to* the optimal schedule. We use a tunable parameter d, and stop trying the permutations any further when $su'(n, in, out) - su'(n, \varnothing, \varnothing) \leq d$. For the experimental evaluation in Section 4.5, we set d to 1.

For a cluster c with $|c| > 8$, we also apply a partitioning heuristic, which recursively partitions the subtrees in c into sub-partitions where each sub-partition can be of a maximum size p, with $p < 8$. The partitioning is based on either of the two criteria:

- on "label affinity": the subtrees that share the maximum labels are greedily assigned to the same sub-partition as long as the size of the sub-partition is less than p. Such partitioning is based on the notion that evaluating subtrees with maximum uses together will potentially reduce passthrough labels, and MAXLIVE.
- on "release potential": the subtrees that have the last uses of some labels are placed in a sub-partition, and that sub-partition is eagerly evaluated. This partitioning is based on the notion that the released registers can be reused by the next partition.

Once the sub-partitions are created, we only exhaustively explore all permutations of subtrees within a sub-partition. If the number of sub-partitions created is less than 8, then we also try all the permutations of the sub-partitions themselves. For example, if $|c| = 8$, and the partitioning heuristic creates two sub-partitions p_1 and p_2 of size 4 each, then our exploration space will be $\{p_1, p_2\}$ and $\{p_2, p_1\}$, while exploring all 4! permutations of subtrees within p_1 and p_2 each – a total of $2 \times 4! \times 4!$ permutations instead of 8! permutations.

We also let the user externally specify a threshold that upper-bounds the total number of permutations for a tree.

Memoization For a node n, a lot of permutations of its children will differ in only a few positions. In such cases, we end up recomputing su' for a child multiple times, even when the live-in/out context for the child remains unchanged.

These recomputations can be avoided by a simple memoization, where for a node n, we map $su'(n)$ as a function of a *minimal* context. The minimal context strips away labels that are not in $uses(n)$, but are in $passthrough(n)$. The $su'(n)$ with minimal context can be suitably adjusted to get $su'(n)$ with a different context that has some passthrough labels added to the minimal live-in/out. For example, suppose that $su'(n)$ is 3 when the minimal $in(n) = \{\text{a,b}\}$ and the minimal $out(n) = \varnothing$. Then $su'(n)$ when evaluating it with $in(n) = \{a, b, c\}$, $out(n) = \{c\}$ and $c \notin uses(n)$ will be 2+1=3, and the optimal schedule will remain unchanged. Memoization greatly reduces the total evaluation time, thereby enabling the exploration of a large number of permutations.

4.3.4 Scheduling a DAG of expression trees

For each stencil statement that is mapped to an expression tree, Section 4.3.2 described a way to schedule it. This section ties everything together for a multi-statement stencil by describing how to schedule a DAG of expression trees. For optimal scheduling, one needs to explore all topological orders for the trees in the DAG, and then evaluate all the trees independently for each topological order. This may be practical if the size of the DAG is small. Otherwise, we must sacrifice optimality for tractability, and fix the evaluation order of the trees in the DAG before the trees are individually evaluated.

We use a version of the greedy heuristic proposed in Chapter 4 to fix the evaluation order of trees in the DAG. At each step, the heuristic tries to fix the evaluation order of two nodes in the DAG. We begin by computing, for each pair of transitive dependence-free trees p_i in the DAG, a metric M_i that encodes: (a) the number of labels shared between them, and (b) the number of common input arrays read by them. Among the computed M_i, we choose the one that has the highest non-zero value, and fix the evaluation order of its tree pair to be contiguous to enhance reuse proximity in the final schedule. The DAG is updated by fusing the nodes corresponding to the two trees into a "macro node". Post fusion, we update the dependence edges to and from the macro node, and recompute the metrics for the next step. The algorithm terminates when no more nodes can be fused.

Once the algorithm terminates, we perform a topological sort of the final DAG, and expand the DAG macro nodes to their tree sequences. For these ordered trees, we can generate code versions with different degree of splits. One extreme would be a version where all the trees are in a single kernel (*max-fuse*), and another extreme would be a version where each tree is a distinct kernel (*max-split*) [13, 117]. For compute-intensive stencils with many-to-many reuse, a single kernel can have extremely high register pressure, sometimes causing spills despite allowing for the maximum permitted registers per thread. For such cases, performing kernel fission instead of generating a single kernel for the entire computation might improve performance. The split kernels will incur additional data transfers from global memory, but the register pressure per kernel will be much lower, giving the user an opportunity to further enhance register-level reuse via unrolling. Note that none of the production GPU compilers are capable of performing kernel fusion/fission optimizations. For

Algorithm 4.2: Schedule-DAG (D, R): evaluating a DAG of expression trees under the atomic evaluation constraint

```
   Input  : $D$: DAG of expression trees, $R$: Per-thread register limit
   Output: An optimal schedule $S$ for the $D$
1  $D' \leftarrow D$; $fusion\_feasible \leftarrow true$; $tree\_order \leftarrow \varnothing$; $S \leftarrow \varnothing$;
2  while fusion_feasible do
3      for each pair of transitive dep-free nodes $t_i$,$t_j$ in $D'$ do
4          $M \cup = compute\_metric(D', t_i, t_j)$; (sec. 4.3.4)
5      $sort\_descending\ (M)$;
6      $(t_p, t_q, fusion\_feasible) \leftarrow find\_fusion\_candidate(M)$;
7      fuse $t_p$ and $t_q$;
8      $update\_dependence\_edges\ (D', t_p, t_q)$;
9  for each node $d$ in $D'$ do
10     append the tree sequence of $d$ in $tree\_order$;
11 $split\_versions \leftarrow create\_split\_versions(tree\_order)$;
12 for each split in split_versions do
13     $S' \leftarrow \varnothing$;
14     for each kernel $k$ in split do
15         for each tree $t$ in $k$ do
16             compute $in$ and $out$ for $t$;
17             append output of Schedule-Tree$(t, in, out)$ to $S'$;
18     execute $S'$ after compiling it with register limit $R$;
19     $S \leftarrow S'$if $S'$ is a faster schedule than $S$, or if $S$ is $\varnothing$;
20 return $S$;
```

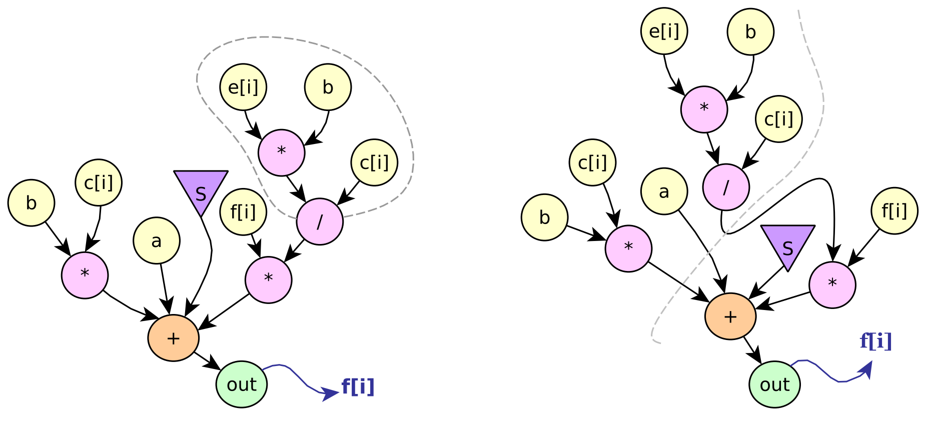
each split version created, the tree sequence in it is evaluated using Algorithm 4.1. The returned schedule is the one that gives maximum performance. Algorithm 4.2 outlines the entire process.

4.4 Interleaving Expressions

At this point, we have a schedule for the entire DAG of trees, but with atomic evaluation enforced. However, interleaving within/across trees can be instrumental in reducing MAXLIVE. For example, in the unrolled stencil of Listing 4.1, there is no reuse within a stencil statement, but plenty of reuse across stencil statements. We will see later in Section 4.5 that relaxing the constraint of atomic evaluation, and performing interleaving is imperative for performance in such stencils. A compiler optimization that performs some interleaving is common subexpression elimination (CSE). However, we require a more general interleaving that works at the granularity of common labels instead of common subexpressions. For example, Figure 4.5a shows an expression tree where $su'(S)$ is the largest, and the operands of the accumulation node are evaluated in order from left to right in the final schedule. Also, $\{c[i], b\} \notin uses(S)$. The fact that $\{c[i],b\} \in passthrough(S)$ adds to $su'(S)$. By slicing the expression $(e[i] * b)/c[i]$ and placing it after the expression $b * c[i]$ as shown in Figure 4.5b, $c[i]$ and b will no longer be in $in(S)$. Instead of those two labels, a temporary label holding the value of the sliced expression will be added to $in(S)$, and hence $su'(S)$ will reduce by 1. Note that this is not CSE, but a more general optimization aimed to reduce MAXLIVE. This slice-and-interleave optimization slices a *target* expression, and interleaves it with a *source* expression, so that su' at a program point reduces. It subsumes CSE if the source and target expressions are the same.

We perform the slice-and-interleave optimization at two levels: (a) within an expression tree, where the source and target expressions belong to the same tree; and (b) across the expression trees in the DAG, where source and target expressions belong to different trees. For a chosen source expression e_s rooted at node n, we compute a set of labels, L_{ilv}, which is a union of all the labels that were observed in the schedule till now, with the labels that have a single occurrence in the DAG.

We now try to find a set of target expressions operating on just the labels from L_{ilv}. To find the target expressions, we start with minimal expressions, i.e., the simplest expressions whose operands are leaf nodes $\in N_{mem}$. Once we find a minimal expression e_m that operates only on the labels $\in L_{ilv}$, we find the root node r of e_m, and



(a) Original tree **(b)** Interleaving expressions

Figure 4.5: Example: interleaving to reduce MAXLIVE

Algorithm 4.3: slice-and-interleave (T, in, out): Performing interlaving within an expression tree

```
   Input  : $T$: an input tree with schedule $S$ and contexts $in$ and $out$
   Output: $S$: The schedule after applying slice-and-interleave
1  $L_{ilv} \leftarrow \varnothing$;
2  $min\_exprs \leftarrow$ sequence of minimal expressions extracted from $S$ whose operands are
     leaf nodes;
3  for each expression $s$ in $min\_exprs$ do
4      $L_{ilv} \leftarrow$ all the labels seen in the schedule till $s \cup$ labels with single occurrence in
         T;
5      for each expression $t$ appearing after $s$ in $min\_exprs$ do
6          if $t$ only operates on the labels in $L_{ilv}$ then
7              $t' \leftarrow$ maximal expression obtained by growing $t$ until it operates on the
                 labels in $L_{ilv}$;
8              if live ranges reduced by placing $t'$ after $s$ then
9                  slice and place $t'$ after $s$ in $S$;
10                 move $t$ after $s$ in $min\_exprs$;
11 return $S$;
```

grow e_m to the expression rooted at $parent(r)$. We continue to grow the expression till we have a maximal expression that only operates on the labels $\in L_{ilv}$. For each target expressions thus discovered, we check if slicing and placing it between the source expression e_s and subtree t_s immediately following e_s in the schedule decreases $|in(t_s)|$. If it does, then slice-and-interleave is performed.

Illustrative Example Let $b*c[i]$ be the source expression in the tree of figure 4.5a. One of the explored target expressions will be $e[i]*b$, since it only uses nodes $\in L_{ilv}$. Now we try to grow the target expression by changing the root from $*$ to $/$, making $(e[i]*b)/c[i]$ the new target expression. All the labels used in the grown target expression also belong to L_{ilv}. A further attempt to grow will change the target expression to $((e[i]*b)/c[i])*f[i]$. However, $f[i] \notin L_{ilv}$. Therefore, we backtrack and finalize $(e[i]*b)/c[i]$ as a target expression, since it is the maximal expression with all the labels in L_{ilv}. Placing the target expression after the source expression decreases $in(S)$ by 1. Therefore, we perform the slice-and-interleave optimization. Algorithm 4.3 outlines the slice-and-interleave algorithm that tries out different source expressions, and continuously finds the target expressions within the tree to interleave in order to reduce the live ranges. The slice-and-interleave across the trees in a DAG is similar.

4.5 Experimental Evaluation

Our framework parses stencil statements written in a subset of C: the array access indices in the stencil statements must be an affine function of the surrounding loop iterators, program parameters, and constants; loop iterators and parameters must be immutable in the stencil statements. The framework supports auto-unrolling along different dimensions to expose spatial reuse across stencil statements.

Benchmark	N	UF	k	F	R	A	U
2d25pt	8192^2	4	2	33	2	104	44
2d64pt	8192^2	4	4	73	2	260	92
2d81pt	8192^2	4	4	95	2	328	112
3d27pt	512^3	4	1	30	2	112	58
3d125pt	512^3	4	2	130	2	504	204
hypterm	300^3	1	4	358	13	310	152
rhs4th3fort	300^3	1	2	687	7	696	179
derivative	300^3	1	2	486	10	493	165

N: Domain Size, UF: Unrolling Factor, k: Stencil Order, F: FLOPs per Point R: #Arrays, A: Total Elements Accessed per Thread, U: Unique Elements Accessed per Thread

Table 4.1: Characteristics of register-limited stencil benchmarks

To ensure a tight coupling, several prior efforts on guiding register allocation or instruction scheduling were implemented as a compiler pass in research/prototype compilers [11, 27, 35, 81, 90], or open-source production compilers [52, 91]. However, like some other recent efforts [8, 50, 99], we implement our reordering optimization at source level for the following reasons: (1) it allows external optimizations for closed-source compilers like NVCC; (2) it allows us to perform transformations like exposing FMAs using operator distributivity, and performing kernel fusion/fission, which can be performed more effectively and efficiently at source level; and (3) it is input-dependent, not machine- or compiler-dependent – with an implementation coupled to compiler passes, it would have to be re-implemented across compilers with different intermediate representation. Our framework massages the input to a form that is more amenable to further optimizations by any GPU compiler, and we use appropriate compilation flags whenever possible to ensure that our reordering optimization is not undone by the compiler passes.

We evaluate our framework for the benchmarks listed in Table 5.3 on a Tesla K40c GPU (peak double-precision performance 1.43 TFLOPS, peak bandwidth 288 GB/s) with NVCC-8.0 [75] and LLVM-5.0.0 compiler (previously gpucc [122]). The first five benchmarks are stencils typically used in iterative processes such as solving partial differential equations [45]. The remaining three are representative of complex stencil operations extracted from applications. *hypterm* is a routine from the ExpCNS Compressible Navier-Stokes mini-application from DoE [29]; the last two stencils are from the Geodynamics Seismic Wave SW4 application code [104]. For each benchmark, the original version is as written by application developers without any loop unrolling; the unrolled version has the loops unrolled explicitly; and the reordered version is the output from our code generator. On an Intel i7-4770K processor, the code generator generated each reordered version under 4 seconds. When the net unrolling factor is limited to 4, the size of each reordered version is under 600 lines of code. The read-only arrays are annotated with the *restrict* keyword in all the versions to allow efficient loads via the texture pipeline. All the stencils are double-precision, compiled with NVCC flags *'–use_fast_math Xptxas "-dlcm=ca"'*, and LLVM flags *'-O3 -ffast-math -ffp-contract=fast'*. Since none of the versions use shared memory, using *'dlcm=ca'* for NVCC enhances performance by caching the global memory accesses at L1. However, we notice no discernible performance difference when compiling without the *'–use_fast_math'* flag in NVCC. We explore different instruction schedulers implemented in LLVM (*default, list-hybrid, and list-burr*) for the original and unrolled code, and report the best performance. To minimize instruction reordering for our reordered code, we use LLVM's default instruction scheduler, and do not use the *-ffast-math* option during compilation. We test all the versions against a standard C implementation of the benchmarks for correctness: the difference in each computed output value with that of the C implementation must be less than a tolerance of 1E-5.

Loop Unrolling For the experiments, the iterative kernels were unrolled along a single dimension to expose spatial reuse. Loop unrolling offers the compiler an opportunity to exploit ILP, but scheduling independent instructions contiguously may increase register pressure. Consider an unrolled version of *2d25pt*, compiled with 32 registers. From table 5.3, it is clear that the unrolled code has a high degree of reuse. Listing 4.2 shows the SASS (Shader ASSembler) snippet generated using NVCC for the unrolled version of *2d25pt* after register allocation. The instructions not relevant to the discussion are omitted in Listing 4.2 (and 4.3), leading to non-contiguous line numbers. The lines highlighted in red show the instructions involving the same memory location – line 1 loads a value from global memory into register `R4`, and spills it in line 2 without using `R4` in any of the intermediate instructions. Such wasteful spills are a characteristic of register-constrained codes. The same value is reloaded from local memory into `R4` in line 4, and `R4` is subsequently used in lines 5 and 8. The uses of `R4` are placed far apart in SASS, adding to the register pressure. Interspersed with these instructions are the load (line 3) and subsequent uses of register `R12`. The interleaving increases ILP, but the uses of `R12` are also placed very far apart. A better schedule can perhaps achieve the same ILP with less register pressure and fewer spills.

Listing 4.3 shows the SASS snippet for the reordered code generated by our code generator. Using operator distributivity, the multiplication of the coefficient to the additive contributions is converted by our preprocessing pass into fused multiply-adds. Notice that all the uses of register `R20` (highlighted in red) are tightly coupled.

Listing 4.2: SASS snippet for the unrolled code

```
106    /*0328*/     @P0 LDG.E.64 R4, [R24];
144    /*0458*/     @P0 STL [R1+0x10], R4;
332    /*0a38*/     @P0 LDG.E.64 R12, [R8];
350    /*0ac8*/     @P0 LDL.LU R4, [R1+0x10];
354    /*0ae8*/     @P0 DADD R16, R16, R4;
358    /*0b08*/     @P0 DADD R16, R12, R10;
376    /*0b98*/     @P0 DFMA R6, R12, c[0x2][0x40], R14;
374    /*0b88*/     @P0 DADD R16, R6, R4;
436    /*0d78*/     @P0 DADD R12, R12, R24;
```

Listing 4.3: SASS snippet for the reordered code

```
163    /*04f0*/    @P0 DFMA R14, R22, c[0x2][0x8], R14;
164    /*04f8*/    @P0 DFMA R8, R22, c[0x2][0x18], R8;
166    /*0508*/    @P0 DFMA R12, R22, c[0x2][0x30], R12;
175    /*0550*/    @P0 DFMA R8, R20, c[0x2][0x30], R8;
176    /*0558*/    @P0 DFMA R12, R20, c[0x2][0x18], R12;
178    /*0568*/    @P0 DFMA R8, R30, c[0x2][0x38], R8;
183    /*0590*/    @P0 DFMA R22, R20, c[0x2][0x8], R10;
184    /*0598*/    @P0 DFMA R10, R20, c[0x2][0x18], R14;
187    /*05b0*/    @P0 DFMA R16, R30, c[0x2][0x20], R12;
191    /*05d0*/    @P0 DFMA R10, R30, c[0x2][0x20], R10;
```

The same holds for registers `R22,R30`, and the remaining instructions. Independent FMAs are scheduled together without increasing the MAXLIVE. This reduces register pressure without compromising ILP. Therefore, even though the unrolled version performs fewer FLOPs than the reordered version, we incur less spill LDL/STL instructions per thread (101 for unrolled vs. 7 for reordered).

For the *3d125pt* stencil, table 4.2 shows some profiling metrics gathered by Nvprof with NVCC. The texture throughput for the original code indicates that the stencil performance is limited by the texture cache bandwidth. Loop unrolling halves the accesses to texture cache and the executed load instructions, but results in a significant drop in IPC due to lowered occupancy. To better expose reordering opportunities

Version	reg	IPC	inst.exec.	ld/st	FLOP	L2 read	tex txn	tex GB/s
Original	128	1.76	2.7E+9	5.3E+8	1.7E+10	5.3E+8	4.2E+9	899.5
Unrolled	255	1.12	1.4E+9	2.1E+8	1.7E+10	2.9E+8	1.7E+9	457.2
Reordered	64	2.00	1.4E+9	2.1E+8	3.3E+10	1.5E+8	1.7E+9	791.2

Table 4.2: Metrics for $3d125pt$ for tuned configurations

after unrolling, the preprocessing pass of the reordering framework exploits operator distributivity and converts all the contributions in an individual statement to FMA operations. Therefore, instead of 130 FLOPs per stencil point, the reordered version performs 250 FLOPs. As measured by Nvprof, we incur a 2× increase in floating point operations, but achieve significant reuse in registers at a higher occupancy, which consequently improves the IPC and execution time.

Register Pressure Sensitivity In GPUs, the number of registers per thread can be varied at compile time by trading off the occupancy. Many auto-tuning efforts have recently been proposed to that end [41, 58]. Table 4.4 shows the performance with NVCC compiler, and the local memory transactions reported by Nvprof by varying register pressure. We make the following observations: (a) our optimization strategy reduces the register pressure for all the thread configurations; (b) increasing registers per thread for codes exhibiting very high spills results in better performance, e.g., 8× for *rhs4th3fort*; and (c) for low spills, better performance can be achieved by either increasing occupancy (e.g., reordered code for *3d125pt* and *hypterm*), or maximizing registers per thread (e.g., all the codes for *rhs4th3fort*).

Finding a right balance between register pressure and occupancy is non-trivial, and an active research field [109, 123, 41, 58]. We perform a simple auto-tuning by

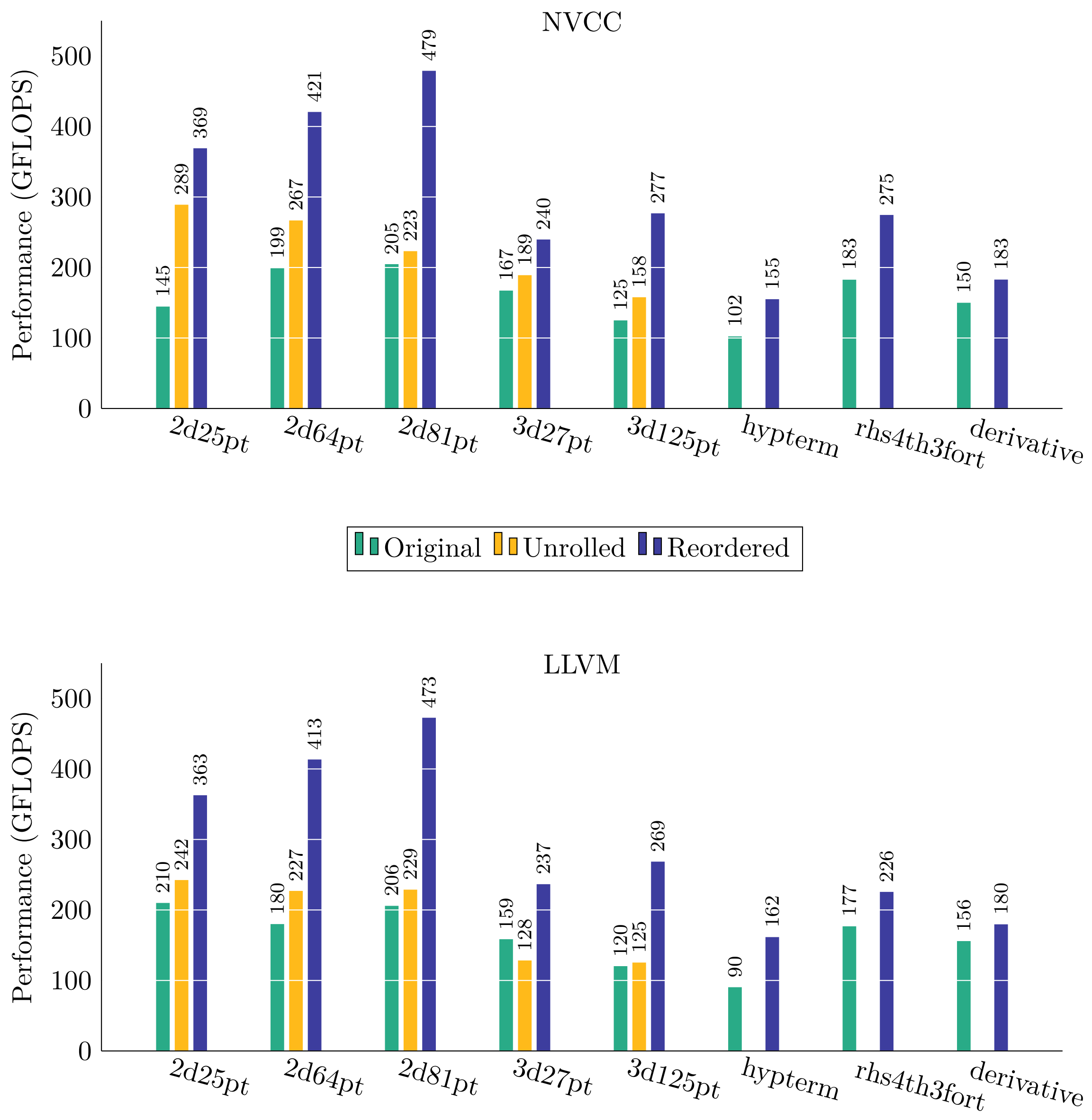


Figure 4.6: Performance on Tesla K40c with the register-limited stencil benchmarks tuned for tile size and register limit

varying the tile sizes by powers of 2, and varying registers per thread [58]. The best performance in GFLOPS for the auto-tuned code with NVCC and LLVM compilers is shown in figure 4.6. Unlike the case with 32 and 64 registers per thread, the unrolled

Metrics	*rhs4th3fort*		*hypterm*		*derivative*	
	maxfuse	split-3	maxfuse	split-3	maxfuse	split-2
Inst. Exec.	8.52E+9	8.25E+8	7.48E+8	7.71E+8	8.79E+8	8.96E+8
IPC	1.07	1.11	0.97	1.06	1.02	1.14
DRAM reads	9.07E+7	1.65E+8	1.57E+8	1.77E+8	1.34E+8	2.47E+8
ldst exec.	1.55E+8	1.08E+8	1.27E+8	1.46E+8	1.45E+8	1.30E+8
FLOPs	1.73E+10	1.81E+10	9.66E+9	9.36E+9	1.28E+10	1.34E+10
tex txn.	1.11E+9	8.24E+8	9.72E+8	1.06E+9	1.14E+9	1.01E+9
l2 read txn.	4.64E+8	3.79E+8	6.52E+8	5.90E+8	4.97E+8	4.51E+8
GFLOPS	237.16	274.52	140.71	155.02	168.27	182.83

Table 4.3: Metrics for reordered max-fuse vs. split versions for high-order stencil DAGs

code outperforms the original code for all benchmarks, highlighting the importance of loop unrolling and register-level reuse. Our reordering optimization improves the performance by (a) producing a code version that uses fewer registers, and hence can achieve higher occupancy; and (b) helping expose and schedule independent `FMA`s together for simple accumulation stencils, thereby hiding latency.

Kernel fission From table 5.3, we select the last three multi-statement, compute-intensive stencils, for which we anticipate high volume of spills in the max-fuse form, and expect kernel fission to be beneficial. For these three stencils, we generate versions with varying degree of splits (Section 4.3.4). Some splits require promoting the storage from scalars to global arrays, while others require recomputations due to dependence edges in the DAG. Table 4.3 shows the Nvprof metrics with NVCC for two reordered codes: a version with maximum fusion (max-fuse), and a version with split kernels. Note that in each case, even though the DRAM reads increase going from max-fuse to split version, the IPC also increases. This is because the register pressure per kernel is much lower in the split version, and hence we can unroll the computation to further exploit register-level reuse. This increase in register-level reuse is reflected in the reduced L2 read transactions. We observe nearly 10% performance improvement for the split version over max-fuse version for all three stencils. Prior works have noted the importance of kernel fusion for bandwidth-bound stencils [116, 38, 87], and trivial kernel fission to aid fusion by reducing shared memory usage [117]. Such a kernel fission has very limited applicability in stencils with significant many-to-many reuse across statements. However, our motivation for kernel fission is orthogonal to prior efforts – we use kernel fission as a means to reduce the register usage of the max-fuse kernel, and then improve the register reuse for split kernels by ample unrolling and instruction reordering.

With our reordering optimizations applied to the benchmarks, we achieve speedups in the range of 1.22×–2.34× for NVCC, and 1.15×–2.08× for LLVM. We finally discuss the effect of optimizations discussed in Section 4.3.3. The benchmark *derivative* has a large number of independent trees; each tree is an accumulation. The framework takes 3.42 seconds to generate code for it, and the memoization function is invoked 1.42E+05 times. Without memoization, the code generation time increases to 5.71 seconds. Our framework is well suited to enhance the performance of "optimize once, execute multiple times" stencils found in production codes where the compilation/optimization time is amortized over the stencil execution.

4.6 Summary

Despite a rich literature on register allocation and instruction scheduling, current production compilers are not sufficiently effective in reducing register pressure for

Bench.	Reg	Original		Unrolled		Reordered	
		LMT	GFLOPS	LMT	GFLOPS	LMT	GFLOPS
2d25pt	32	1.83E+7	**144.56**	1.18E+8	57.34	4.21E+6	302.12
	48	0	127.35	0	**289.03**	0	357.76
	64	0	115.03	0	261.43	0	**369.09**
2d64pt	32	3.39E+7	111.75	7.17E+8	18.07	6.74E+6	315.49
	48	0	191.17	4.04E+8	26.03	0	393.90
	64	0	**198.95**	2.76E+8	38.89	0	**420.61**
	128	0	146.64	1.31E+7	**231.72**	0	303.02
2d81pt	32	3.94E+7	101.72	8.2E+8	19.49	4.62E+6	426.87
	48	0	202.73	5.13E+8	25.12	0	466.03
	64	0	**204.73**	3.96E+8	30.67	0	**478.95**
	128	0	151.38	7.11E+07	161.00	0	415.30
	255	0	83.58	0	**223,17**	0	340.90
3d27pt	32	4.28E+7	126.07	2.24E+8	37.93	1.65E+7	182.53
	48	0	**167.23**	2.01E+7	149.51	0	229.01
	64	0	160.55	0	172.37	0	**269.69**
	128	0	160.93	0	**180.49**	0	211.78
3d125pt	32	5.04E+7	99.77	2.13E+9	19.06	1.85E+7	327.64
	48	0	96.26	1.42E+9	21.65	0	**339.16**
	64	0	107.61	1.35E+9	25.62	0	336.77
	128	0	**143.12**	5.25E+8	64.00	0	282.68
	255	0	93.64	0	**188.36**	0	173.738
hypterm	32	9.14E+8	21.82	-	-	4.51E+7	83.83
	48	1.04E+8	74.33	-	-	0	100.63
	64	0	100.66	-	-	0	138.12
	128	0	**102.27**	-	-	0	**155.02**
	255	0	74.92	-	-	0	141.52
rhs4th3fort	32	2.10E+9	19.93	-	-	1.37E+9	31.47
	48	1.21E+9	28.76	-	-	4.30E+8	87.82
	64	8.73E+8	38.26	-	-	9.99E+7	171.01
	128	1.60E+8	166.59	-	-	0	241.16
	255	0	**182.67**	-	-	0	**274.52**
derivative	32	1.63E+9	13.54	-	-	6.56E+8	34.25
	48	1.16E+9	17.04	-	-	1.29E+8	84.69
	64	8.90E+8	20.91	-	-	0	116.02
	128	3.60E+8	53.15	-	-	0	153.61
	255	0	**149.95**	-	-	0	**182.83**

LMT: Local Memory Transactions

Table 4.4: Spill metrics and performance in GFLOPS for register-limited stencils when compiled with NVCC on K40c device for different register configurations

compute-intensive high-order stencil codes. For such codes, register spills are a major performance limiter. Unfortunately, the compiler fails to perform an instruction reordering that can relieve register pressure, and the reordering it does perform to increase ILP often increases register pressure.

This chapter presents a register optimization framework for multi-statement, high-order stencils, which views such computations as a collection of trees with significant data reuse across nodes, and systematically attempts to reduce register pressure by decreasing the simultaneously live ranges. Just as pattern-specific optimization techniques have demonstrably been more beneficial over traditional compiler optimizations for stencil computations, we show through this work that a specialized register management framework can be highly beneficial for stencil computations.

CHAPTER 5

A Generalized Associative Instruction Reordering Scheme to Alleviate Register Pressure in Straight-Line Codes

5.1 Introduction

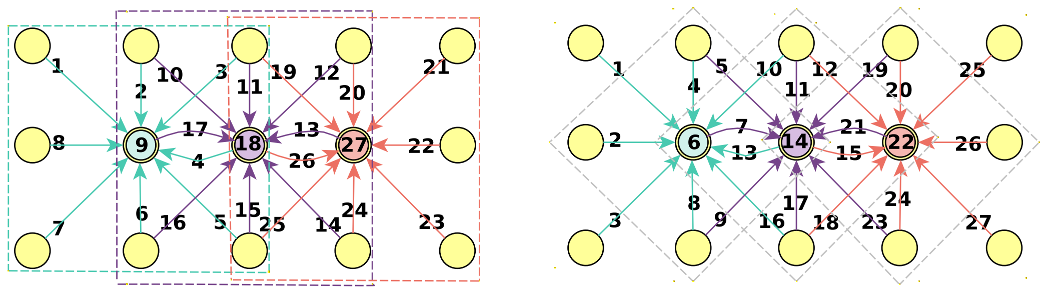
Register allocation is generally considered a practically solved problem. For most programs, the register allocation and instruction scheduling strategies in production compilers are very satisfactory, and the number of register spills is well controlled. However, this is not the case for many compute-intensive array-based computations applications in computational science and machine-learning/data-science that feature multiple inter-related reuses. With such computations, a number of variables in a basic block of instructions exhibit many coupled reuses, with different groups of instructions referencing different combinations of the set of reused variables. As shown with quantitative data later in the chapter, when a large number of data elements is involved in the many-to-many reuses, existing register management strategies in production compilers are unable to effectively control the number of register spills. These many-to-many reuse patterns typically involve associative accumulation of multiple contributions into destination variables. In this chapter, we develop an effective instruction reordering strategy that exploits the flexibility offered by associative operations. We demonstrate significant alleviation of register pressure and spilling and enhancement of performance.

We use a high-order "box" stencil computation to elaborate on the addressed problem. First, let us consider a simple 2D 9-point Jacobi computation shown in Listing 5.1.

Listing 5.1: 2D 9-point Jacobi computation

```
for(i=1;i<N-1;i++)
  for(j=1;j<N-1;j++)
     A[i][j] = c*(B[i-1][j-1] + B[i-1][j] + B[i-1][j+1] +
                  B[i][j-1] + B[i][j] + B[i][j+1] + B[i+1][j-1] +
                  B[i+1][j] + B[i+1][j+1]);
```

For computing each element $A[i][j]$, the corresponding element $B[i][j]$ and all 8 neighboring elements are read. For two adjacent result elements, $B[i][j]$ and $B[i][j+1]$, six of the needed elements from A are common. By explicitly unrolling the inner j loop, these potential reuses are exposed in the resulting basic block of code in the unrolled loop. Additional ILP (Instruction Level Parallelism) is also exposed for the compiler to exploit. Figure 5.1a shows the output and input data elements accessed by the code for 3-way unrolling of the j loop. The order of accesses to elements of A for such an unrolled code is shown in Figure 5.1a: each output point is computed in entirety before the computation of the next output point begins. As the order of the stencil increases, the number of simultaneously live variables grows quadratically for the "natural" order of access for elements of A shown in 5.1a. For a 4-way unrolled version of an 81-point convolution stencil, with GCC-7.2.0 as the base compiler on an Intel i7-6700K processor, we observe the number of memory accesses per iteration increase from 269 to 664 in the generated assembly code, and the performance drop



(a) gather-gather scheme **(b)** scatter-gather scheme

The number on the edges represents the order of contributions

Figure 5.1: Different contribution schemes for the 3-way unrolled version of 2D 9-point Jacobi computation

from 72 GFLOPS to 70 GFLOPS when going from the original non-unrolled form to the unrolled version.

For such high-order stencils, the problem has been recognized in prior work [99] and a solution provided - exploit associativity to reorder operations and create a different equivalent stencil pattern. With this changed access pattern, it has been shown that the maximum number of concurrently live registers reduces from $O(k^2)$ to just $O(k)$ for an order-k box stencil [99]. Rearranging the contributions for the 81-point convolution stencil in a manner as shown in Figure 5.1b, where the contribution from an input value is “scattered” to different output points, brings down the memory accesses per iteration in the unrolled code to 206, and the performance increases from 70 GFLOPS to 137 GFLOPS.

Accumulator expansion is performed in compilers, but is done in a local manner without considering impact on register pressure. For simple single-statement high-order stencils, a retiming base solution to alleviate register pressure was proposed by Stock et al. [99]. However, the problem of high register pressure and subsequent register spills arises in more complex settings in many scientific applications, e.g., multiple inter-related stencils, tensor contractions, etc. In this chapter we develop a general solution for the associative reordering problem, without using any specialized abstractions such as stencil patterns as was used by Stock et al. [99]. We devise an instruction reordering strategy that simultaneously considers the flexibility with associative reordering of accumulations and the impact of instruction order on the maximum live values.

We develop a **L**ist-based, **A**daptive, **R**egister-reuse-driven **S**cheduler (LARS) that uses multiple criteria, including affinities of non-live variables to those live in registers, as well as potentials of variables to fire operations and to release registers. We chose to implement the instruction scheduler as a source-to-source pass outside the compiler, to be executed before the standard compiler passes. By doing so, we are able to evaluate its impact with two production compilers, LLVM and GCC. We present experimental results on a large collection of benchmarks that exhibit significant potential register-level reuse for array elements. We demonstrate significant benefits from use of LARS.

The chapter makes the following contributions:

- It develops a framework to reduce register pressure by exploiting the flexibility of associative reordering for computational kernels with multiple inter-related reuses.
- It develops a flexible multi-criteria instruction reordering heuristic that can be adapted across different architectures.

- It demonstrates the effectiveness of the proposed framework for a number of scientific kernels when compiled with different compilers and on multiple architectures.

5.2 Background and Motivation

Register Allocation and Instruction Scheduling Register allocation and instruction scheduling are among the most important backend passes in a compiler. Register allocation assigns physical registers to the variables used in the intermediate representation (IR). All variables that are live at a given program point must be assigned to distinct registers if register spills are to be avoided. We denote as MAXLIVE the maximum number of variables that are simultaneously live at any program point through the execution of the program. A *data reuse* refers to multiple accesses to the same variable. Once a variable is loaded from memory, it is desirable to keep it in a register throughout its live range. However, that may not be possible due to the limited number of available physical registers. *Spill* instructions are generated when the allocator runs out of physical registers: some registers are freed after their contents are stored into memory or simply discarded. The spilled contents must be reloaded into registers before their subsequent use. Since memory accesses have higher latency than other instructions, minimizing spills is important for performance. Several techniques have been proposed to perform register allocation. Most compilers use versions of graph coloring [16] or linear scan [81]. Both these techniques perform register allocation on a fixed instruction schedule that is obtained after performing instruction scheduling on the IR. The instruction scheduler orders the instructions to minimize the schedule length and simultaneously increase ILP, so that the functional units and pipelines of the underlying processor are effectively utilized. Clearly, the objectives of instruction scheduling and register allocation can be antagonistic: instruction scheduling may prefer independent instructions scheduled in proximity to increase ILP, whereas register allocation may prefer data-dependent instructions to be scheduled in proximity to shorten the live ranges. This problem is more pronounced for applications that exhibit complex, many-to-many data reuse pattern: the instruction scheduler does not consider the reuse pattern while generating the initial instruction ordering, resulting in increased live ranges for variables.

Associative Instruction Reordering Sometimes, a better instruction reordering can be achieved if one leverages associativity of operations. Although floating-point additions are not strictly associative, it is generally acceptable to perform associative reordering of accumulations in most applications that do not rely on rounding behavior. In fact, many scientific computations are compiled using *-ffast-math -ffp-contract=fast* flags[6], which allows a compiler to exploit associativity of floating-point operations to improve performance at the expense of IEEE compliance. Many recent efforts have leveraged operator associativity to drive code optimization strategies [99, 8, 103].

Consider the input program of Listing 5.2. The two statements read from four common input values. If the computation of the first statement entirely precedes the second statement, then these four values must be kept alive in registers. However, one can leverage the associativity of addition to reorder the computation as shown in Listing 5.3. In the reordered computation, the evaluation of the two output points is

[6]`https://gcc.gnu.org/wiki/FloatingPointMath`

Listing 5.2: An unrolled computation with reuse in inputs

```
for (int j=2; j<N-2; j++) {
     for (int i=2; i<N-2; i++) {
          A[j][i] = a*B[j-2][i] + b*B[j-1][i] + c*B[j][i] +
                    d*B[j+1][i] + B[j][i]*B[j+2][i];
          A[j+1][i] = p*B[j-1][i] + q*B[j][i] + r*B[j+1][i] +
                 s*B[j+2][i] + B[j+1][i]*B[j+3][i];
     }
}
```

Listing 5.3: A reordering for the computation of Listing 5.2 that leverages associativity of + to reduce register pressure

```
for (int j=2; j<N-2; j++) {
     for (int i=2; i<N-2; i++) {
          A[j][i]    = a*B[j-2][i];
          A[j][i]    = c*B[j][i] + B[j][i]*B[j+2][i];
          A[j+1][i]  = s*B[j+2][i] + q*B[j][i];
          A[j][i]   += b*B[j-1][i] + d*B[j+1][i];
          A[j+1][i] += p*B[j-1][i] + r*B[j+1][i];
          A[j+1][i] += B[j+1][i]*B[j+3][i];
     }
}
```

interleaved, so that all the uses of an input value are brought closer, and consequently, its live range is shortened. However, most production compilers perform instruction scheduling and register allocation on an IR that is closer in spirit to the Static Single Assignment (SSA) IR. In SSA, the accumulation operations are converted to a *use-def* chain of contributions. Lifting the SSA IR back to a higher abstraction to leverage the operator associativity is difficult, and therefore most compilers fail to fully utilize such a transformation to relieve register pressure.

Solution Approach Many prior works have studied the implications of phase ordering between register allocation and instruction scheduling on the generated code

[80, 74, 11]. These works proposed an integrated register allocation and instruction scheduler for VLIW or in-order issue superscalar processors. However, none of these works leverage properties of the operators involved in the computations to improve the instruction reordering. We show in Section 5.4 that for many scientific applications executed on out-of-order (OoO) processors, register spills are a performance bottleneck. To this end, we propose LARS – a greedy instruction reordering framework for straight-line code, that a) operates at source-level to fully exploit the associativity of operations, b) has a much better global perspective of the computational reuse pattern, and c) reorders the instructions in the computation to minimize MAXLIVE.

5.3 Instruction Reordering With LARS

5.3.1 Preprocessing Steps

LARS parses statements within a computational loop nest that are expressed in a subset of C with the following restrictions: (1) the loop iterators and the program parameters must be immutable in the statements; (2) the right-hand side expression of a statement must be side-effect free. Side-effect-free mathematical functions like *sine, sqrt*, etc. are allowed in the RHS expression; and (3) the array index expressions in each statement must be an *affine* function of the loop iterators, program parameters, and literals. LARS therefore is capable of parsing the stencils expressed in StencilGen language.

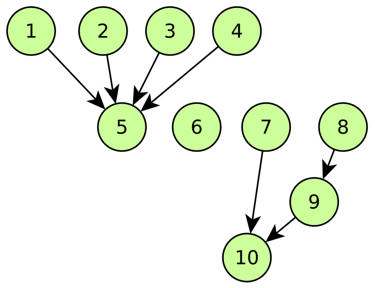
Based on the unroll factors specified in the StencilGen input to LARS, a preprocessing pass performs loop unrolling, and then lowers each statement in the optimization region into a sequence of instructions using operator associativity and/or distributivity. The instructions are somewhat similar in spirit to the three-address

```
t1 = ((d*c) + (b/c) + (b*e) + (d/f)) * g + k * g;
        t2 = (n*p) + ((f+e) * p);
```

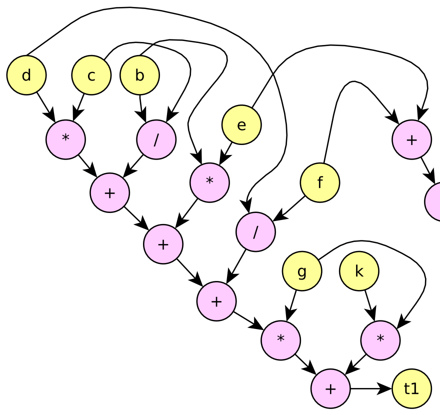
(a) Illustrative computation

```
 1.  a = d * c;
 2.  a += b / c;
 3.  a += b * e;
 4.  a += d / f;
 5.  t1 = a * g;
 6.  t1 += k * g;
 7.  m = n * p;
 8.  q = f + e;
 9.  r = q * p;
10.  t2 = m + r;
```

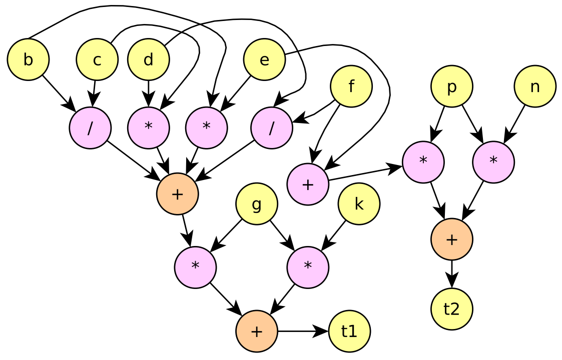
(b) Initial Schedule



(c) Dependence Graph



(d) Input CDAG



(e) CDAG with accumulations

Figure 5.2: Example: statements lowered down to sequence of instructions, and corresponding CDAG

GIMPLE IR of GCC, where the right-hand side of an instruction has at most two operands, and the operator is either an assignment or an accumulation (Appendix A). In terms of assembly language, these instructions are synonymous to the register-register instructions ($r_1 \leftarrow r_2 \; op \; r_3$ where r_1 and r_2 can be the same register). Figure 5.2a shows an illustrative computation which is lowered into the instructions shown in

Figure 5.2b using the associativity and distributivity of $+, *$. Note that even though all the operands in the illustrative example are scalars, in practice, the operands can be a mix of array accesses, scalars, and literals. An abstraction commonly used to represent such computations is a computational DAG (CDAG) [33, 93], where the leaf nodes represent the storage locations read, the root represents the output, and the internal nodes represent the operators. For example, Figure 5.2d shows the CDAG corresponding to the computation of Figure 5.2b, whereas Figure 5.2e shows the CDAG corresponding to the instructions after converting the additive contributions in the original DAG to accumulations, represented in the figure by orange "accumulation +" nodes. Throughout the text, we will shift from instruction sequence to their CDAG abstraction for ease of explanation.

In order to reduce register pressure, LARS must gauge the data reuse between the instructions in the original schedule. To recognize the common uses of a value, the accessed storage locations are assigned a *label*. All the accesses to the same storage location within a statement will have the same label. Across statements, the accesses to a storage location M will be identified by the same label if there is no write to M in between their execution.

5.3.2 Creating Multiple Initial Schedules

The time taken to reorder instructions is significantly lower with a greedy heuristic than techniques like dynamic programming [32] or integer linear programming [17]. We use this to our advantage by reordering multiple versions of the input program instead of just one, and choosing the reordering with the best performance.

One can rewrite the pragma-demarcated computation as different dependence-preserving permutations of input statements; each permutation can produce a re-ordering schedule with different register requirement. A brute-force approach will examine all the dependence-preserving permutations of the input statements, and apply LARS to each permutation. However, for a program with n independent statements, this would imply exploring $n!$ statement sequences, a formidable task as n increases. Instead, we use a simple algorithm to generate a few different permutations that cluster the statements with reuses together, and then apply LARS to only these few statement permutations.

The algorithm operates on the dependence graph G of the statements. For each pair of nodes in G, the algorithm computes the number of common labels between them. The statement pair with the highest reuse must be closer in the permutations. The algorithm ensures this by *merging* the two nodes with the highest reuse into a "macro-node" in G. Merging is only valid if it is dependence-preserving. After merging nodes s_i and s_j, G is modified appropriately - the incoming (outgoing) edges to (from) s_i and s_j become the incoming (outgoing) edges to (from) the merged node, and the edges between s_i and s_j are removed. This process is repeated till only a single macro-node remains. The permutations are obtained by expanding the final macro node into statements: we expand a macro-node comprising two statements $\{s_i, s_j\}$ as $[s_i, s_j]$ in one permutation, and $[s_j, s_i]$ in another, depending on the dependence between s_i and s_j. We restrict the number of the generated statement permutations to a user-specified limit. Once we have the permutations, we choose one permutation at a time, apply the preprocessing step described in Section 5.3.1 to it, and then use LARS to reorder the version. The final reordered version is the one that is more efficient in execution. For the rest of the discussion, we will assume that the input to LARS is a valid permutation P.

5.3.3 Overview of the Reordering Strategy

Given the initial schedule P, the main objective of LARS is to compute a reordered schedule that preserves the dependences, and reduces register pressure while maintaining sufficient ILP. We use the initial schedule of Figure 5.2b as an example to give a brief overview of the reordering strategy implemented in LARS.

Prior to reordering, we construct a dependence graph (DG) for the initial schedule. Since we allow the contributions to an accumulation node to be in arbitrary order, we do not include the true/output dependences on the accumulation node in DG. Figure 5.2c shows the dependence graph for the schedule of Figure 5.2b.

Next, we compute the labels occurring in the initial schedule. Figure 5.3a shows the mapping from the labels to the instructions in which they appear. We maintain a set L of labels that are currently live in pseudo registers, assuming a *spill-free*[7] model of computation. An instruction I_j in the schedule can fall in one of the following three categories:

- *Blocked:* If I_j has a true dependence on another instruction I_k in the DG, and I_k has not yet been fired
- *Unblocked:* If I_j is not blocked
- *Fireable:* If I_j is unblocked, and all the labels of I_j are in L.

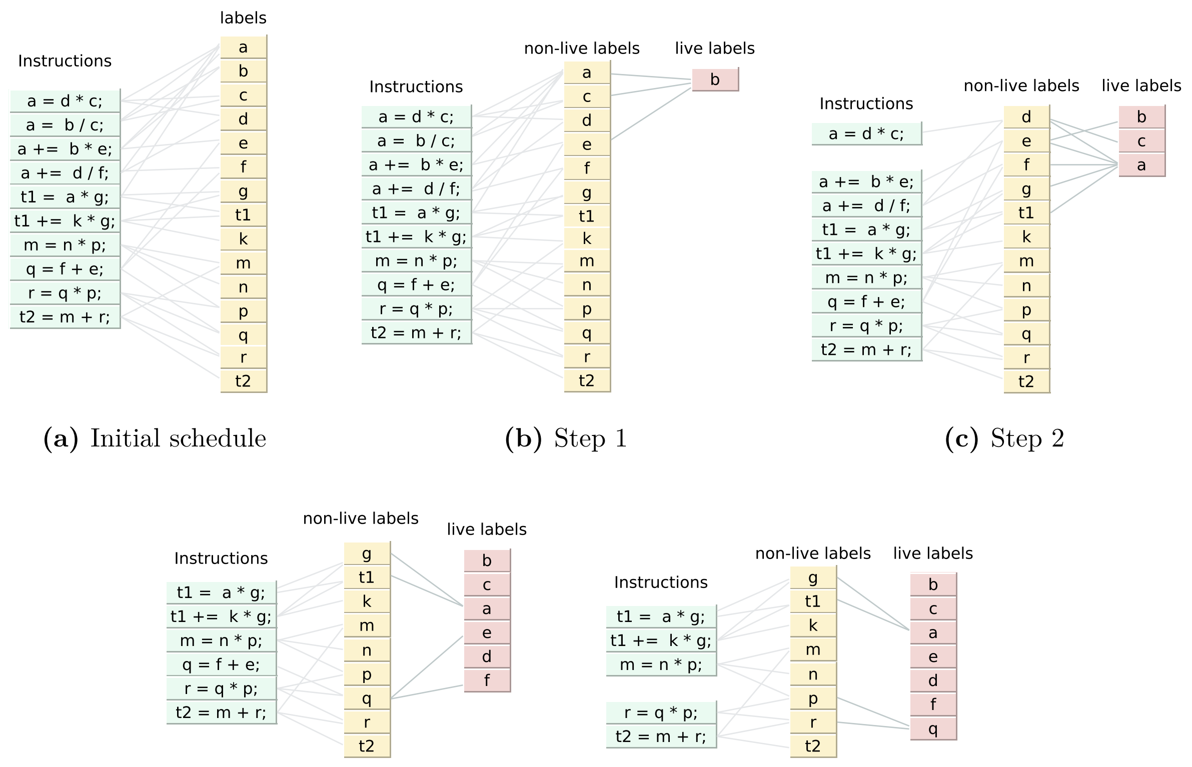
Initially, only the instructions with no incoming dependence edges in DG are unblocked (i.e., instructions 1, 2, 3, 4, 6, 7, and 8 in the schedule of Figure 5.2b), and none of the labels are live. The reordering algorithm can be summed up as

[7]a value once loaded in a register will remain so for all its def/uses

two iterative steps: make the labels live in some order, so that instructions become fireable, and append the fired instructions into the final reordered schedule. The order in which the labels are made live and the fired instructions are appended to the reordered schedule will affect the register pressure; the heuristic used to determine the order forms the core of this section.

In order to determine the first *seed* label that becomes live, we compute an initial priority metric for each label; the computation is described in details in Section 5.3.5. Simply put, the labels occurring in instructions that are closer to the source of the longest (critical) path in the CDAG are assigned a higher initial priority. This conforms to the priority assignment in the instruction scheduling phase of most production compilers, like GCC. For example, the critical path in the schedule of Figure 5.2b involves the accumulation into a. Since division operation has the highest latency, we assign high initial priority to labels $\{a,b,c,d,f\}$. Among these, let us assume that LARS chooses b as the seed label (Figure 5.3b).

Once a label is made live, the algorithm checks if any unblocked instruction becomes fireable. If so, the algorithm fires/executes it, and appends it to the final reordered schedule. Otherwise, a cost tuple is constructed for each label, which captures the interaction of the label with the already live labels, and the effect of making that label live on the current unblocked statements (Section 5.3.5). The label deemed most profitable by the cost tuple is iteratively made live till one or more of the unblocked instructions becomes fireable. In our example, none of the unblocked instructions become fireable when the seed label b is made live, more labels need to be made live in order to fire any instruction. Figure 5.3b shows the interaction between the live label b, and the non-live labels $\{a,c,e\}$: they appear with b in instructions 1,



(a) Initial schedule **(b)** Step 1 **(c)** Step 2

(d) Step 3 **(e)** Step 4

Figure 5.3: Applying LARS reordering strategy to the input of Figure 5.2b

2, and 3 respectively, and hence, their live ranges interfere with *b*. This interaction is captured by the cost tuple of labels *a*, *c* and *e*, giving them a higher priority over other non-live labels. Based on the priority of cost tuples, one of the three labels will be added to the live set; let us assume that LARS picks *c* to be added to the live set. Once *c* is live, instruction 2 can become fireable if *a* is made live next. When recomputing the cost tuple for the non-live labels, the *firing potential* of *a* is accounted for in its cost tuple, giving it higher priority over other non-live labels. Therefore *a* is made live next, and instruction 2 is fired (Figure 5.3c).

Once the computation reaches the state of Figure 5.3d, making label q live will enable firing of instruction 8. Instruction 8 has the last uses of labels f and e: once fired, the pseudo-register assigned to these two labels can be reused to store other labels. This *release potential* of q is accounted for in its cost tuple, resulting in it getting a higher priority over other non-live labels, and becoming live in the next step (Figure 5.3e).

Once an instruction is fired, its dependence edges are removed from DG, which can possibly unblock some instructions. The cost tuple for each non-live label is recomputed with a change in the set of live labels, or the set of unblocked statements. The algorithm terminates when all the instructions in the initial schedule have been fired. The final reordered schedule is then printed out using appropriate intrinsics for multi-core CPUs. Algorithm 5.1 sketches out the high level reordering algorithm.

5.3.4 Adding Instructions to Reordered Schedule

At any step, if there are more than one fireable instructions, then priority is given to one that has minimum *interlocks*[8] with the recently scheduled instructions in the final reordered schedule. For example, at the stage of Figure 5.3e, if labels *t1* and g are made live, then instructions 5 and 6 become fireable simultaneously. However, the previously fired instruction 8 interlocks with instruction 6 on label k. Therefore, instruction 5 is scheduled before instruction 6 in the reordered schedule (Figure 5.4a). If all the fireable instructions have the same degree of interlock with the instructions in the reordered schedule, then we fire them in order of their appearance in the

[8]an instruction I_a interlocks with a previously fired instruction I_b if there is a true dependence from I_b to I_a, or both I_b and I_a contribute to the same accumulator [33].

Algorithm 5.1: Reorder (P, k): Reorder the input schedule P using the greedy list-based scheduling of LARS

```
Input  : $P$: Initial schedule, $k$: Interlock window size
Output: $R$: Reordered schedule
1  $DG \leftarrow$ dependence graph for $P$;
2  $L \leftarrow \varnothing$; // set of live labels
3  $S_{ub} \leftarrow$ instructions that have no incoming dependences in DG;
4  $S_{fire} \leftarrow \varnothing$; // set of fireable instructions
5  compute $EST(n_i)$ for each instruction $n_i$ using equation 5.1;
6  for each label $l$ in $P$ do
7      compute initial priority of $l$ using $EST$ of instructions (Sec. 5.3.5)
8  while not all instructions in $P$ are fired do
9      $M \leftarrow \varnothing$; // list of cost tuple for each non-live label
10     for each label $l \notin L$ such that $l$ occurs in an instruction in $S_{ub}$ do
11         $M \leftarrow M \cup$ Create-Tuple $(l, L, S_{ub})$; (Algo. 5.2)
12     Adaptive-Custom-Sort $(M)$ (Sec. 5.3.6);
13     add the most profitable label in sorted $M$ to $L$;
14     for each instruction $s$ in $S_{ub}$ do
15         if all the labels of $s$ are in $L$ then
16             $S_{fire} \leftarrow S_{fire} \cup s$;
17             $S_{ub} \leftarrow S_{ub} \setminus s$;
18     while $S_{fire} \neq \varnothing$ do
19         remove an instruction $s \in S_{fire}$ that has the least interlocks with the
             instructions in $R$;
20         for each label $l$ in $s$ do
21             if this is the last use of $l$, remove $l$ from $L$;
22         if $s$ contributes to an accumulation node $t$ then
23             create a shadow accumulation output if any of the last $k$
                 instructions in $R$ also contribute to $t$;
24         append $s$ to $R$;
25         remove the dependence edges from $s$ in $DG$, add instructions with no
             incoming dependences in $DG$ to $S_{ub}$;
```

original schedule. This provides the scheduling phase of the backend compiler with opportunities to exploit ILP [33].

If instructions I_a and I_b interlock due to the same accumulation output, we can resolve the interlock by register renaming [96], a technique often used by both compilers and the hardware to remove false dependences and improve ILP. When firing the instruction I_a, we check if it interlocks due to the accumulation output with any of the k previously fired instructions, where k is a code generator parameter, which we term *interlock window size*. If it does, then we replace the accumulator in I_a with its *shadow*, which will act as a partial accumulator. The shadow accumulator is appropriately initialized to the identity of the accumulation operator (e.g., 0 for addition/subtraction, 1 for multiplication), and its will be added to the actual accumulator before the subsequent use of the accumulation output. There will be at most k shadows per accumulator, after which they will be cyclically reused. By register renaming, we remove the dependence between instructions that are less that k hops apart in the final schedule, thereby improving ILP. For example, in the reordered schedule of Figure 5.4a, the true dependences due to the accumulation output a in instructions 1–4 limits the achieved ILP. We can increase the ILP by creating a shadow accumulator *a1*, and collecting partial accumulations from alternate instructions into it, as shown in Figure 5.4b. The partial contribution from the shadow *a1* is added to a before its subsequent use in instruction 6.

5.3.5 Adding Labels to Live Set

Assigning initial priority to labels

None of the labels are initially live. To start the reordering, we have to choose the first non-live *seed label*, and make it live. However, this choice is not arbitrary. In order to determine the seed label, we assign an initial priority to all the labels. This initial priority based on the concept of *earliest starting time*. For each instruction n

```
1. a = b / c;
2. a += d * c;
3. a += b * e;
4. a += d / f;
5. q = f + e;
6. t1 = a * g;
7. t1 += k * g;
8. r = q * p;
9. m = n * p;
10. t2 = m * r;
```

(a) Final Schedule

```
1. a = b / c;
2. a1 += d * c;
3. a += b * e;
4. a1 += d / f;
5. q = f + e;
6. t1 = (a + a1) * g;
7. t1 += k * g;
8. r = q * p;
9. m = n * p;
10. t2 = m * r;
```

(b) Final schedule with shadows

Figure 5.4: Example: Using shadows to avoid interlocks

in the initial schedule, the earliest starting time, $EST(n)$, can be computed based on the pipeline latencies of the architecture [92].

$$EST(n) = \max_i(EST(p_i) + latency(p_i)) \tag{5.1}$$

where p_i is the i^{th} predecessor of n in DG, and $latency(p_i)$ is the latency of the dependence edge from p_i to n.

A source in the DG is a node with no predecessors, and a sink is a node with no successors. For a path p in DG starting at source node n_s and ending at sink node n_t, we define the path execution time, $PET(p) = EST(n_t)$. A *critical path* in DG corresponding to V is defined as the source-to-sink path with the highest PET. There can be more than one critical paths in the program. In most instruction scheduling algorithms, the first scheduled instruction (*seed instruction*) is the source instruction in the critical path [92, 33].

If an instruction n occurs in k source-to-sink paths in the DG, then we can compute a cumulative cost for n, similar to the one defined in [33], as $\max_{1 \leq r \leq k}(PET(p_r) - EST(n))$.

Thus, the source in the critical path will have the highest cumulative cost, and the instructions closer to the source instruction in relatively longer paths in the DG will have higher cumulative costs than other instructions. The labels occurring in an instruction with higher cumulative cost must be assigned a higher priority, so that such instructions are scheduled with urgency. The initial priority P_l of a label l is therefore set to $\max_i\{\mathit{cumulative\ cost}(n_i)\}$, where n_i represents all the instructions in which the label l appears.

Creating a cost tuple for each label

In order to model the profitability of making a non-live label l live, we capture its impact on the current set of unblocked instructions and its interaction with the live labels $\in L$ using the following metrics:

– Fire potential (F_{pot}): It indicates the number of instructions that become fireable if label l is made live. If there are k unblocked instructions that only need l to become live in order to become fireable, then the fire potential of l is k.

– Release potential (R_{pot}): Once the label l is made live, some instructions may become fireable. The release potential of l refers to the number of labels that have their last uses in the instructions that become fireable when l is made live.

– Cumulative Primary affinity (P_{aff}): A non-live label l has primary affinity of strength p with an already live label t if both l and t occur simultaneously as the labels in exactly p instructions. The cumulative primary affinity of a non-live label l is the sum of its primary affinity with all live labels.

– Cumulative Secondary affinity (S_{aff}): A non-live label l has secondary affinity of strength s with an already live label t if they both have a non-zero primary

Algorithm 5.2: Create-Label-Cost-Tuple (l, L, S_{ub}): Create a cost tuple for each non-live label that captures its interaction with the live labels and unblocked instructions

```
Input  : $L$: Set of live labels, $l$: a label $\notin L$, $S_{ub}$: Unblocked instructions
Output: $T$: Cost tuple for $l$
1  $R_{pot} \leftarrow 0$; $F_{pot} \leftarrow 0$; $N_{lk} \leftarrow 0$;
2  for each $s \in S_{ub}$ that becomes fireable when $l \in L$ do
3      increment $F_{pot}$;
4  for each $t$ in $L$ that has the last use in $s$ do
5      increment $R_{pot}$;
6  $P_l \leftarrow$ the label priority of $l$ based on its initial priority, and the current leading
     statement;
7  $N_{npaff} \leftarrow$ number of labels $\notin L$ that have non-zero primary affinity with $l$;
8  $P_{aff} \leftarrow$ cumulative primary affinity of $l$ to the labels $\in L$;
9  $S_{aff} \leftarrow$ cumulative secondary affinity of $l$ to the labels $\in L$;
10 $T \leftarrow \langle R_{pot}, F_{pot}, P_{aff}, S_{aff}, -N_{npaff}, P_l \rangle$;
11 return $T$;
```

affinity with exactly s common labels. The cumulative secondary affinity of a non-live label l is the sum of its secondary affinity with all live labels.

Rationale: The release potential of a label reflects its ability to reduce register pressure directly. Therefore, the label with highest release potential must always be made live before others. Making a label with high fire potential live can provide the compiler with independent instructions to schedule. Also, even if the fired instructions do not release registers, they may still be a step in decreasing the remaining uses of their operands, and towards a consequent release of the registers occupied by their operands. If the label l has a non-zero primary affinity with an already-live label t, then the live range of l and t definitely interfere, and t cannot be released till l becomes live. Therefore, if label l has a high cumulative primary affinity with the already-live labels, then we eagerly make l live, so that we get a step closer to releasing

l, and the labels it has primary affinity with. Similarly, if a label l has a non-zero secondary affinity with an already-live label t due to a label m, then the live range of both l and t will interfere with m. Making l live will shrink the live range of m and bring us one step closer to releasing m.

To help us determine the seed label, we had initialized the initial priority P_l to each label based on the cumulative cost of the instructions. Multiple labels along disjoint critical paths could be initialized with the same priority if those paths have the same *PET*. However, when an instruction n is fired by LARS, the priority of the labels in the instructions dependent on n must be increased to favor depth-first traversal of the CDAG containing n. Otherwise, the instructions along different CDAGs may get unnecessarily interleaved, increasing the register pressure [93]. Whenever an instruction is fired, the priority of each label is updated based on the current *leading statement*. A statement S_i is currently leading if maximum number of the instructions that it was lowered down to have been fired. All the non-live labels occurring in the unblocked instructions corresponding to S_i are assigned a higher P_l than other non-live labels.

The primary affinity of a non-live label l to other non-live labels is also an important metric. Suppose the label l has a non-zero primary affinity with r non-live labels. Then, if l is made live at the current step, it needs to be live until all the r labels are live as well. For each label l, $N_{npaff}(l)$ denotes the number of non-live labels with which l has non-zero primary affinity. A label with higher N_{npaff} must be penalized, as making it live may increase live-range interference in future.

The resultant combined metric for each label can be thought of as an 6-tuple: $\langle R_{pot}, F_{pot}, P_{aff}, S_{aff}, P_l, -N_{npaff}\rangle$. The collection of tuples for all the labels is then lexicographically sorted in descending order to rank-order the labels based on their profitability to be made live. The label that is deemed most profitable after the rank-ordering is chosen to be added to the set L. Algorithm 5.2 depicts the creation of the cost tuple.

5.3.6 Adaptivity in LARS

The relative order of the metrics in the tuple affects the reordering objective. For example, assigning the highest priority to R_{pot} followed by P_l would allow an interleaved execution of two statements only when the interleaving results in release of registers. This provides us the flexibility to define multiple sort functions, each acting as an objective function to determine the sequence in which the labels are added to the set L. In the implementation, we choose the objective function based on the nature of the computation. For the computations where the intra-statement reuse is less than half the inter-statement reuse (e.g., computation of Figure 5.5e), we order the metrics as $\langle R_{pot}, F_{pot}, P_{aff}, S_{aff}, -N_{npaff}, P_l\rangle$. This facilitates interleaving across the statements to reduce register pressure. However, if the inter-statement reuse is less than one-third the intra-statement reuse, we order the metrics as $\langle R_{pot}, P_l, F_{pot}, P_{aff}, S_{aff}, -N_{npaff}\rangle$, so that the interleaving across CDAGs is minimized.

5.3.7 Putting it all together

Algorithm 5.3 recapitulates the broad steps applied by LARS to reorder an input program M. The unrolling factors are applied to M to obtain an unrolled version M'. The dependence graph is then computed for M' (line 4) using which, multiple permutations $\{V\}$ of M' are created (line 5). The reordering heuristic is then applied

Algorithm 5.3: End-to-end algorithm of LARS

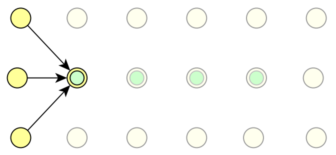
```
Input  : $M$: Input program, $uf$: Unrolling factors
Output: $R$: Reordered output
1  $T_m \leftarrow \infty$;
2  $k \leftarrow$ user-specified interlock window size;
3  $M' \leftarrow$ Unroll $(M, uf)$;
4  $G' \leftarrow$ Dependence-Graph $(M')$;
5  $\{V\} \leftarrow$ Create-Permutations $(M', G')$; (Section 5.3.2)
6  for $P \in \{V\}$ do
7      $r \leftarrow$ Reorder $(P, k)$;
8      $t \leftarrow$ execution time of $r$;
9      $T_m \leftarrow$ min $(T_m, t)$;
10     $R \leftarrow$ faster version between $r$ and the one corresponding to $T_m$;
11 return $R$;
```

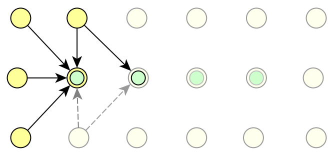
to each of these versions (line 7). The end goal is to find the fastest amongst all the versions, which is then returned as the final reordered code (lines 8–11).

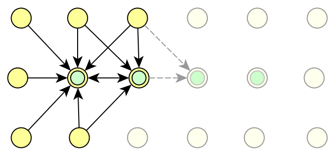
5.3.8 Example: LARS in action

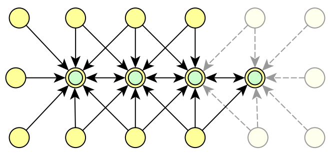
We demonstrate the power of LARS by using it to reorder a Jacobi stencil computation [99]. We assume that the interlock window size is 1. The computation has only inter-statement reuse, and the cost-tuple for labels are sorted as described in Section 5.3.6.

Figure 5.5e shows a 4-way unrolled version of the 2D 9-point Jacobi stencil. We assume that the coefficients in the stencil are literals, and all the input contributions to outputs are additive. There are 4 expression trees, each corresponding to the computation of a single output point, and there is significant reuse between these trees: two consecutive output points reuse 6 input values, and the output points two hops away reuse 3 input values. For the ease of description, let us assume that each input (output) point in Figure 5.5e is assigned an integer identifier which starts from


(a) Step 1


(b) Step 2


(c) Step 3


(d) Step 4

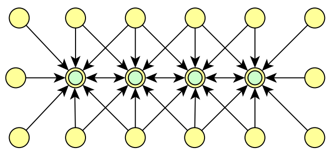

(e) Step 5

Figure 5.5: LARS in action: steps in instruction reordering with LARS for a 4-way unrolled 2D 9-point Jacobi stencil

1, and is incremented during the lexicographical scan, going from left-to-right, top-to-bottom. We identify an input (output) label as i_v (o_v), where v is its identifier. All the labels have the same initial priority.

LARS starts by scheduling the solo contributions from the inputs i_1, i_7 and i_{13} that contribute to o_1, since these labels can be immediately released (Figure 5.5a). Next, i_2 is made live, since it has firing potential of 1, and has the lowest primary affinity to other non-live labels. Now, making o_2 live will release i_2 (Figure 5.5b). i_{14} is made live next, as it will be released after making its contributions to o_1 and o_2. i_8 will be made live next, forcing o_3 to become live in the next step. At this point, we make i_3 live, and it contributes to three already-live output labels (Figure 5.5c).

LARS follows this pattern to progress to the schedule shown in Figure 5.5d, and ultimately that of 5.5e. At each step, we had at most three outputs points, that had high secondary affinity to each other, live consecutively. Also, we made exactly one input point live, which contributed to all the live output points, and was immediately released. Therefore, irrespective of the degree of unrolling, the maximum register pressure for the schedule by LARS will be 4.

Obtaining such a schedule is difficult with the existing instruction schedulers. The existing schedulers may interleave the computation across trees to increase ILP. However, among multiple interleavings with the same degree of ILP, there will be only a few that will simultaneously reduce the register pressure. Finding such interleavings requires more than just a greedy decision based on a local perspective of register pressure. LARS is able to find such an interleaving by having a broader perspective of register pressure based on release/fire potential, and cumulative primary/secondary affinities.

5.4 Experimental Evaluation

Experimental Setup The experimental results presented here were obtained on Intel Xeon Phi, and a skylake i7-6770K processor. The hardware details are listed in Table 5.1. All the benchmark codes are compiled with GCC-7.2.0, and the version of LLVM from the svn (llvm/trunk 311831). The compilation flags for both the compilers are listed in Table 5.2. Both GCC and LLVM provide some fine-grain control over the instruction scheduling passes via the compilation flags: GCC allows the user to enable or disable the prepass and postpass instruction scheduler, whereas LLVM provides multiple implementations for the prepass scheduler. With the fine-grain access to the compiler passes, we can control the effect of the scheduling passes on the LARS reordered schedule.

Resource	Details
Intel multi-core CPU	Intel core i7-6700K (4 cores, 2 threads/core 4.00 GHz, 8192K L3 cache)
Intel Xeon phi	Intel Xeon Phi 7250 (68 cores, 4 threads/core 1.40GHz, 1024K L2 cache)

Table 5.1: Benchmarking hardware to evaluate LARS

Compiler	Flags
gcc	-Ofast/Os -fopenmp -ffast-math -fstrict-aliasing -march=core-avx2/{knl -mavx512f -mavx512er -mavx512cd -mavx512pf} -mfma -f(no)schedule-insns -fschedule-insns2 -fsched-pressure -ffp-contract=fast
llvm	-Ofast/Os -fopenmp=libomp -mfma -ffast-math -fstrict-aliasing -march=core-avx2/knl -mllvm -ffp-contract=fast -pre-RA-sched=”source/list-ilp/list-hybrid/list-burr”

Flags for multi-core CPU and Xeon Phi

Table 5.2: Compilation flags for GCC and LLVM on different benchmarking hardwares

Benchmarks We evaluate the efficacy of LARS on a wide variety of benchmarks, that can be grouped into five sets: the first set comprises generalized versions of computations typically used in iterative processes such as solving partial differential equations and convolutions [67]; the second set comprises smoothers used in HPGMG benchmark suite [45]; the third set comprises high-order stencil computations from the ExpCNS Compressible Navier-Stokes mini application from DoE [29], and the Geodynamics Seismic Wave SW4 application code [104]; the fourth set comprises the kernels from Cloverleaf benchmark suite [62]; the fifth set comprises the tensor contraction kernels CCSD(T) from the NWChem suite [77]. These benchmarks are listed in Table 5.3. All the benchmarks are double-precision. Note that we evaluate all

Benchmark	N	FPP	R	Benchmark	N	FPP	R
2d25pt	8192^2	49	2	cell-advec 3D	256^3	49	16
2d49pt	8192^2	97	2	mom-advec 3D	256^3	55	15
2d64pt	8192^2	127	2	acceleration 3D	256^3	57	13
2d81pt	8192^2	161	2	ideal-gas 3D	256^3	12	4
2d121pt	8192^2	241	2	PdV 3D	256^3	98	16
3d27pt	512^3	53	2	calc-dt 3D	256^3	67	11
3d125pt	512^3	249	2	fluxes 3D	256^3	30	12
chebyshev	512^3	39	6	viscosity 3D	256^3	139	9
7-point	512^3	11	2	cell-advec 2D	4096^2	47	14
poisson	512^3	21	2	mom-advec 2D	4096^2	41	13
helmholtz-v2	512^3	22	7	acceleration 2D	4096^2	38	10
helmholtz-v4	512^3	115	7	ideal-gas 2D	4096^2	12	4
27-point	512^3	30	2	PdV 2D	4096^2	51	13
hypterm	300^3	358	13	calc-dt 2D	4096^2	36	9
diffterm	300^3	415	11	fluxes 2D	4096^2	12	8
rhs4th3fort	300^3	687	11	viscosity 2D	4096^2	58	7
derivative	300^3	486	10	sd-t-d1-{1:9}	24^8	2	3

N: Domain Size, FPP: FLOPs per Point R: Arrays Accessed

Table 5.3: Characteristics of straight-line scientific benchmarks

the compute kernels from Cloverleaf benchmark suite, leaving only the trivial kernels that copy data from one array to another.

Code Generation The original version (*Original*) for each benchmark is as written by application developers, parallelized and vectorized by adding appropriate OpenMP pragma to the outermost parallel loop, and SIMD pragma to the innermost vectorizable loop, but without any explicit loop unrolling. We generate multiple unrolled versions (*Unrolled*) for each benchmark by explicitly unrolling all but the innermost loop by powers of 2, restricting the maximum unroll factor to 8. For each unrolled version, we generate an accumulation version with all the additive contributions converted to accumulations (*Accumulation*). Corresponding to each unrolled version, we create a LARS-optimized version (*Reordered*) as well. All the benchmarks evaluated are control flow-free and conform to the restrictions described in Section 5.3.1, and hence LARS is able to parse them without any code modification. Except for the CCSD(T) kernels, LARS generates a reordered schedule with appropriate SIMD intrinsics for multi-core CPU and Xeon Phi. We do not generate accumulation version for CCSD(T) benchmarks, since their unrolled versions are already in accumulation form. No other optimization (e.g. tiling) is applied to the reordered code to get a fair performance comparison with respect to the unrolled versions. For code generation, the interlock window size, k, is set to 2. For all the benchmarks, the reordered versions were generated under 15 seconds by LARS. We check the correctness of each reordered code against the original version.

Performance Results Figures 5.6 – 5.11 plot the performance of the original, unrolled, accumulation, and reordered code for the different benchmark sets. For the convolution kernels (Figure 5.6), the reordered version significantly outperforms the original and unrolled versions over all compilers/architectures, especially when the *order* of the computation increases. The order here refers to the extent of elements read from the center. For example, the *2d21pt* kernel benefits the most from reordering, and it has the highest order, 5. This observation is consistent with that of Stock et al. [99]. The volume of data read per point usually increases with an increase in the order for most of the dense computations, and this can exacerbate the spills in the unrolled code. LARS exploits associativity to judiciously reorder the instructions by interleaving computations across the unrolled statements, thereby reducing the spill volume. The reordered version also outperforms the accumulation version, indicating

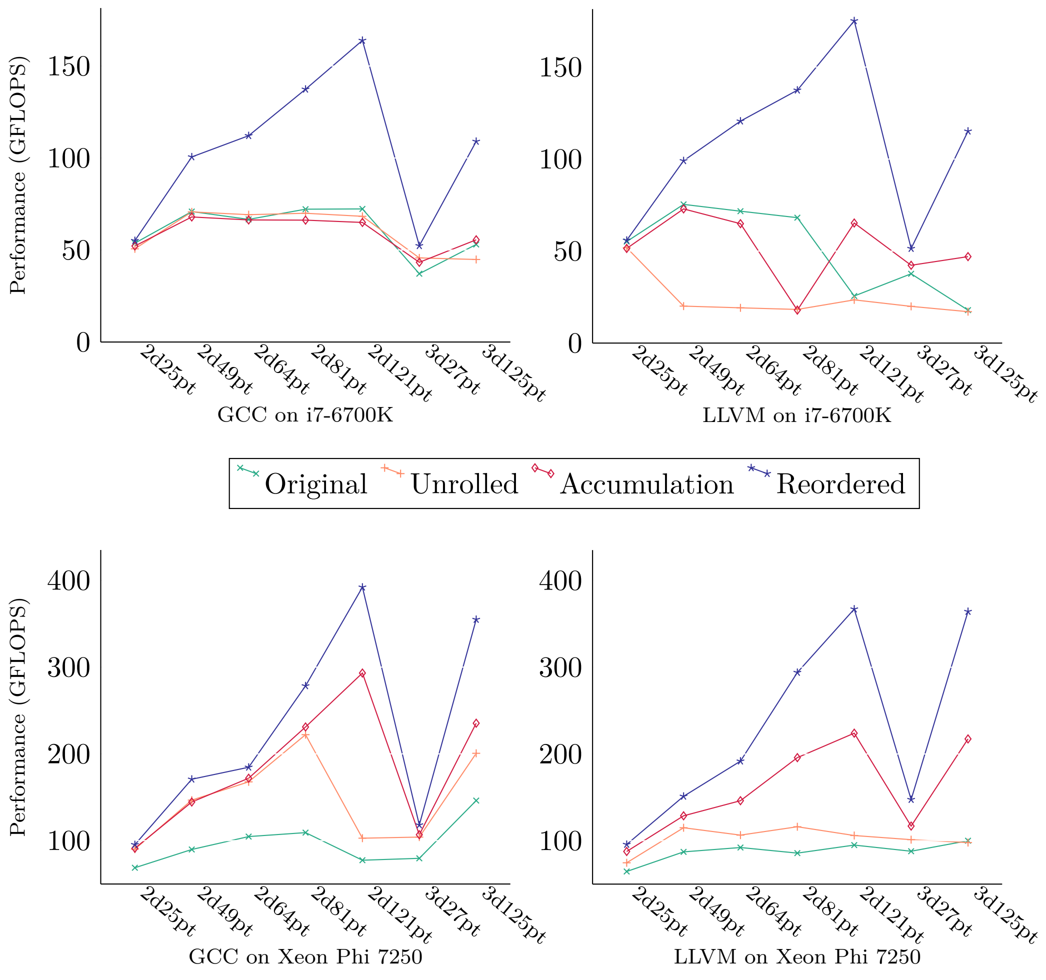


Figure 5.6: Performance of convolution kernels on different architectures

that reordering the instructions after rewriting the computation in accumulation form is crucial to performance.

LARS outperforms both original and unrolled versions for the HPGMG smoothers (Figure 5.7) as well, but the performance gains are lower when compared to the convolution kernels due to two reasons: (a) LARS leverages operator distributivity in order to exploit associativity for smoothers, thereby increasing the computation; (b) the increase in arithmetic intensity (defined as the FLOPs relative to the memory

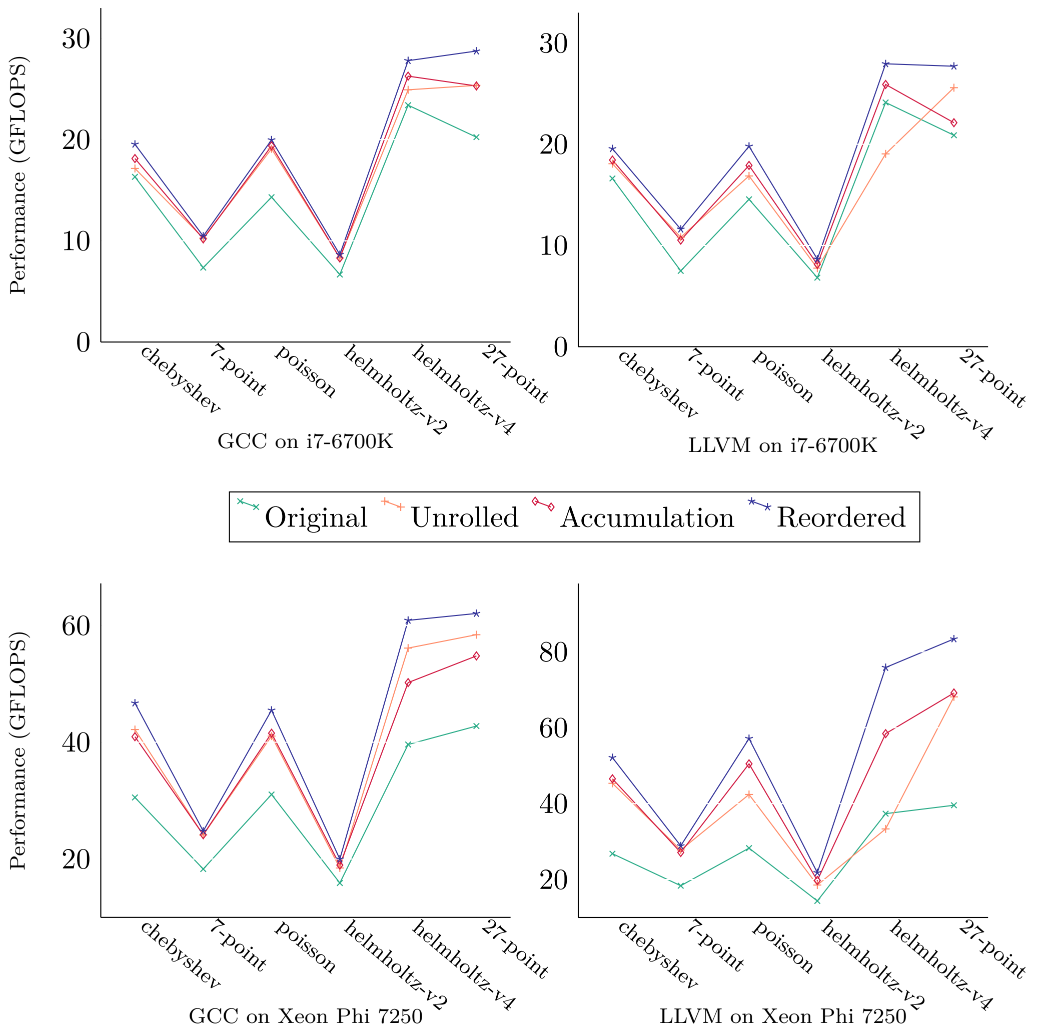


Figure 5.7: Performance of smoother kernels on different architectures

accesses) with unrolling is less significant for smoothers, which implies that they are more bandwidth-bound than convolutions.

For Cloverleaf 3D and 2D benchmarks, the performance of LARS-reordered code is almost the same (1.0× – 1.13× for multi-core CPU), or slightly better (1.14× – 1.7× with GCC on Xeon Phi). This can be mainly attributed to the nature of the computations in the Cloverleaf benchmark suite. As observed from Table 5.3, each

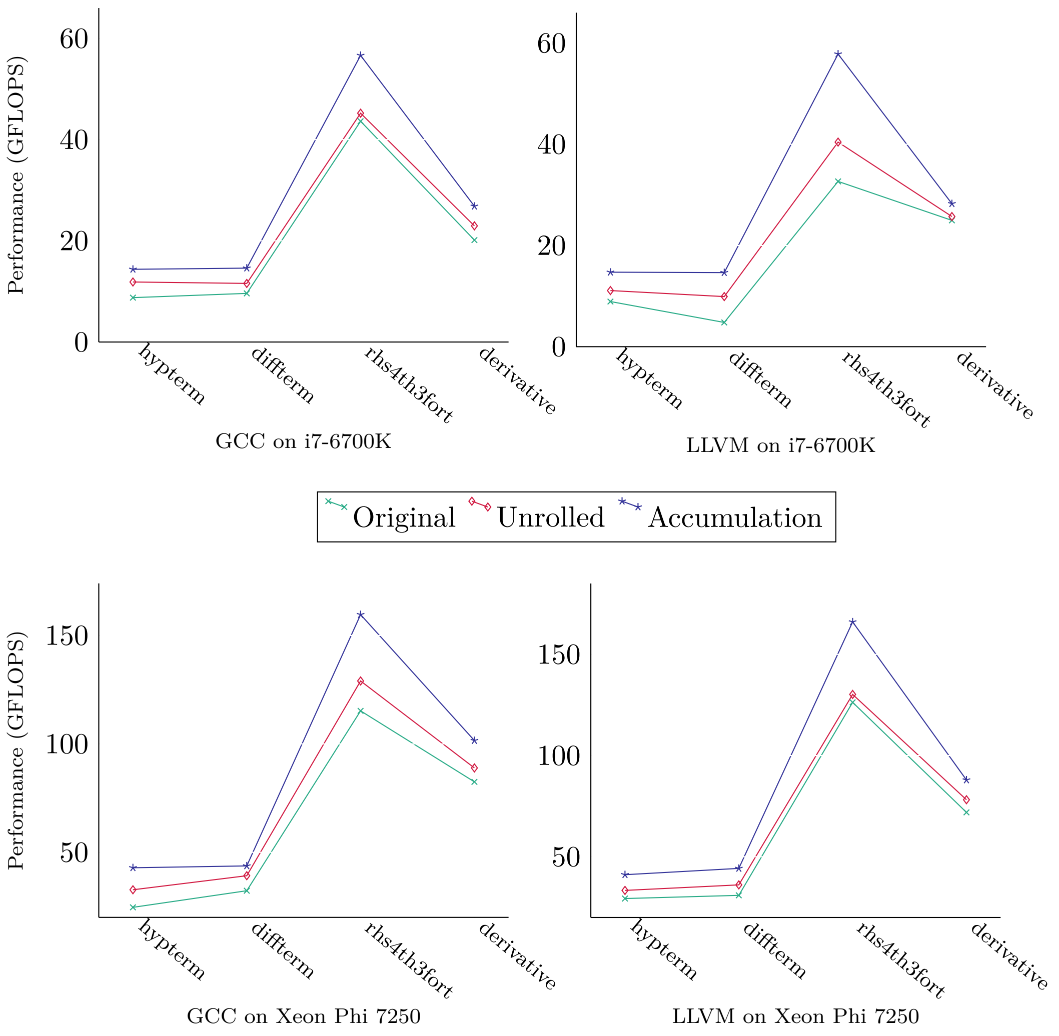


Figure 5.8: Performance of high-order stencils on different architectures

kernel in the Cloverleaf benchmark reads from a multitude of arrays, but the computations in each kernel is scarce. Thus, there very little reuse to be exploited via unrolling, and the computation is severely bandwidth-bound. Both these factors reduce the benefits of unrolling and reordering. However, the performance results serve to demonstrate that the reordering done by LARS does not degrade the performance for such benchmarks.

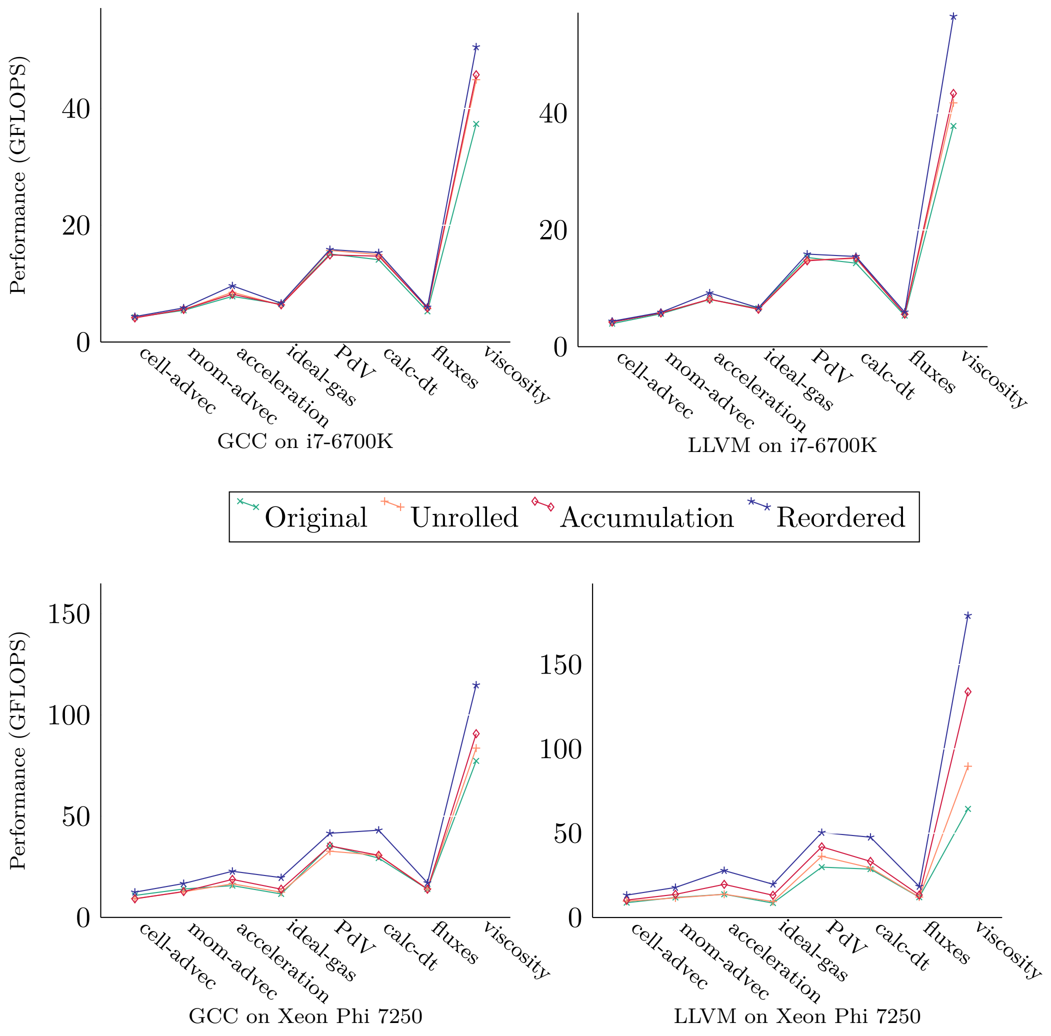


Figure 5.9: Performance of Cloverleaf 3D kernels on different architectures

High-order stencils are slowly becoming commonplace in scientific simulations, and optimizing them is the current focus of the HPC community. These high-order stencils can be thought of as a forest of CDAGs, with high input data reuse across the CDAGs. This abstraction is particularly challenging for a traditional compiler to optimize, since most of the integrated instruction schedulers are designed to schedule a single CDAG. With LARS, we were able to reduce the total memory accesses per

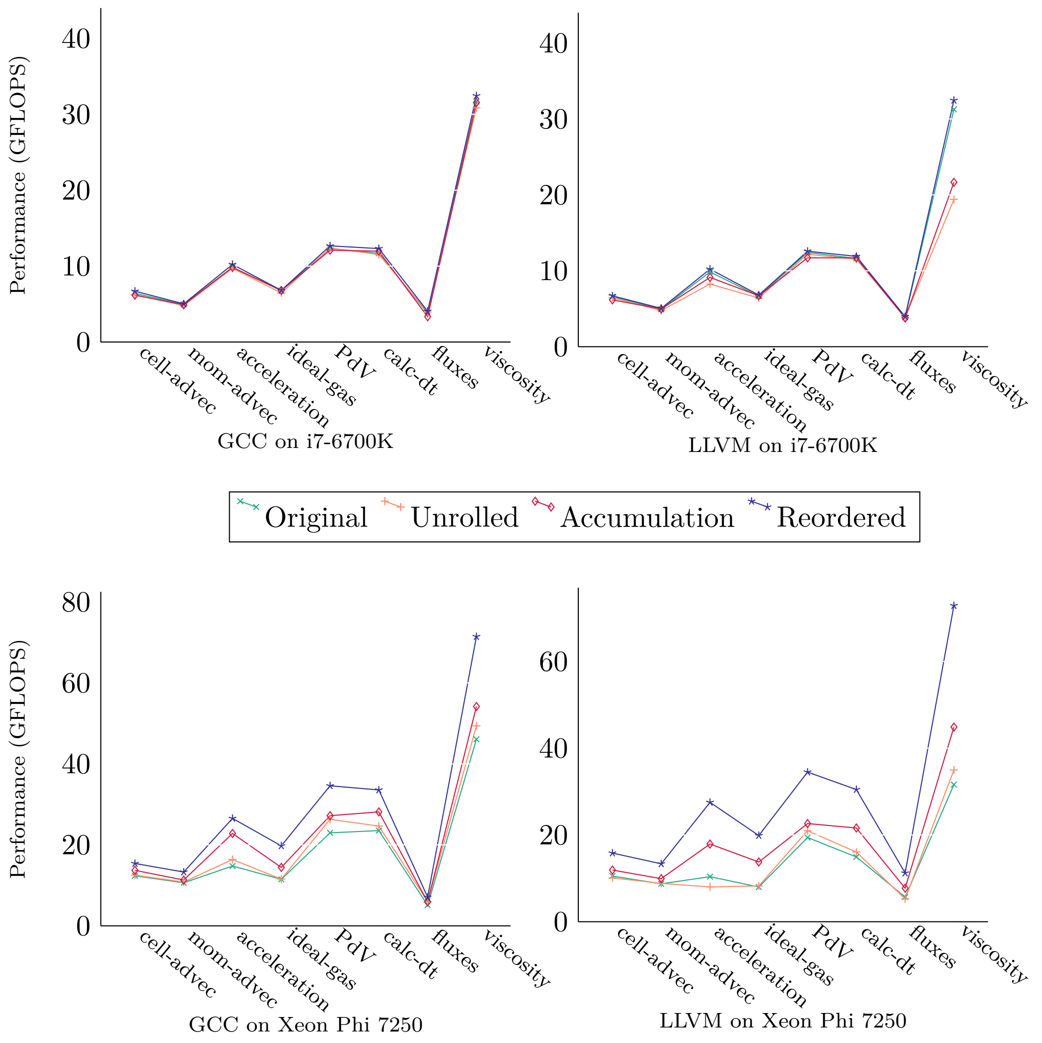


Figure 5.10: Performance of Cloverleaf 2D kernels on different architectures

point even with the non-unrolled version, and achieve a $1.22\times - 3\times$ speedup over the base version (figure 5.8).

A similar performance trend is observed for CCSD(T) tensor contractions. Since each contraction has a loop nest of depth 7 and loop unrolling in tensor contractions is a research problem in itself [61], we fix two loops as the unrolling candidates. Since the computation is an accumulation, the unrolling candidates are chosen to

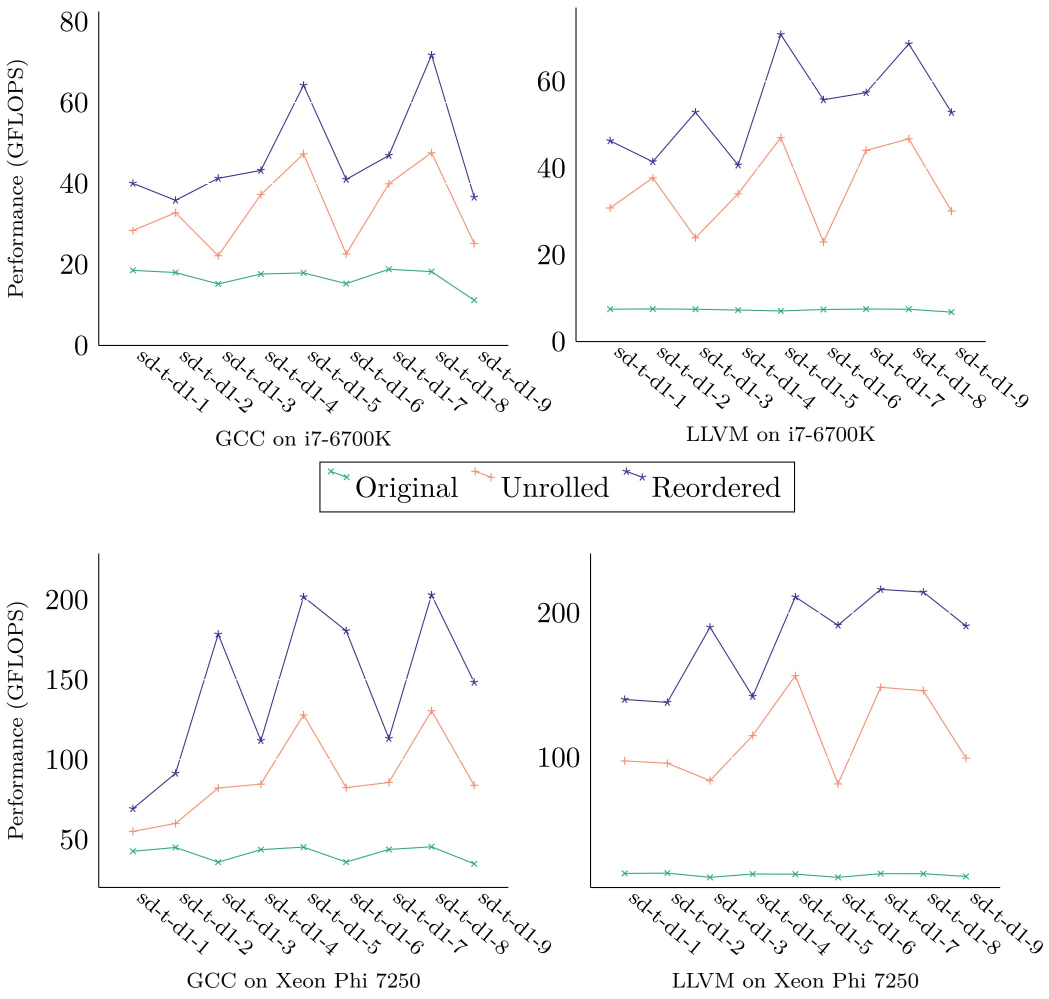


Figure 5.11: Performance of CCSD(T) kernels on different architectures

favor the reuse of the output element. From Figure 5.11, we observe that unrolling greatly improves the performance, and we get a 1.09× – 2.26× speedup with simple reordering of the instructions to reduce the memory accesses.

We also perform an analysis of the generated assembly code for the various versions, and count the number memory accesses via loads into the *ymm/zmm* registers in the computational loop. The numbers for Xeon Phi 7250 are presented in Table 5.4. One can observe from the data that the reordered version incurs far lesser memory accesses than the unrolled version, indicating that the reordering with LARS reduces the number of spills and reloads.

5.5 Summary

Register spills and low performance are a problem when compiling scientific codes with a high degree of may-to-many reuse. This chapter presents LARS, a list-based, adaptive, register pressure driven source-level scheduler, that leverages operator associativity to reorder instructions to reduce register pressure. The metrics used by LARS are powerful enough to exploit patterns in the computation, and general enough to benefit a variety of input applications. We demonstrate the usability of LARS, and the effectiveness of its reordering heuristics over several benchmarks using multiple compilers and architectures.

Bench.	GCC					LLVM				
	Orig.	Unrolled		LARS		Orig.	Unrolled		LARS	
	A	*A*	*UF*	*A*	*UF*	*A*	*A*	*UF*	*A*	*UF*
2d25pt	55	77	{2,1}	58	{2,1}	67	207	{4,1}	70	{4,1}
2d49pt	141	344	{4,1}	144	{4,1}	153	399	{4,1}	138	{4,1}
2d64pt	201	494	{4,1}	174	{4,1}	213	519	{4,1}	165	{4,1}
2d81pt	269	664	{4,1}	206	{4,1}	281	655	{4,1}	194	{4,1}
2d121pt	429	1064	{4,1}	294	{4,1}	286	2056	{8,1}	763	{8,1}
3d27pt	65	143	{8,1,1}	93	{8,1,1}	71	223	{4,1,1}	88	{4,1,1}
3d125pt	468	1133	{2,2,1}	361	{2,2,1}	277	1019	{4,1,1}	497	{4,1,1}
chebyshev	42	131	{2,2,1}	73	{2,2,1}	50	297	{4,2,1}	127	{4,2,1}
7-point	15	67	{8,1,1}	62	{8,1,1}	26	34	{4,1,1}	32	{4,1,1}
poisson	27	43	{2,1,1}	35	{2,1,1}	36	166	{8,1,1}	102	{8,1,1}
helmholtz-v2	21	62	{4,1,1}	60	{4,1,1}	33	75	{4,1,1}	60	{4,1,1}
helmholtz-v4	74	151	{2,1,1}	111	{2,1,1}	76	266	{4,1,1}	233	{4,1,1}
27-point	36	228	{4,2,1}	99	{4,2,1}	48	131	{4,1,1}	67	{4,1,1}
hypterm	317	-	-	268	{1,1,1}	307	-	-	214	{1,1,1}
diffterm	345	-	-	328	{1,1,1}	440	-	-	386	{1,1,1}
rhs4th3fort	632	-	-	426	{1,1,1}	327	-	-	287	{1,1,1}
derivative	299	-	-	223	{1,1,1}	311	-	-	290	{1,1,1}
cell-advec 3D	49	92	{2,1,1}	84	{2,1,1}	50	94	{2,1,1}	79	{2,1,1}
mom-advec 3D	61	114	{2,1,1}	60	{1,1,1}	84	118	{2,1,1}	55	{1,1,1}
accelerate 3D	76	150	{2,1,1}	75	{1,1,1}	75	156	{2,1,1}	51	{1,1,1}
ideal-gas 3D	7	12	{2,1,1}	5	{1,1,1}	6	13	{2,1,1}	6	{1,1,1}
PdV 3D	47	86	{2,1,1}	46	{1,1,1}	41	86	{2,1,1}	70	{2,1,1}
calc-dt 3D	44	82	{2,1,1}	71	{2,1,1}	42	77	{2,1,1}	58	{2,1,1}
fluxes 3D	33	120	{2,2,1}	95	{2,2,1}	32	120	{2,2,1}	92	{2,2,1}
viscosity 3D	47	84	{2,1,1}	80	{2,1,1}	50	100	{2,1,1}	80	{2,1,1}
cell-advec 2D	76	143	{2,1}	132	{2,1}	72	139	{2,1}	127	{2,1}
mom-advec 2D	47	164	{4,1}	141	{4,1}	24	163	{4,1}	85	{4,1}
accelerate 2D	42	162	{4,1}	99	{4,1}	40	170	{4,1}	97	{4,1}
ideal-gas 2D	7	12	{2,1}	7	{1,1}	6	13	{2,1}	6	{1,1}
PdV 2D	28	48	{2,1}	42	{2,1}	22	42	{2,1}	38	{2,1}
calc-dt 2D	23	42	{2,1}	39	{2,1}	24	43	{2,1}	34	{2,1}
fluxes 2D	16	50	{4,1}	44	{4,1}	14	50	{4,1}	43	{4,1}
viscosity 2D	23	42	{2,1}	38	{2,1}	28	47	{2,1}	36	{2,1}
sd-t-d1-1	13	76	{4,8}	48	{4,8}	13	76	{4,8}	50	{4,8}
sd-t-d1-2	13	76	{4,8}	48	{4,8}	13	76	{4,8}	48	{4,8}
sd-t-d1-3	15	76	{4,8}	50	{4,8}	16	69	{4,8}	57	{4,8}
sd-t-d1-4	13	82	{4,8}	54	{4,8}	13	76	{4,8}	50	{4,8}
sd-t-d1-5	13	126	{4,8}	93	{4,8}	13	104	{4,8}	62	{4,8}
sd-t-d1-6	15	141	{4,8}	72	{4,8}	16	106	{4,8}	57	{4,8}
sd-t-d1-7	13	82	{4,8}	60	{4,8}	13	76	{4,8}	48	{4,8}
sd-t-d1-8	13	126	{4,8}	93	{4,8}	13	104	{4,8}	73	{4,8}
sd-t-d1-9	12	121	{4,8}	97	{4,8}	18	126	{4,8}	71	{4,8}

A: total number of memory accesses, *UF*: unrolling factor along loop dimensions {k,j,i}

Table 5.4: Assembly code analysis for LARS on Xeon Phi 7250

CHAPTER 6

Related Work

Automatic high-performance GPU code generation for stencils has been a topic of active research for both CPUs [18, 72, 84, 106, 7, 71, 42, 125, 99, 8, 24, 32] and GPUs [113, 86, 36, 43, 37, 108, 64, 116, 9, 127]. The main issues discussed in this dissertation about developing high-performance code for stencil computations have also been addressed in similar ways in these efforts. In this section, we provide a brief discussion of the related work: Section 6.1 discusses the related work in context of Chapter 3, and Section 6.2 discusses the related work in context of Chapters 4 and 5.

6.1 Optimizations for bandwidth-bound stencils

Tiling Strategies Tiling is a well-known loop transformation which partitions the computation into smaller tiles that are executed atomically. Spatial and temporal tiling has been recognized as the key to realize high performance for bandwidth-bound stencils by many prior efforts [37, 43, 85, 36, 55, 106, 18, 125, 128]. Many prior efforts on optimizing stencils from the bandwidth perspective have proposed a range of tiling approaches: overlapped tiling [55, 85, 43, 86], split tiling [37, 42], hexagonal tiling [36], hierarchical overlapped tiling [128], diamond tiling [7, 12], and cache oblivious tiling [106, 101], all enabling concurrent execution of tiles, which is essential for coarse-grained parallelization of the computation. Of these, only overlapped tiling [43, 85], split tiling [37], and hexagonal tiling [36] have been used to enable concurrent execution of thread blocks on GPUs. From our experience, coalescence can pose a major challenge when reading the halo region from the boundary of neighboring tiles for split tiling. Overlapped tiling, on the other hand, avoids this problem with redundant initial loads and recomputations. Therefore, many manual [72, 63] and automated optimizing efforts [43, 85, 116] have used overlapped tiling for concurrent execution on GPUs.

Code Generators for GPUs PPCG [113] is a polyhedral source-to-source compiler that generates classically time tiled OpenCL and CUDA code from an annotated sequential program. Patus [18] is a code generation and auto-tuning framework for stencil computations that can generate spatially tiled CUDA code without shared memory usage from the input DSL stencil specification. Mint [108] is a pragma-based source-to-source translator implemented in the ROSE compiler [47] that generates a spatially tiled CUDA code from traditional C code. Unlike these approaches that generate code for a single GPU device, Physis [64] translates user-written structured grid code into CUDA+MPI code for GPU-equipped clusters. Zhang and Mueller [127] develop an auto-tuning strategy for 3D stencils on GPUs. Their framework uses thread coarsening along different dimensions with judicious use of shared memory and registers to improve performance of spatially-tiled 3D stencils.

Overtile [43] and Forma [85] are DSL compilers that generate a time-tiled CUDA code from an input stencil DSL specification. PolyMage [71] is a DSL based code generator for automatic optimization of image processing pipelines. All these approaches use overlapped tiling to fuse the stencil operators in the computation, with the intermediate arrays stored in shared memory. While PolyMage is restricted to 2D domains, the code generated for 3D stencils by both Overtile and Forma suffers from sub-optimal performance to the volume of redundant computations. Grosser et al. [37] implement the split tiling approach of [42], and hexagonal tiling [36] for temporal tiling in GPUs.

Code Generators for CPUs The Pochoir [106] stencil compiler uses a DSL approach to generate high-performance Cilk code that uses an efficient parallel cache-oblivious algorithm to exploit reuse. The Pochoir system provides a C++ template library that allows the stencil specification to be executed directly in C++ without the Pochoir compiler, which aids debugging. Henretty et al. [42] propose hybrid split-tiling and nested split-tiling to achieve parallelism without incurring redundant computations. Halide [84] is a DSL language for image processing pipelines. It decouples algorithm specification from its execution schedule. The advantage of this separation is that one can write multiple schedules for the same computation without rewriting the entire computation. Halide schedules can express loop nesting, parallelization, loop unrolling and vector instruction. The performance of the optimized code depends on the efficacy of the schedule. The schedule can be either written manually by a domain expert, or generated by extensive auto-tuning (e.g., with OpenTuner [4]); these approaches either require some degree of expertise with Halide DSL, or are time consuming. Recently, Mullapudi et al. [70] extended the scheduling strategy of PolyMage [71] to automatically generate schedules for Halide.

Bandishti et al. [7] propose diamond tiling to ensure concurrent start-up as well as perfect load-balance whenever possible. Yount et al. describe YASK [125] framework to simplify the task of defining stencil functions, generating high-performance code targeted for Intel Xeon Phi architecture. Chapter 3 presented experimental results comparing the performance of StencilGen with several of these stencil optimizers.

Manual Implementations Micikevicius [67] presents a CUDA implementation of 3D finite difference computation that performs spatial tiling, and uses registers to alleviate shared memory pressure. Nguyen et al. [72] extend this implementation to a 3.5D-blocking algorithm for 3D stencils: streaming along one dimension, and temporal tiling along the other two dimensions. For CPUs, their approach reduces the cache footprint, and for GPUs, it reduces the shared memory required to time-tile a class of cross stencils. Datta et al. [24] analyze the performance of a 7-point 3D stencil (as implemented by Nguyen et al. [72]) on an Nvidia GTX280. Maruyama and Aoki [63] present techniques to optimize a 3D 7-point Jacobi stencil on Fermi M2075 and Kepler K20X GPUs. They achieve best performance on K20X with a combination of temporal tiling, streaming, use of shared memory and registers for storage, and warp specialization [10]. Barring warp specialization, their strategy to achieve best performance is similar to 3.5D-blocking.

Performance Modeling of Stencils on GPUs Another category of research addresses the development of *analytical models* for the performance modeling of stencil computations on GPUs. These models are intended for integration into code generators to guide kernel optimizations. Hong and Kim [44] propose an analytical model to predict performance of GPU kernels. However, their model relies on manual analysis of low-level information about the kernel. Lai et al. [56] analyze a performance upper bound for SGEMM on GPUs, and use register blocking assuming maximum register reuse. Their model only provides performance upper-bounds for compute-bound kernels. Su et al. [102] propose a model that estimates the execution time based on the data traffic between different memory hierarchies. Prajapati et al. [82] propose an analytical model that predicts the execution time of the code generated with hexagonal tiling. A common challenge for all developed models to date is the precision in prediction, stemming from the complexity of the underlying hardware. Wahib and Maruyama [116] extend the analytical model of Lai et al. [56]. They pose kernel fusion as an optimization problem, and use a code-less performance model to choose a near-optimal fusion configuration amongst other possible variants. The space of feasible solutions is pruned using a search heuristic based on a hybrid grouping genetic algorithm. Gysi et al. [38] also propose a model-driven stencil optimization approach. Like Wahib and Maruyama [116], they use a code-less performance model to find the best fusion configuration among *all* valid topological sorts of the stencil DAG. The model has been used to guide kernel fusion in the Stella library [39].

6.2 Register-Level Optimizations

Register allocation has been extensively studied: from the seminal work of Chaitin [16] on using graph coloring, to the more recent SSA based schemes that exploit the possible decoupling of the allocation phase to the assignment phase [21, 68]. Many extensions/improvements have been applied [60, 97, 83], but almost all the existing works are restricted to the register allocation for fixed schedule. However, it is folk knowledge that improvements to register allocation alone does not bring much performance gain. The interplay of register allocation and instruction scheduling has been studied by a body of prior research [69, 80, 74, 11, 54, 33], and becomes quite important for architectures with ILP, since there is a tradeoff between increasing ILP and exposing locality. Hence, prior work on hyper/super-block list or modulo pre-pass scheduling [20, 120, 69] were extended to account for register pressure. Other works proposed a reverse scheme where register allocation was made sensitive to not change the minimum initiation interval for the scheduler to expose sufficient ILP [107]. Most of the current mainstream open-source compilers [98, 57] have adopted the first approach: when the register pressure is too high, the pre-pass list scheduling heuristics prioritize scheduling instructions that reduce the register pressure. However, such algorithms lack a global view, focusing only on the local register pressure at the current scheduled point. The associated optimization problem is NP-hard, and it is known that heuristics implemented in production compilers perform quite poorly on long straight line code [53], such as loop-body of highly unrolled loops.

This observation motivated developers of auto-tuned libraries [31, 34] to consider specific properties of the computational DAG to generate codelets that expose good locality for register reuse at source level: for certain domain-specific applications like FFT, a scheduling that minimizes the spill volume is well understood [49]. The problem addressed in Chapters 4 and 5 belongs to a similar category, where one has to optimize register reuse for long straight-line code arising from domain-specific kernels. The main difference is that for the computational DAGs considered by our framework, optimal or nearly-optimal scheduling is unknown. Our proposed heuristic is a solution to address the optimization of such DAGs. Prior work on code generation for expression trees [2, 3, 93] were discussed in details in Section 4.3. We briefly discuss selected works on combined register allocation and instruction scheduling, and register optimizations for high-order stencils.

Integrated Register Allocation and Instruction Scheduling Sethi et al. [93] propose an algorithm to translate an expression tree into machine code using optimal number of registers. The algorithm does not extend to CDAGs though. Goodman and Hsu [33] present a prepass scheduler that is register-pressure sensitive. Motwani et al. [69] show that integrated register allocation and instruction scheduling is NP-hard. They propose a combined heuristic that provides relative weights for controlling register pressure and instruction parallelism. For instructions where the register of an operand can be used for the result, Govindrajan et al. [35] try to generate an instruction sequence from data dependence graph that is optimal in register usage. Berson et al. [11] use register reuse DAGs to identify instructions whose parallel scheduling will require more resources than available, and optimize them to reduce their resource demands. Pinter [80] describes an algorithm that colors a parallel interference graph to obtain a register allocation that does not introduce false dependences, and therefore exploits maximal parallelism. Norris et al. [74] propose an algorithm that constructs an interference graph with all feasible schedules, and then removes interferences for schedules that are to be least likely followed. All these approaches are designed for in-order or VLIW processors, and an experimental evaluation by Valluri et al. [110] show that integrated efforts rarely benefit OoO processors.

While these efforts consider integrated register management and instruction scheduling, the goals are very different and the contexts quite dissimilar. Prior work in this category has focused on maximizing parallelism without significantly increasing the

MAXLIVE. In our context, the main reason for reordering instructions is to effectively exploit the significant potential for many reuses of values held in registers, while reducing the MAXLIVE.

In context of OoO processors, Barany and Krall [54] propose an optimistic integrated approach that performs prepass scheduling, and then rearranges instructions to mitigate register pressure during register allocation. However, the rescheduling is done from the local perspective of a single instruction, and tends to reduce ILP. Silvera et al. [94] propose an instruction reordering framework for dynamic issue hardware that takes the output schedule after prepass scheduling, and reorders the instructions within the instruction window to reduce register pressure. With such constraints on reordering, their method will not be able to do aggressive reordering that reduces register pressure while maintaining ILP, as done by LARS. While Barany and Krall [54] note that their rescheduling may not reduce register pressure as much as a register-pressure-driven scheduler, both they and Silvera et al. [94] emphasize the importance of reducing register spills in OoO processors. Domagala et al. [27] present a register allocation strategy that is cognizant of loop unrolling and instruction scheduling. However, it does not consider associative reordering to improve register allocation/instruction scheduling.

Register Pressure and Occupancy Trade-offs on GPUs The performance implications of the trade-off between registers per thread and the achievable occupancy is an under-studied topic. The work by Volkov and Demmel [115] was the first to disperse the myth that higher occupancy (and hence higher thread-level parallelism) translates to high performance; sometimes better performance can be achieved by increasing instruction-level parallelism at the cost of reduced occupancy. Since then, few research efforts have been directed towards finding the right balance of thread-level parallelism on GPUs [59, 123, 58, 41]. Xie et al. propose CRAT, a framework that coordinates register allocation and thread-level parallelism. It simulates register allocation on the PTX code for a few code versions with varying degree of thread-level parallelism using GPGPU-sim [6], and executes them on GPGPU-sim before reporting the best-performing version. Li et al. [58] propose a auto-tuning strategy that explores the performance at critical points in the occupancy chart of a Kepler GPU, thereby exploring the various trade-offs between register pressure and occupancy. Hayes et al. [41] propose Orion, a framework that auto-tunes for occupancy in a GPU application. It uses a performance model to navigate the occupancy space by either monotonically increasing or decreasing the occupancy the from the initial value.

Register Optimizations for High-Order Stencils Stock et al. [99] identify a performance issue with register reuse for iterative stencils by noting that even though the computation becomes less memory-bound with increase in the stencil order, their register pressure worsens. They use a generalized version of the semi-stencils algorithm by Cruz and Araya-Polo [25] to interleave the additive contributions in an unrolled computation. Their approach uses retiming to homogenize stencil accesses and reduce register pressure, whereas our approach presented in Chapters 4 and 5 is based on a very different computational abstraction, and is more broadly applicable. Basu et al. [8] propose a partial sum optimization implemented within the CHiLL compiler [40]. The partial sums are computed over planes for 3D stencils, and redundant computation is eliminated by performing array common subexpression elimination (CSE) [26]. However, this optimization is only applicable to stencils with constant and symmetrical coefficients. While their work does not claim to reduce register pressure, it may do so as a consequence of array CSE. Based on the concepts of rematerialization [14], Jin et al. [50] propose a code generation framework that trades off recomputations for reduction in register pressure. It uses dynamic programming to iteratively determine the minimum amount of recomputations required to reduce the register consumption by one. This approach is limited to the stencils described in [50], where the recomputation of an expression does not increase the live ranges of the values involved in it. In summary, existing approaches targeting register optimization for high-order stencils do not generalize well. Unlike these approaches, our frameworks described in Chapters 4 and 5 optimize both iterative and more general multi-statement stencils.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

GPUs provide massive parallelism, higher computing power, and higher bandwidth compared to the counterpart multi-core CPU architectures. Due to this, they have slowly emerged as a favored accelerator for many regular and irregular scientific computations, including stencils. However, accelerating stencil computations on GPU is challenging, and requires devising optimization strategies based on the nature of the computation and the resource constraints. With the increase in the complexity of the stencil and the optimizations required to achieve high performance, performing such optimizations manually can be a daunting task for an application developer.

To address the productivity challenge for stencil code optimization on GPUs, many DSL code generators have been proposed in past, including, but not limited to Stella [39], Halide [84], PolyMage [71], Forma [85], SDSL [42], and Overtile [43]. From an outsider's perspective, stencil optimization is a practically solved problem, given the existence of popular code generators like Halide and PolyMage. However, the set of optimizations implemented in these code generators do not generalize well to a broader class of 2D/3D stencil computations. For example, PolyMage is only evaluated for 2D imaging pipelines, and Halide requires expertise in writing efficient schedules.

This dissertation was conceived with the objective of automatically accelerating 3D stencil computations on GPUs. During its inception, Overtile [43] and Halide [84] were the state-of-the-art DSL code generators for stencil computations. Overtile overlap-tiled all the dimensions of the input domain, incurring high volume of recomputations for 3D stencils. Halide required writing the schedules manually; auto-tuning using OpenTuner [4] was not feasible for GPU codes. As a first step towards controlling the volume of recomputations, we created several manual implementations that used split tiling instead of overlapped tiling. However, we soon realized that guaranteeing coalesced reads and writes for the halo region was extremely challenging with split tiling, and the code generation would be complicated due to the increase in the tile shapes with increasing stencil dimensionality. We then automated a variation the 2.5D (3.5D) streaming strategy proposed by Micikevicius [67] (Nguyen et al. [72]). Unlike their manual implementations, our auto-generated variation could optimize a wider class of stencil computations by leveraging associative reordering of the stencil operator [88, 87].

For multi-statement stencil computations that can be expressed as a DAG, fusion is the equivalent of time tiling. Such stencil DAGs are commonplace in image processing pipelines, climate modeling, and seismic wave applications. Several efforts proposed fusion heuristics that would search a vast space of possible fusion candidates [38, 116]. Instead, we proposed a simple and adaptive greedy fusion heuristic to find the most profitable fusion candidate pair, and greedily fuse the nodes in the pair if the resulting code did not exceed the hardware limits on GPU constraints, or incur large volume of recomputations [87]. Since then, several works have explored similar greedy heuristics, demonstrating the applicability of greedy algorithms in fusing stencil operators in a DAG [48, 111].

While trying to optimize the high-order, compute-bound stencils from SW4 benchmark [104], we realized that the bandwidth optimizations were ineffective on them – the real performance bottleneck for those stencils was not the memory bandwidth, but the memory access latency combined with a high register pressure and low achievable occupancy. Prior efforts on optimizing compute-bound stencils on GPUs involved either reducing the computations via common subexpression elimination (CSE) [8], or introducing recomputations via rematerialization [50]. However, the high-order stencils in SW4 are not symmetrical to benefit from CSE, and exhibit a rather complex many-to-many reuse to benefit from rematerialization. To optimize such stencils, we developed novel reordering strategies aimed towards alleviating register pressure. For compute-bound, register-limited stencil kernels, our optimization can be adapted as a tuning strategy that splits the kernel into compute-bound, but slightly less register-limited sub kernels, increases the unrolling factor in each sub kernel, and then performs associative reordering on each sub kernel to exploit register-level reuse without spilling.

All the above-mentioned optimization strategies are integrated in StencilGen, which is the central contribution of this dissertation. StencilGen is a DSL code generator that can auto-generate optimized CUDA code for both bandwidth- and compute-bound stencils. StencilGen incorporates many code optimization strategies, namely overlapped tiling, serial and concurrent streaming, efficient data placement in GPU resources, loop unrolling, double buffering, fusion of stencil statements, and associative reordering. For a bandwidth-bound stencil DAG, StencilGen aims to (a) efficiently balance the use of GPU resources like registers and shared memory; (b) minimize the data movement by using spatial/temporal streaming and fusing stencil operators; and (c) account for the redundancy introduced by overlapped tiling in the fusion heuristic. To assist the underlying compiler in efficient register allocation for high-order stencils, StencilGen leverages operator associativity and distributivity for register-pressure-aware reordering. To the best of our knowledge, StencilGen is the first to automate such diverse optimization strategies in a single framework. The work in the dissertation advances the state-of-the-art by improving the programmer productivity and addressing the performance challenges of a wide variety of stencil computations. The effectiveness of the proposed optimizations is demonstrated through experimental evaluation on a range of stencils extracted from application codes, comparing the performance with several state-of-the-art stencil optimizers.

7.2 Ongoing Research

Implementing optimizations to hide memory latency Many register-limited stencils perform best when the number of registers per thread is set to the hardware maximum (255). However, this configuration results in only 12.5% occupancy. With lesser number of active threads per SM, memory latency can easily become a performance bottleneck. Double buffering is often used to overlap computation with communication, thereby hiding memory access latency. Although double buffering increases the register usage, it may lead to performance improvement in bandwidth-bound, and possibly some compute-bound stencils that are not register-limited. Implementing double buffering in StencilGen is a currently ongoing engineering effort.

Auto-tuning with OpenTuner OpenTuner [4] is a framework for building domain-specific multi-objective program auto-tuners. OpenTuner provides an easy interface to communicate with the tuned program, and customizable configuration to allow inclusion of custom search strategies. OpenTuner uses an ensemble of disparate search techniques to tune for performance.

The performance of a stencil code can vary dramatically with variations in the block shape and size, the unrolling factors, the per-thread register configuration, and the tiling strategy. We are currently exploring the use of OpenTuner to navigate the configuration space, and converge on the best code version that can be generated by StencilGen for an input stencil computation.

7.3 Future Research Directions

Despite the fact that there is a rich and vast literature dedicated to optimization of stencil computations on GPUs, the growing complexity of stencil computations and the evolution of GPU architecture presents opportunities to extend and enhance the contributions of the dissertation. Below, we discuss three avenues for future research.

Extending StencilGen to generate multi-GPU code GPU clusters are often used to accelerate stencil computations beyond the size that can be held on a single GPU, or stencil computations with a mesh hierarchy [28, 118]. In multi-GPU stencil computation, the computational domain is decomposed and the subdomains are assigned to individual GPU nodes. One has to carefully orchestrate the communication of the halo regions between the different GPU nodes so that it overlaps with the computation. Manually writing multi-GPU codes can be tedious and error-prone. Even though recent advances like CUDA-aware MPI[9] and NVIDIA GPUDirect[10][119] simplify programmability and code generation, there are still several engineering and research challenges, right from deciding what the input specification should be, to the tuning methodology for the subdomain size configuration.

Implementing the proposed optimizations in a source-to-source compiler Most of the existing works on optimizing stencil computations require the input to be expressed in a domain specific language (DSL). Some of these DSLs are stand-alone (e.g. Patus [18] Forma [85], PolyMage [71]), and some are embedded (e.g. SDSL [42], Halide [84], Stella [39]). The expressive power and user-friendliness of these DSLs vary, which makes choosing one amongst these difficult. In using these DSL compilers, one has to rewrite the computation in the domain specific language. This may prove challenging for large-scale applications, which have a mix of stencil computations interspersed between general non-compute-intensive code. Therefore, even though these DSL compilers are powerful for kernels that are completely expressible in the DSL, their use can be limited in such large-scale applications. On the other hand, polyhedral model-based compilers like PPCG offer a more natural way of expressing the computation, but do not offer aggressive optimizations like the DSL compilers.

A solution to this problem can be formulated by taking inspiration from the existing retargetable compiler generation frameworks. These frameworks parse different

[9]https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[10]https://developer.nvidia.com/gpudirect

input languages into a common Intermediate Representation (IR), so that the compiler optimizations are language and platform agnostic. We can choose a compiler framework with the capability of handling input languages that are most commonly used in the real-world stencil applications (usually C, C++, Fortran). We can then add the functionality to detect the regions of code that are data-parallel (e.g. SCoP detection in PolyOpt[11]), and amenable to acceleration on GPU using our optimization framework. The code corresponding to these IR nodes can be redirected to any transformation framework, and replaced by a simple invocation of the accelerated function in the original code. A separate global optimization pass over the original program can further optimize the data transfers between accelerated code and the unoptimized code.

Revisiting register allocation from the perspective of interference graph Register pressure in a program is dominated by the size of maximum clique in the interference graph. Optimizing other parts of the code may not benefit if the bottleneck is not optimized first. A possible research direction is to explore a more general low-level transformation implemented within GCC or LLVM, which would iteratively find the maximum clique in the interference graph, and try to reorder instructions so that the size of the maximum clique is reduced. The reduction will be achieved by adjusting live ranges so that all the `uses` of a chosen variable m are moved before any `def` of another variable n, effectively eliminating any conflicts between m and n.

An easier, but expensive way is to record all the possible statement reorderings that reduce the size of the maximum clique, and select a subset of reorderings that preserve the convexity in the dependence graph. Instead, we desire to come up with

[11]http://web.cse.ohio-state.edu/ pouchet/software/polyopt/

a set of rules that, at best, comes up with an instruction ordering so that the number of live ranges at each program point are almost uniform. This research direction is complex, and involves further concretizing of ideas.

APPENDIX A

StencilGen Grammar

StencilGen accepts as input a description of the stencil computation consistent with the grammar described in Figure A.1. The input to StencilGen code generator is a *.idsl* file containing the stencil specification. The parser parses the input and populates the internal AST data structure. A lightweight preprocessing phase verifies the correctness of the syntax of parsed data, and matches the data types of the stencil call arguments to the stencil function parameters. For register optimizations, the stencil statements have to be further lowered into three-address instructions with accumulations. The lowered instructions must conform to the grammar described in Figure A.2.

In the grammar, *Params* represents program parameters that are used to define the array and stencil domains (M, N in Listing 2.2), and *Iterators* represents loop iterators (i, j in Listing 2.2). The parameters are invariant with respect to the stencil loop nest, i.e., their values do not change during the execution of all the stencil functions in the *.idsl* file. The computational loops are not explicitly defined in the DSL, as some of the loop nests will be implicit due to data parallelism in the CUDA code. However, we assume that (a) each computational loop has a unique iterator

```
         ⟨Program⟩ ::= ⟨Params⟩⟨Iters⟩⟨Decls⟩⟨CopyIn⟩⟨StencilDefns⟩⟨StencilCalls⟩⟨CopyOut⟩
          ⟨Params⟩ ::= parameter ⟨ParamList⟩;
       ⟨ParamList⟩ ::= ⟨Parameter⟩,⟨ParamList⟩ | ⟨Parameter⟩=⟨Const⟩,⟨ParamList⟩ | ε
           ⟨Iters⟩ ::= iterator ⟨IterList⟩;
        ⟨IterList⟩ ::= ⟨Iterator⟩,⟨IterList⟩ | ⟨Iterator⟩
           ⟨Decls⟩ ::= ⟨DataType⟩ ⟨AccessesList⟩;⟨Decls⟩ | ε
    ⟨AccessesList⟩ ::= ⟨ArrayDecl⟩,⟨AccessesList⟩ | ⟨ID⟩,⟨AccessesList⟩ | ε
       ⟨ArrayDecl⟩ ::= ⟨ID⟩⟨ArrayDoms⟩
       ⟨ArrayDoms⟩ ::= [⟨DomExpr⟩]⟨ArrayDoms⟩ | [⟨DomExpr⟩]
          ⟨CopyIn⟩ ::= copyin ⟨IDList⟩;
          ⟨IDList⟩ ::= ⟨ID⟩,⟨IDList⟩ | ⟨ID⟩
    ⟨StencilDefns⟩ ::= ⟨StencilDefn⟩ ⟨StencilDefns⟩ | ⟨StencilDefn⟩
     ⟨StencilDefn⟩ ::= function ⟨StencilName⟩ (⟨IDList⟩) { ⟨StencilBody⟩ }
     ⟨StencilBody⟩ ::= ⟨Decls⟩ ⟨DataPlacements⟩ ⟨StmtList⟩;
  ⟨DataPlacements⟩ ::= shmem ⟨IDList⟩ | gmem ⟨IDList⟩ | ε
        ⟨StmtList⟩ ::= ⟨Assignment⟩;⟨StmtList⟩ | ⟨Assignment⟩
      ⟨Assignment⟩ ::= ⟨BaseExpr⟩⟨AssignOp⟩⟨Expr⟩
            ⟨Expr⟩ ::= Side-effect-free expression of ⟨BaseExpr⟩
        ⟨BaseExpr⟩ ::= ⟨ID⟩ | ⟨ArrayExpr⟩
       ⟨ArrayExpr⟩ ::= ⟨ID⟩⟨ArrayDims⟩
       ⟨ArrayDims⟩ ::= [⟨AccessExpr⟩]⟨ArrayDims⟩ | [⟨AccessExpr⟩]
      ⟨AccessExpr⟩ ::= Side-effect-free affine function of ⟨AccessElement⟩
   ⟨AccessElement⟩ ::= ⟨Iterator⟩ | ⟨Const⟩
    ⟨StencilCalls⟩ ::= ⟨StencilCall⟩ ⟨StencilCalls⟩ | ⟨StencilCall⟩
     ⟨StencilCall⟩ ::= ⟨StencilDomain⟩ : ⟨FuncName⟩ (⟨IDList⟩);
   ⟨StencilDomain⟩ ::= [⟨DomExpr⟩:⟨DomExpr⟩]⟨StencilDomain⟩ | [⟨DomExpr⟩:⟨DomExpr⟩]
         ⟨DomExpr⟩ ::= ⟨Const⟩ | ⟨Parameter⟩ | ⟨Parameter⟩ + ⟨Const⟩ | ⟨Parameter⟩ - ⟨Const⟩
         ⟨CopyOut⟩ ::= copyout ⟨IDList⟩;
        ⟨AssignOp⟩ ::= = | += | -= | *= | ...
        ⟨DataType⟩ ::= double | float | int | bool | ...
```

Figure A.1: Grammar for valid input to StencilGen.

assigned to it; (b) an iterator is only modified in unit steps by the increment expression of the loop; (c) the iterators are immutable within the stencil statements.

$$
\begin{aligned}
\langle\text{CodeBody}\rangle &::= \langle\text{StmtList}_l\rangle\,\texttt{;}\\
\langle\text{StmtList}_l\rangle &::= \langle\text{Assignment}_l\rangle\,\texttt{;}\,\langle\text{StmtList}_l\rangle \mid \langle\text{Assignment}_l\rangle\\
\langle\text{Assignment}_l\rangle &::= \langle\text{BaseExpr}\rangle\langle\text{AssignOp}\rangle\langle\text{Expr}_l\rangle\\
\langle\text{Expr}_l\rangle &::= \langle\text{BaseExpr}\rangle\langle\text{BinaryOp}_l\rangle\langle\text{BaseExpr}\rangle \mid \langle\text{BaseExpr}\rangle\\
\langle\text{BinaryOp}_l\rangle &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{|} \mid \texttt{\&} \mid \ldots
\end{aligned}
$$

Figure A.2: Grammar for the lowered IR for instruction reordering

The *AccessesList* in *Decls* represents all the arrays and scalars that will be used in the stencil computation. The declarations inside the stencil function represented by *StencilBody* have local scope; the declarations in *Program* have global scope, and represent the formal parameters to the host function. The scalars are represented by *ID*. They could be invariants in the computation, but unlike the parameters represented by *Params*, they are not used to define an array or stencil domain. For Listing 2.2, the *AccessesList* will have arrays *out*[*M*][*N*] and *in*[*M*][*N*], and scalar *a*. The arrays and scalars belonging to *CopyIn* and *CopyOut* will copied in from host to device, and copied out from device to host, respectively; these arrays and scalars must be present in the *AccessesList*. No data layout transformations are performed at present while copying data from host to device or vice versa.

An *.idsl* file can have multiple stencil functions, for example, a central stencil function to compute the inner core of the output domain, and several boundary functions to compute the output domain at the boundary. These stencil function definitions (*StencilDefns*) are followed by the stencil calls (*StencilCalls*). The arrays and scalars in the stencil call arguments must appear in *AccessesList* of *Decls*. Irrespective of whether the formal parameters in the stencil function appear in *AccessesList* or not, their data types and domain shape is inferred by matching the data types and domain shape of the corresponding argument in the stencil call. For example, in Listing 2.2, the data type of both A and B in line 7 and line 13 will be matched to that of array *in* and *out* respectively. The preprocessing step verifies that the formal parameters of a stencil function are assigned the same data type, and mapped exclusively to either an array or a scalar across all the stencil calls to that function. Currently, we do not support type casting.

The input domain on which a stencil has to be applied is defined by the *StencilDomain* in the stencil call; the output domain is automatically computed as the convex hull over which the stencil application will produce a valid result. The iterators are mapped on to these stencil domains in sequence. For example, in the *five_point_average* stencil called at line 21 of Listing 2.2, iterator j will loop from 1 to $M-2$, and iterator i will loop from 1 to $N-2$.

The *StencilBody* comprises a sequence of stencil statements (*StmtList*). Each statement has an LHS *BaseExpr*, and an RHS *Expr*; the operator can be an assignment operator, or an update operator (e.g, +=, -=, *=). *BaseExpr* refers to an array expression, or a variable. The RHS is a side-effect free expression comprising *BaseExpr*. This allows the RHS to have simple, side-effect free mathematical functions like sine, sqrt, min, etc., with *BaseExpr* as arguments.

For the arrays in *BaseExpr*, the index expressions, represented by $AccessExpr$ are themselves side-effect-free, and affine functions of the iterators, parameters, and constants. Thus, we allow `A[i+2]`, but not `A[i*j+2]`.

In summary, the following restrictions are imposed on the input:

- The stencil statements (lines 8–10 and line 14) must not modify the iterators (i and j) and the parameters (M, and N)
- The statements must be side-effect-free, and can only have mathematical functions like *sine*, *sqrt*, etc.
- The memory accesses in the stencil statements are either scalars or array elements. The array index expressions are affine function of the loop iterators, parameters, and constants.

APPENDIX B

Code Generation with StencilGen

Command Line Options

- `--out-file`: accepts string, the output file name. Defaults to out.cu
- `--blockdim`: accepts input in the form *id*=*val*, where *id* represents the block dimension, and *val* represents the block size along that dimension
- `--unroll`: accepts input in the form *id*=*val*, where *id* represents the iterator, and *val* represents the unroll factor along that iterator
- `--serial-stream`: accepts string, the dimension along which to stream in serial. Defaults to the outermost iterator
- `--concurrent-stream`: accepts string, the dimension along which to tile and stream in parallel. Defaults to the outermost iterator
- `--shared-mem`: accepts bool, whether to use shared memory or not. Defaults to true
- `--cregs`: accepts int, the number of registers that will be used in compilation

Listing B.1: j2d5pt.idsl: two time steps of a 2d 5-point Jacobi stencil

```
parameter M=512, N=512;
iterator j, i;
double in[M,N], out[M,N];
copyin in, out;

stencil j2d5pt (out, in) {
    double t[M,N];
    temp[j][i] = 0.1*in[j-1][i] +
                 0.2*(in[j][i-1] + in[j][i] + in[j][i+1]) +
                 0.3*in[j+1][i];

    out[j][i] = 0.1*t[j-1][i] +
                0.2*(t[j][i-1] + t[j][i] + t[j][i+1]) +
                0.3*t[j+1][i];

j2d5pt (out, in);
copyout out;
```

Invocation Example Given an input DSL file `sw4.idsl`, one can generate CUDA code that uses shared memory, and serial-streams along the k dimension with:

```
$. ./stencilgen sw4.idsl --out-file sw4.cu --serial-stream k --cregs 255 --shared-mem true --blockdim x=16,y=8,z=1 --unroll k=2,j=1,i=1
```

and a version that only uses global memory, and no streaming with:

```
$. ./stencilgen sw4.idsl --out-file sw4.cu --shared-mem false --cregs 255 --blockdim x=8,y=8,z=4 --unroll k=2,j=1,i=1
```

We demonstrate the different type of code generated with StencilGen on the example of Listing B.1, which computes two time steps of a 2D 5-point Jacobi stencil. The device functions for the generated code is shown in Listings B.2 – B.5.

Listing B.2: StencilGen-generated device code with shared memory, without streaming

```
#define MM 512
#define NN 512
__global__ void j2d5pt (double * __restrict__ d_out, double * __restrict__ d_in, int M, int N) {
  int i0 = (int)(blockIdx.x)*((int)blockDim.x-4);
  int i = i0 + (int)(threadIdx.x);
  int j0 = (int)(blockIdx.y)*((int)blockDim.y-4);
  int j = j0 + (int)(threadIdx.y);

  double (*out)[NN] = (double (*)[NN]) d_out;
  double (*in)[NN] = (double (*)[NN]) d_in;

  double __shared__ in_shm[8][16];
  double __shared__ t_shm[8][16];

  if (j<=min(j0+blockDim.y-1, M-1) && i<=min(i0+blockDim.x-1, N-1)) {
        in_shm[j-j0][i-i0] = in[j][i];
  }

  __syncthreads ();
  if (j>=max(1, j0+1) && j<=min(j0+blockDim.y-2, M-2) && i>=max(1, i0+1) && i<=min(i0+blockDim.x-2, N-2)) {
        t_shm[j-j0][i-i0] = 0.1 * in_shm[j-j0-1][i-i0] + 0.2 * (in_shm[j-j0][i-i0-1] + in_shm[j-j0][i-i0] + in_shm[j-j0][i-i0+1]) + 0.3 * in_shm[j-j0+1][i-i0];
  }
  __syncthreads ();
  if (j>=max(2, j0+2) && j<=min(j0+blockDim.y-3, M-3) && i>=max(2, i0+2) && i<=min(i0+blockDim.x-3, N-3)) {
        out[j][i] = 0.1 * t_shm[j-j0-1][i-i0] + 0.2 * (t_shm[j-j0][i-i0-1] + t_shm[j-j0][i-i0] + t_shm[j-j0][i-i0+1]) + 0.3 * t_shm[j-j0+1][i-i0];
  }
}
```

Listing B.3: StencilGen-generated device code with shared memory, registers, and serial streaming

```
__global__ void j2d5pt (double * __restrict__ d_out, double * __restrict__ d_in, int M, int N) {
 int i0 = (int)(blockIdx.x)*((int)blockDim.x-4);
 int i = i0 + (int)(threadIdx.x);

 double (*out)[NN] = (double (*)[NN]) d_out;
 double (*in)[NN] = (double (*)[NN]) d_in;

 double in_reg_m1, __shared__ in_shm_c0[16], in_reg_p1;
 double t_reg_m2, __shared__ t_shm_m1[16], t_reg_c0;

 if (i<=min(i0+blockDim.x-1, N-1)) {
        in_shm_c0[i-i0] = in[1][i];
        in_reg_m1 = in[0][i];
        in_reg_p1 = in[2][i];
 }

 for (int j=1; j<=M-2; j++) {
   __syncthreads ();
   if (i>=max(1, i0+1) && i<=min(i0+blockDim.x-2, N-2)) {
        t_reg_c0 = 0.1 * in_reg_m1 + 0.2 * (in_shm_c0[i-i0-1] + in_shm_c0[i-i0] + in_shm_c0[i-i0+1]) + 0.3 * in_reg_p1;
   }
   __syncthreads ();
   if (i>=max(2, i0+2) && i<=min(i0+blockDim.x-3, N-3)) {
        out[j-1][i] = 0.1 * t_reg_m2 + 0.2 * (t_shm_m1[i-i0-1] + t_shm_m1[i-i0] + t_shm_m1[i-i0+1]) + 0.3 * t_reg_c0;
   }
   __syncthreads ();

   if (i<=min(i0+blockDim.x-1, N-1)) {
        in_reg_m1 = in_shm_c0[i-i0];
        in_shm_c0[i-i0] = in_reg_p1;
        t_reg_m2 = t_shm_m1[i-i0];
        t_shm_m1[i-i0] = t_reg_c0;
        in_reg_p1 = in[min(j+2, M-1)][i];
   }
 }
}
```

Listing B.4: StencilGen-generated device code with shared memory, registers, and concurrent streaming

```
__global__ void j2d5pt (double * __restrict__ d_out, double *
    ↪ __restrict__ d_in, int M, int N) {
  int i0 = (int)(blockIdx.x)*((int)blockDim.x-4);
  int i = i0 + (int)(threadIdx.x);
  int j0 = (int)(blockIdx.y)*(8);

  double (*out)[NN] = (double (*)[NN]) d_out;
  double (*in)[NN] = (double (*)[NN]) d_in;

  double in_reg_m1, __shared__ in_shm_c0[16], in_reg_p1;
  double t_reg_m2, __shared__ t_shm_m1[16], t_reg_c0;

  if (i<=min(i0+blockDim.x-1, N-1)) {
      in_shm_c0[i-i0] = in[j0+1][i];
      in_reg_m1 = in[j0][i];
      in_reg_p1 = in[j0+2][i];
  }

  for (int j=j0+1; j<=min (M-2, j0+10); j++) {
      __syncthreads ();
      if (i>=max(1, i0+1) && i<=min(i0+blockDim.x-2, N-2)) {
          t_reg_c0 = 0.1 * in_reg_m1 + 0.2 * (in_shm_c0[i-i0-1]
             ↪ + in_shm_c0[i-i0] + in_shm_c0[i-i0+1]) + 0.3 *
             ↪ in_reg_p1;
      }
      __syncthreads ();
      if (i>=max(2, i0+2) && i<=min(i0+blockDim.x-3, N-3)) {
          out[max (j0+2, j-1)][i] = 0.1 * t_reg_m2 + 0.2 * (
             ↪ t_shm_m1[i-i0-1] + t_shm_m1[i-i0] + t_shm_m1[i-
             ↪ i0+1]) + 0.3 * t_reg_c0;
      }
      __syncthreads ();

      if (i<=min(i0+blockDim.x-1, N-1)) {
          in_reg_m1 = in_shm_c0[i-i0];
          in_shm_c0[i-i0] = in_reg_p1;
          t_reg_m2 = t_shm_m1[i-i0];
          t_shm_m1[i-i0] = t_reg_c0;
          in_reg_p1 = in[min(j+2, M-1)][i];
      }
  }
}
```

Listing B.5: StencilGen-generated device code with shared memory, registers, unrolling, and serial streaming

```
__global__ void j2d5pt (double * __restrict__ d_out, double *
   ↪ __restrict__ d_in, int M, int N) {
 int i0 = (int)(blockIdx.x)*(2*(int)blockDim.x-4);
 int i = i0 + (int)(threadIdx.x);
 ...
 double in_reg_m1[2], __shared__ in_shm_c0[32], in_reg_p1[2];
 double t_reg_m2[2], __shared__ t_shm_m1[32], t_reg_c0[2];
 #pragma unroll 2
 for (int l1_i2=0,i2=i; l1_i2<2; l1_i2++, i2 += blockDim.x)
  if (i2<=min(i0+2*blockDim.x-1, N-1)) {
    in_shm_c0[i2-i0] = in[1][i2];
    in_reg_m1[l1_i2] = in[0][i2];
    in_reg_p1[l1_i2] = in[2][i2];
  }
 for (int j=1; j<=M-2; j++) {
  __syncthreads ();
  #pragma unroll 2
  for (int r0_i2=0,i2=i; r0_i2<2; r0_i2++, i2+=blockDim.x)
   if (i2>=max(1, i0+1) && i2<=min(i0+2*blockDim.x-2, N-2)) {
     t_reg_c0[r0_i2] = 0.1 * in_reg_m1[r0_i2] + 0.2 * (in_shm_c0
        ↪ [i2-i0-1] + in_shm_c0[i2-i0] + in_shm_c0[i2-i0+1]) +
        ↪ 0.3 * in_reg_p1[r0_i2];
   }
  __syncthreads ();
  #pragma unroll 2
  for (int r1_i2=0,i2=i; r1_i2<2; r1_i2++, i2+=blockDim.x)
   if (i2>=max(2, i0+2) && i2<=min(i0+2*blockDim.x-3, N-3)) {
     out[j-1][i2] = 0.1 * t_reg_m2[r1_i2] + 0.2 * (t_shm_m1[i2-
        ↪ i0-1] + t_shm_m1[i2-i0] + t_shm_m1[i2-i0+1]) + 0.3 *
        ↪ t_reg_c0[r1_i2];
   }
  __syncthreads ();
  #pragma unroll 2
  for (int l2_i2=0,i2=i; l2_i2<2; l2_i2++, i2 += blockDim.x)
   if (i2<=min(i0+2*blockDim.x-1, N-1)) {
     in_reg_m1[l2_i2] = in_shm_c0[i2-i0];
     in_shm_c0[i2-i0] = in_reg_p1[l2_i2];
     t_reg_m2[l2_i2] = t_shm_m1[i2-i0];
     t_shm_m1[i2-i0] = t_reg_c0[l2_i2];
     in_reg_p1[l2_i2] = in[min(j+2, M-1)][i2];
  }
 }
}
```

BIBLIOGRAPHY

[1] Aho, A., Lam, M., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools (2nd ed).* Pearson, 2007.

[2] Aho, A. V., and Johnson, S. C. Optimal code generation for expression trees. In *Proceedings of Seventh Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1975), STOC '75, ACM, pp. 207–217.

[3] Aho, A. V., Johnson, S. C., and Ullman, J. D. Code generation for expressions with common subexpressions. *J. ACM 24*, 1 (Jan. 1977), 146–160.

[4] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 303–316.

[5] Appel, A. W., and Supowit, K. J. Generalization of the sethi-ullman algorithm for register allocation. *Softw. Pract. Exper. 17*, 6 (June 1987), 417–421.

[6] Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., and Aamodt, T. M. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software* (April 2009), pp. 163–174.

[7] Bandishti, V., Pananilath, I., and Bondhugula, U. Tiling stencil computations to maximize parallelism. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for* (Nov 2012), pp. 1–11.

[8] Basu, P., Hall, M., Williams, S., Straalen, B. V., Oliker, L., and Colella, P. Compiler-directed transformation for higher-order stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International* (May 2015), pp. 313–323.

[9] Basu, P., Williams, S., Straalen, B. V., Oliker, L., Colella, P., and Hall, M. Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers. *Parallel Computing 64* (2017), 50 – 64. High-End Computing for Next-Generation Scientific Discovery.

[10] Bauer, M., Cook, H., and Khailany, B. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 12:1–12:11.

[11] Berson, D. A., Gupta, R., and Soffa, M. L. Integrated instruction scheduling and register allocation techniques. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, UK, 1999), LCPC '98, Springer-Verlag, pp. 247–262.

[12] Bertolacci, I. J., Olschanowsky, C., Harshbarger, B., Chamberlain, B. L., Wonnacott, D. G., and Strout, M. M. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (New York, NY, USA, 2015), ICS '15, ACM, pp. 197–206.

[13] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 101–113.

[14] Briggs, P., Cooper, K. D., and Torczon, L. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1992), PLDI '92, ACM, pp. 311–321.

[15] Briggs, P., Cooper, K. D., and Torczon, L. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 428–455.

[16] Chaitin, G. J. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (New York, NY, USA, 1982), SIGPLAN '82, ACM, pp. 98–105.

[17] Chang, C.-M., Chen, C.-M., and King, C.-T. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers and Mathematics with Applications* (1997).

[18] Christen, M., Schenk, O., and Burkhart, H. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium* (2011), IPDPS '11, IEEE Computer Society, pp. 676–687.

[19] Christen, M., Schenk, O., Neufeld, E., Messmer, P., and Burkhart, H. Parallel data-locality aware stencil computations on modern micro-architectures. In *2009 IEEE International Symposium on Parallel Distributed Processing* (May 2009), pp. 1–10.

[20] Codina, J. M., Sanchez, J., and Gonzalez, A. A unified modulo scheduling and register allocation technique for clustered processors. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques* (2001), pp. 175–184.

[21] Colombet, Q., Boissinot, B., Brisk, P., Hack, S., and Rastello, F. Graph-coloring and treescan register allocation using repairing. In *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)* (Oct 2011), pp. 45–54.

[22] Cong, J., Huang, M., and Zou, Y. Accelerating fluid registration algorithm on multi-fpga platforms. In *Proc. of Intl. Conference on Field Programmable Logic and Applications* (2011), FPL. to appear.

[23] Cong, J., and Zou, Y. Lithographic aerial image simulation with fpga-based hardware acceleration. In *ACM/SIGDA symp. on Field programmable gate arrays* (2008), FPGA, pp. 67–76.

[24] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008), SC '08, IEEE Press, pp. 4:1–4:12.

[25] De La Cruz, R., Araya-Polo, M., and Cela, J. M. Introducing the semi-stencil algorithm. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I* (Berlin, Heidelberg, 2010), PPAM'09, Springer-Verlag, pp. 496–506.

[26] Deitz, S. J., Chamberlain, B. L., and Snyder, L. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th International Conference on Supercomputing* (New York, NY, USA, 2001), ICS '01, ACM, pp. 65–77.

[27] Domagala, L., van Amstel, D., Rastello, F., and Sadayappan, P. Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Proceedings of the 25th International Conference on Compiler Construction* (New York, NY, USA, 2016), CC 2016, ACM, pp. 143–151.

[28] Dubey, A. Stencils in scientific computations. In *Proceedings of the Second Workshop on Optimizing Stencil Computations* (New York, NY, USA, 2014), WOSC '14, ACM, pp. 57–57.

[29] ExaCT: Center for Exascale Simulation of Combustion in Turbulence: Proxy App Software. `https://exactcodesign.org/proxy-app-software/`, 2013.

[30] Advanced Stencil-Code Engineering (ExaStencils). `http://www.exastencils.org/`.

[31] Frigo, M., and Johnson, S. G. The design and implementation of fftw3. *Proceedings of the IEEE 93*, 2 (Feb 2005), 216–231.

[32] Gan, L., Fu, H., Xue, W., Xu, Y., Yang, C., Wang, X., Lv, Z., You, Y., Yang, G., and Ou, K. Scaling and analyzing the stencil performance on multi-core and many-core architectures. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)* (Dec 2014).

[33] Goodman, J. R., and Hsu, W.-C. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 2Nd International Conference on Supercomputing* (New York, NY, USA, 1988), ICS '88, ACM.

[34] Goto, K., and Geijn, R. A. v. d. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3 (May 2008), 12:1–12:25.

[35] Govindarajan, R., Yang, H., Zhang, C., Amaral, J. N., and Gao, G. R. Minimum register instruction sequence problem: Revisiting optimal code generation for DAGs. In *Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium* (Washington, DC, USA, 2001), IPDPS '01, IEEE Computer Society, pp. 26–33.

[36] Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., and Verdoolaege, S. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2014), CGO '14, ACM, pp. 66:66–66:75.

[37] Grosser, T., Cohen, A., Kelly, P. H. J., Ramanujam, J., Sadayappan, P., and Verdoolaege, S. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (2013), GPGPU-6, ACM, pp. 24–31.

[38] Gysi, T., Grosser, T., and Hoefler, T. MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (New York, NY, USA, 2015), ICS '15, ACM, pp. 177–186.

[39] Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., and Schulthess, T. C. STELLA: a domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), SC '15, ACM, pp. 41:1–41:12.

[40] Hall, M., Chame, J., Chen, C., Shin, J., Rudy, G., and Khan, M. M. Loop transformation recipes for code generation and auto-tuning. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing* (Berlin, Heidelberg, 2010), LCPC'09, Springer-Verlag, pp. 50–64.

[41] Hayes, A. B., Li, L., Chavarría-Miranda, D., Song, S. L., and Zhang, E. Z. Orion: A framework for GPU occupancy tuning. In *Proceedings of the 17th International Middleware Conference* (New York, NY, USA, 2016), Middleware '16, ACM, pp. 18:1–18:13.

[42] Henretty, T., Veras, R., Franchetti, F., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, ACM, pp. 13–24.

[43] Holewinski, J., Pouchet, L.-N., and Sadayappan, P. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing* (2012), ICS '12, ACM, pp. 311–320.

[44] Hong, S., and Kim, H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 152–163.

[45] High-Performance Geometric Multigrid. `https://hpgmg.org/`, 2016.

[46] Intel C++ compiler. `https://software.intel.com/en-us`, 2017.

[47] J. Quinlan, D. Rose: Compiler support for object-oriented frameworks. 215–226.

[48] Jangda, A., and Bondhugula, U. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2018), PPoPP '18, ACM, pp. 261–275.

[49] Jia-Wei, H., and Kung, H. T. I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1981), STOC '81, ACM, pp. 326–333.

[50] Jin, M., Fu, H., Lv, Z., and Yang, G. Libra: An automated code generation and tuning framework for register-limited stencils on GPUs. In *Proceedings of the ACM International Conference on Computing Frontiers* (New York, NY, USA, 2016), CF '16, ACM, pp. 92–99.

[51] Ke ler, C. W., and Rauber, T. Generating optimal contiguous evaluations for expression dags. *Computer Languages 21*, 2 (1995), 113 – 127.

[52] Koes, D. R., and Goldstein, S. C. A global progressive register allocator. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 204–215.

[53] Kral, S., Franchetti, F., Lorenz, J., Ueberhuber, C. W., and Wurzinger, P. FFT compiler techniques. In *Compiler Construction: 13th International Conference, CC 2004* (2004), Springer Berlin Heidelberg, pp. 217–231.

[54] Krall, A., and Barany, G. Optimistic integrated instruction scheduling and register allocation. CPC '10, John Wiley and Sons, Ltd.

[55] Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), PLDI '07, ACM, pp. 235–244.

[56] Lai, J., and Seznec, A. Performance upper bound analysis and optimization of SGEMM on fermi and kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Washington, DC, USA, 2013), CGO '13, IEEE Computer Society, pp. 1–10.

[57] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime*

Optimization (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–.

[58] Li, A., Song, S. L., Kumar, A., Zhang, E. Z., Chavarría-Miranda, D., and Corporaal, H. Critical points based register-concurrency autotuning for GPUs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)* (March 2016), pp. 1273–1278.

[59] Li, A., van den Braak, G.-J., Kumar, A., and Corporaal, H. Adaptive and transparent cache bypassing for gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 17:1–17:12.

[60] Lueh, G.-Y., Gross, T., and Adl-Tabatabai, A.-R. Fusion-based register allocation. *ACM Trans. Program. Lang. Syst. 22*, 3 (May 2000), 431–470.

[61] Ma, W., Krishnamoorthy, S., Villa, O., Kowalski, K., and Agrawal, G. Optimizing tensor contraction expressions for hybrid CPU-GPU execution. *Cluster Computing 16* (2013).

[62] Mallinson, A. C., Beckingsale, D. A., Gaudin, W. P., Herdman, J. A., Levesque, J. M., and Jarvis, S. A. Cloverleaf: Preparing hydrodynamics codes for exascale.

[63] Maruyama, N., and Aoki, T. Optimizing stencil computations for NVIDIA Kepler GPUs.

[64] Maruyama, N., Nomura, T., Sato, K., and Matsuoka, S. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 11:1–11:12.

[65] McMECHAN, G. A. Migration by extrapolation of time-dependent boundary values*. *Geophysical Prospecting 31*, 3 (1983), 413–420.

[66] Meng, J., and Skadron, K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 256–265.

[67] Micikevicius, P. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (2009), GPGPU-2, ACM, pp. 79–84.

[68] Mössenböck, H., and Pfeiffer, M. *Linear Scan Register Allocation in the Context of SSA Form and Register Constraints.* Springer Berlin Heidelberg, 2002, pp. 229–246.

[69] Motwani, R., Palem, K. V., Sarkar, V., and Reyen, S. Combining register allocation and instruction scheduling. Tech. rep., Stanford, CA, USA, 1995.

[70] Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph. 35*, 4 (July 2016), 83:1–83:11.

[71] Mullapudi, R. T., Vasista, V., and Bondhugula, U. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15, ACM, pp. 429–443.

[72] Nguyen, A., Satish, N., Chhugani, J., Kim, C., and Dubey, P. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), SC '10, IEEE Computer Society, pp. 1–13.

[73] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53.

[74] Norris, C., and Pollock, L. L. A scheduler-sensitive global register allocator. In *Supercomputing '93. Proceedings* (Nov 1993), pp. 804–813.

[75] NVCC. `docs.nvidia.com/cuda/cuda-compiler-driver-nvcc`, 2017.

[76] Nvidia profiler. `http://docs.nvidia.com/cuda/profiler-users-guide`, 2017.

[77] NWChem Download, 2017.

[78] OpenACC compiler. `https://www.pgroup.com/doc/openacc_gs.pdf`, 2017.

[79] Pereira, A. D., Castro, M., Dantas, M. A. R., Rocha, R. C. O., and Góes, L. F. W. Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks. In *2017 International Conference on High Performance Computing Simulation (HPCS)* (July 2017), pp. 719–726.

[80] Pinter, S. S. Register allocation with instruction scheduling. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1993), PLDI '93, ACM, pp. 248–257.

[81] Poletto, M., and Sarkar, V. Linear scan register allocation. *ACM Trans. Program. Lang. Syst. 21*, 5 (Sept. 1999), 895–913.

[82] Prajapati, N., Ranasinghe, W., Rajopadhye, S., Andonov, R., Djidjev, H., and Grosser, T. Simple, accurate, analytical time modeling and optimal tile size selection for gpgpu stencils. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2017), PPoPP '17, ACM, pp. 163–177.

[83] Quintão Pereira, F. M., and Palsberg, J. Register allocation by puzzle solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 216–226.

[84] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), PLDI '13, ACM, pp. 519–530.

[85] Ravishankar, M., Holewinski, J., and Grover, V. Forma: A DSL for image processing applications to target GPUs and multi-core CPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs* (2015), GPGPU-8, ACM, pp. 109–120.

[86] Ravishankar, M., Micikevicius, P., and Grover, V. Fusing convolution kernels through tiling. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (2015), ARRAY 2015, ACM, pp. 43–48.

[87] Rawat, P. S., Hong, C., Ravishankar, M., Grover, V., Pouchet, L.-N., Rountev, A., and Sadayappan, P. Resource conscious reuse-driven tiling for GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (2016), PACT '16, ACM, pp. 99–111.

[88] Rawat, P. S., Hong, C., Ravishankar, M., Grover, V., Pouchet, L.-N., and Sadayappan, P. Effective resource management for enhancing performance of 2D and 3D stencils on GPUs. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (2016), GPGPU '16, ACM, pp. 92–102.

[89] Rawat, P. S., Sukumaran-Rajam, A., Rountev, A., Rastello, F., Pouchet, L.-N., and Sadayappan, P. Register optimizations for stencils on GPUs. In *Proceedings of the 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2018), PPoPP '18, ACM.

[90] Rong, H. Tree register allocation. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 67–77.

[91] Sarkar, V., and Barik, R. Extended linear scan: An alternate foundation for global register allocation. In *Proceedings of the 16th International Conference on Compiler Construction* (Berlin, Heidelberg, 2007), CC'07, Springer-Verlag, pp. 141–155.

[92] Sarkar, V., Serrano, M. J., and Simons, B. B. Register-sensitive selection, duplication, and sequencing of instructions. In *Proceedings of the 15th International Conference on Supercomputing* (New York, NY, USA, 2001), ICS '01, ACM.

[93] Sethi, R., and Ullman, J. D. The generation of optimal code for arithmetic expressions. *J. ACM 17*, 4 (Oct. 1970), 715–728.

[94] Silvera, R., Wang, J., Gao, G. R., and Govindarajan, R. A register pressure sensitive instruction scheduler for dynamic issue processors. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques* (Nov 1997).

[95] Smith, G. *Numerical Solution of Partial Differential Equations: Finite Difference Methods.* Oxford University Press, 2004.

[96] Smith, J. E., and Pleszkun, A. R. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.* (May 1988).

[97] Smith, M. D., Ramsey, N., and Holloway, G. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2004), PLDI '04, ACM, pp. 277–288.

[98] Stallman, R. M., and Community, G. D. *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3.* CreateSpace, Paramount, CA, 2009.

[99] Stock, K., Kong, M., Grosser, T., Pouchet, L.-N., Rastello, F., Ramanujam, J., and Sadayappan, P. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 65–76.

[100] Stone, J. E., Gohara, D., and Shi, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test 12*, 3 (May 2010), 66–73.

[101] Strzodka, R., Shaheen, M., Pajak, D., and Seidel, H.-P. Cache oblivious parallelograms in iterative stencil computations. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2010), ICS '10, ACM, pp. 49–59.

[102] Su, H., Cai, X., Wen, M., and Zhang, C. An analytical GPU performance model for 3D stencil computations from the angle of data traffic. *J. Supercomput. 71*, 7 (July 2015), 2433–2453.

[103] Suriana, P., Adams, A., and Kamil, S. Parallel associative reductions in halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Piscataway, NJ, USA, 2017), CGO '17, IEEE Press.

[104] Seismic Wave Modeling (SW4) - Computational Infrastructure for Geodynamics. `https://geodynamics.org/cig/software/sw4/`, 2014.

[105] Taflove, A. *Computational electrodynamics: The Finite-difference time-domain method*. Artech House, 1995.

[106] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2011), SPAA '11, ACM, pp. 117–128.

[107] Touati, S., and Eisenbeis, C. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters 14*, 2 (June 2004), 287–313.

[108] Unat, D., Cai, X., and Baden, S. B. Mint: Realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the International Conference on Supercomputing* (New York, NY, USA, 2011), ICS '11, ACM, pp. 214–224.

[109] Unkule, S., Shaltz, C., and Qasem, A. Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In *Proceedings of the 21st International Conference on Compiler Construction* (Berlin, Heidelberg, 2012), CC'12, Springer-Verlag, pp. 21–40.

[110] Valluri, M. G., and Govindarajan, R. Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors. In *1999 International Conference on Parallel Architectures and Compilation Techniques* (1999).

[111] Vasista, V., Narasimhan, K., Bhat, S., and Bondhugula, U. Optimizing geometric multigrid method computation using a dsl approach. In *Proceedings of the International Conference for High Performance Computing,*

Networking, Storage and Analysis (New York, NY, USA, 2017), SC '17, ACM, pp. 15:1–15:13.

[112] Venkatasubramanian, S., Vuduc, R. W., and none, n. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 244–255.

[113] Verdoolaege, S., Juega, J. C., Cohen, A., Gómez, J. I., Tenllado, C., and Catthoor, F. Polyhedral parallel code generation for CUDA. *ACM TACO 9*, 4 (Jan. 2013), 54:1–54:23.

[114] Vizitiu, A., Itu, L., Niţă, C., and Suciu, C. Optimized three-dimensional stencil computation on fermi and kepler GPUs. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)* (Sept 2014), pp. 1–6.

[115] Volkov, V., and Demmel, J. W. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008), SC '08, IEEE Press, pp. 31:1–31:11.

[116] Wahib, M., and Maruyama, N. Scalable kernel fusion for memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), SC '14, IEEE Press, pp. 191–202.

[117] Wahib, M., and Maruyama, N. Automated GPU kernel transformations in large-scale production stencil applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, ACM, pp. 259–270.

[118] Wahib, M., Maruyama, N., and Aoki, T. Daino: A high-level framework for parallel and efficient AMR on GPUs. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2016), pp. 621–632.

[119] Wang, H., Potluri, S., Luo, M., Singh, A. K., Sur, S., and Panda, D. K. Mvapich2-gpu: optimized gpu to gpu communication for infiniband clusters. *Computer Science - Research and Development 26* (Apr 2011).

[120] Wang, J., Krall, A., Ertl, M. A., and Eisenbeis, C. Software pipelining with register allocation and spilling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (New York, NY, USA, 1994), MICRO 27, ACM, pp. 95–99.

[121] Williams, S., Waterman, A., and Patterson, D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (Apr. 2009), 65–76.

[122] Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J., Roune, B., Springer, R., Weng, X., and Hundt, R. gpucc: An open-source GPGPU compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (2016), CGO '16, pp. 105–116.

[123] Xie, X., Liang, Y., Li, X., Wu, Y., Sun, G., Wang, T., and Fan, D. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture* (New York, NY, USA, 2015), MICRO-48, ACM, pp. 395–406.

[124] Xue, J. On tiling as a loop transformation. *Parallel Processing Letters 07*, 04 (1997), 409–424.

[125] Yount, C., Tobin, J., Breuer, A., and Duran, A. YASK-yet another stencil kernel: A framework for HPC stencil code-generation and tuning. In *Proceedings of the Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC* (Piscataway, NJ, USA, 2016), WOLFHPC '16, IEEE Press, pp. 30–39.

[126] Zhang, N., Driscoll, M., Markley, C., Williams, S., Basu, P., and Fox, A. Snowflake: A lightweight portable stencil DSL. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2017), pp. 795–804.

[127] Zhang, Y., and Mueller, F. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 155–164.

[128] Zhou, X., Giacalone, J.-P., Garzarán, M. J., Kuhn, R. H., Ni, Y., and Padua, D. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 207–218.