REDUCING IRREGULARITIES IN CONTROL FLOW AND MEMORY ACCESS ON GRAPHICS PROCESSING UNIT ARCHITECTURES

by

James Sokhom King

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2017

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **James Sokhom King**

has been approved by the following supervisory committee members:

Robert M. Kirby , Chair(s) **25 May 2016**
Date Approved

Christopher R. Johnson , Member **17 May 2016**
Date Approved

Mary Hall , Member **18 May 2016**
Date Approved

Ross T. Whitaker , Member **17 May 2016**
Date Approved

Spencer Sherwin , Member **10 June 2016**
Date Approved

by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

Memory access irregularities are a major bottleneck for bandwidth limited problems on Graphics Processing Unit (GPU) architectures. GPU memory systems are designed to allow consecutive memory accesses to be coalesced into a single memory access. Noncontiguous accesses within a parallel group of threads working in lock step may cause serialized memory transfers. Irregular algorithms may have data-dependent control flow and memory access, which requires runtime information to be evaluated. Compile time methods for evaluating parallelism, such as static dependence graphs, are not capable of evaluating irregular algorithms. The goals of this dissertation are to study irregularities within the context of unstructured mesh and sparse matrix problems, analyze the impact of vectorization widths on irregularities, and present data-centric methods that improve control flow and memory access irregularity within those contexts.

Reordering associative operations has often been exploited for performance gains in parallel algorithms. This dissertation presents a method for associative reordering of stencil computations over unstructured meshes that increases data reuse through caching. This novel parallelization scheme offers considerable speedups over standard methods.

Vectorization widths can have significant impact on performance in vectorized computations. Although the hardware vector width is generally fixed, the logical vector width used within a computation can range from one up to the width of the computation. Significant performance differences can occur due to thread scheduling and resource limitations. This dissertation analyzes the impact of vectorization widths on dense numerical computations such as 3D dG postprocessing.

It is difficult to efficiently perform dynamic updates on traditional sparse matrix formats. Explicitly controlling memory segmentation allows for in-place dynamic updates in sparse matrices. Dynamically updating the matrix without rebuilding or sorting greatly improves processing time and overall throughput. This dissertation presents a new sparse matrix format, dynamic compressed sparse row (DCSR), which allows for dynamic streaming updates to a sparse matrix. A new method for parallel sparse matrix-matrix multiplication (SpMM) that uses dynamic updates is also presented.

CONTENTS

CHAPTERS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Challenges

General purpose GPU (GPGPU) programming has become a popular choice for solving many computationally demanding problems in science and engineering [43]. GPUs are a form of single instruction multiple data (SIMD) accelerators. SIMD accelerators are many-core processors with both high memory bandwidth and peak floating point operations per second (FLOPS). There has been a great push in terms of research on how to best use these accelerator cards and how to optimize code for them [14, 60, 41]. Traditionally, programming for GPUs has been a significant challenge due to the architecture specific low-level optimizations required for high performance. GPUs outperform CPUs in terms of FLOP per dollar and FLOP per transistor [64].

A major challenge to achieving high throughput on GPU architectures is minimizing irregularities in control flow and memory access. SIMD accelerator cards achieve high throughput through massive parallelism. However, this parallelism can be disrupted by a number of factors. Groups of threads, known as vectors or warps, that operate synchronously together will stall when one or more of them has a divergent instruction. Also, the entire group must wait for loads/stores to complete, even if only one thread has made a request or if only one thread has a cache miss while the rest have cache hits. This programming paradigm is distinctly different from the *multiple instruction multiple data* (MIMD) paradigm that multicore chips follow.

Irregularities within an algorithm can lower performance by as much as an order of magnitude on SIMD architectures. Irregular algorithms typically operate over data structures such as trees, graphs, unstructured meshes, and priority queues. Parallelizing irregular algorithms is significantly more challenging, as is mapping them to SIMD architectures such as the GPU. Modeling the performance of irregular programs is often difficult because runtime behavior is dependent upon input data. GPUs are known to perform well on regular computations, but achieving high performance on irregular algorithms is still an ongoing area of research.

Much research has shown the efficacy of using GPU architectures to compute regular algorithms. However, many commonly used algorithms are irregular in nature and prove difficult to attain high performance. There are a number of key problem areas that are of high importance to computational science and engineering for which the boundaries of high performance computing (HPC) still need to be pushed. Some of these areas include dense linear algebra, sparse linear algebra, spectral methods, n-body problems, structured grids, unstructured grids, MapReduce, combinatorial logic, graph traversal, dynamic programming, back-track/branch and bound, graphical models, and finite state simulations [4]. A number of these problems are irregular or can be formulated in a way that involves irregular computations. Sparse computations often exhibit irregularities, as in the case of sparse matrix factorizations. Operations over unstructured meshes/grids are inherently irregular. MapReduce, combinatorial logic, graph traversal, dynamic programming, and finite state simulation often exhibit irregular computations as well.

1.2 Thesis Goals

This dissertation targets a number of goals related to control flow and memory access irregularities on GPU architectures:

- Study and analyze irregularities within the context of unstructured mesh and sparse matrix problems on GPU architectures.
- Analyze the impact of vectorization widths on control flow and memory access irregularities.
- Provide data-centric techniques for addressing irregularities with several unstructured mesh and sparse matrix problems.

Both unstructured meshes and sparse matrices involve a level of indirection to access data. This indirection often leads to higher levels of irregularity within control flow and memory access. This dissertation explores two primary applications, namely that of stencil computations over unstructured meshes and dynamic insertion within sparse matrices. Both of these problems involve high degrees of irregularity due to indirection of memory access within the data structures.

1.2.1 Unstructured Meshes

A primary application studied in this work was dG postprocessing which involves computing the convolution of a B-spline kernel and the underlying dG finite element method (FEM) solution defined over a geometric mesh. When computing these convolutions over unstructured meshes, there is a considerable amount of irregularity due to the need to dynamically compute intersections based on the B-spline stencil/mesh overlap.

Associativity of operations can often be exploited for reordering operations to enhance parallelization. Associative operations like addition and multiplication can be reordered, and this forms the basis for core parallel algorithms like prefix-sum [28]. Improved data reuse through associative reordering is particularly useful for stencil computations. Stencils are a key computational pattern used in many numerical methods and algorithms. Stencil computations sample information from a localized region. Through reordering and caching of results, data reuse can improve performance.

DG postprocessing involves convolving a B-spline stencil with a grid of points is defined over the mesh that correspond to the numerical quadrature points of the solution. The geometry of this grid depends upon the mesh's geometric structure. Structured meshes will lead to regular grid patterns, while unstructured grids will lead to irregular grid patterns. The regular access pattern used by structured grids generally leads to contiguous memory accesses, good memory layout patterns, and high cache efficiency. Efficient computation of stencil operations over structured meshes has been widely studied, and great gains have been made by exploiting parallelism and data locality. Stencil computations performed over unstructured grids is generally much harder than those performed over structure grids, and they often exhibit noncontiguous memory access patterns and lower cache efficiency.

One of the biggest challenges in computing stencil operations over unstructured meshes is efficiently sampling the underlying mesh in the mesh/stencil intersection. DG postprocessing requires performing stencil operations that sample information from the neighborhood of mesh elements within the intersection of the stencil and the mesh. In this case, the geometric nature of the mesh has a significant impact on cache efficiency and data locality, especially for many-core architectures.

This dissertation presents an associative reordering transformation for evaluating stencil computations over meshes. Computations are reordered by geometric element instead of by quadrature point. This provides improved data reuse through caching of the polynomial coefficients associated with the element being processed.

1.2.2 Tuning Vectorization Widths

An extension of the dG postprocessing work involved looking at the problem in 3D. The inner integral computation is much denser in 3D than in the 2D case. A higher level of data parallelism is required through fine grained parallelization of the inner integral to achieve high performance. This dissertation explores the tuning of vectorization widths for the dense integral computations involved in 3D dG postprocessing over tetrahedral meshes.

Vectorization is the process by which a scalar operation that operates over a pair of operands is converted to operate over a vector of pairs of operands (a series of adjacent values). Vectorized instructions operate over multiple pairs of data in parallel. Performance is improved through this added level of parallelism and increased memory efficiency. Vectorization is one of the primary design aspects used in SIMD architectures that allows modern GPUs to achieve beyond teraflop level performance.

Vector instruction widths typically range from 2 to 64 (2, 4, 8, 16, 32, or 64 adjacent data elements). Vectorization of an algorithm can lead to significant performance improvements in some cases. Streaming SIMD extensions (SSE) is an example of a vector instruction set designed for the x86 architecture that has seen wide adoption. Modern GPUs group cores into work units (32 or 64 cores typically) that operate in lock step like a vector processor.

Although the vector width of the hardware is generally fixed, the logical vector width used by the programmer within a computation can range from 1 up to the width of the computation. This is done by decreasing the vector width and increasing the number of concurrent vectors. For example, with a hardware vector width of 8, one could compute: 1 operation of width 8, 2 operations of width 4, 4 operations of width 2, or 8 operations of width 1. Adjusting this logical vector width in conjunction with the number of concurrent vectors can have significant impact on overall performance and throughput.

Achieving high performance with complex code bases that involve numerous architecture specific parameters remains a difficult task. Often, tedious manual optimizations are required. This process is time consuming for the programmer, and manual tuning rarely yields the optimal parameter configuration. Autotuning is a method by which optimal or near optimal run-time configuration parameters are selected through an automated testing process [65]. This technique has proven to be a valuable tool for improving performance and increasing programmer efficiency.

This dissertation presents an analysis of the impact of vectorization widths applied to 3D dG postprocessing over tetrahedral meshes. A generalized method for tuning the logical vector width or "stride" width is presented. Both thread divergence within a logical vector of threads and memory access patterns are affected by vectorization widths. This application dependent tuning parameter has significant impact on the performance of dense computations.

1.2.3 Sparse Matrices

Sparse matrix-vector multiplication (SpMV) is the workhorse operation of many numerical simulations, and has seen use in a wide variety of areas such as data mining [39] and graph analytics [32]. Frequently in these algorithms, a majority of the total processing is spent on SpMV operations. Iterative computations such as the power method and conjugate gradient are commonly used in numerical simulations, and require successive SpMV operations [74]. The use of GPUs has become increasingly common in computing these operations as they are, in principle, highly parallelizable. GPUs have both a high computational throughput and a high memory bandwidth. Operations on sparse matrices are generally memory bound; this makes the GPU a good target platform due to its higher memory bandwidth compared to that of the CPU, however it is still difficult to attain high performance with sparse matrices because of thread divergence and noncoalesced memory accesses.

Some applications require dynamic updates to the matrix; generally construed, updates may include inserting or deleting entries. Fully compressed formats such as compressed sparse row (CSR) cannot handle these operations without rebuilding the entire matrix. Rebuilding the matrix is orders of magnitude more costly than performing an SpMV operation. The ellpack (ELL) format allocates a fixed amount of space for each row, allowing fast insertion of new entries and fast SpMV but limits each row to a predetermined number of entries and can be highly memory inefficient. The coordinate (COO) format stores a list of entries and permits both efficient memory use and fast dynamic updates but is unordered and slow to perform SpMV operations. The hybrid-ellpack (HYB) format attempts a compromise between these by combining an ELL matrix with a COO matrix for overflow; where rows may require examination of this overflow matrix, however, SpMV efficiency suffers.

Matrix representations of sparse graphs sometimes exhibit a power-law distribution (when the number of nodes with a given number of edges scales as a power of the number of edges). This results in a long tail distribution in which a few rows have a relatively high number of entries while the rest have a relatively low number. There are important real-world phenomena which exhibit a power-law distribution. Their corresponding matrices can represent things such as adjacency graphs, web communication, and finite-state simulations. Such a matrix is also the pathological case for memory efficiency in the ELL format and requires significant use of the COO portion of a HYB matrix making neither particularly well suited for dynamic sparse-graph applications.

This dissertation presents a new sparse matrix format, *dynamic compressed sparse row* (DCSR), that allows for efficient dynamic updates, exhibits easy conversion with standard CSR, and has fast SpMV. A method for conversion between CSR and DCSR is given along with an efficient method for defragmentation of the format that does not require sorting. Detailed benchmarks of SpMV and insertion operations across a suite of sparse-graph benchmarks are provided.

Computing sparse matrix-matrix multiplication in parallel is a difficult task to perform efficiently. Given an $m \times k$ matrix A and $k \times n$ matrix B, the goal is to compute the $m \times n$ matrix C ($AB = C$). Unlike in the dense case, it is inefficient to multiply a row of A by a column of B, since many of the entries will be zeros. A more efficient method is to compute the set of partial products formed by multiplying each nonzero a_{ij} in A by each nonzero b_{jk} in row j of B. This method produces all of the needed results without wasted work checking against zero values. The fill-in of the resulting C matrix depends on the sparsity patterns of A and B and is unknown until computation is performed.

The traditional parallel algorithm for computing SpMM involves computing all of the partial products, sorting them by row and column, and then performing a segmented reduction. The matrix is converted to COO format to perform these operations and converted back to its original format afterward. The sort and reduction are performed in global memory. The conversions to and from the COO format make this a costly operation to perform.

This dissertation presents an improved SpMM algorithm that asynchronously computes the partial products, sorts, and reduces the results on a per row basis. The results are then dynamically inserted into the resulting C matrix in DCSR format. This provides a considerable performance improvement for rows that can be computed with the shared memory capacity of a streaming multiprocessor (SM).

CHAPTER 2

BACKGROUND

2.1 GPU Architectures

Latency-oriented devices have plateaued in performance due largely to power and thermal constraints. The push towards increased performance, particularly exascale and beyond, has led to the use of throughput-oriented “many-core” devices [89]. Examples of many-core devices are GPUs and the accelerator cards such as the Intel Xeon Phi. GPUs are now commonly employed as accelerator cards for general purpose (GPGPU) programming, and many of the fastest super computers in the world employ SIMD accelerators [82]. Many-core processors are often designed with the *single instruction multiple data* (SIMD) paradigm, which allows engineers to reduce overhead in terms of power and area per core.

GPUs are massively parallel throughput-oriented devices. They provide high throughput by hiding memory latency through context switching between thread workgroups. Modern GPUs operate with a kernel/block/thread programming model. Kernels are user-defined functions that execute across M blocks in parallel, with each block executing N threads in parallel. Each block will be assigned to a physical streaming multiprocessor (SM). There can be more blocks assigned to execute than physical SMs, since execution contexts will be switched between blocks when a previous block finishes. Threads are grouped together into execution units known as warps or wavefronts (typically 32 or 64 threads wide). These warps execute in lock step, computing the same instruction using differing data operands. If threads diverge, their execution will be serialized until they converge again. Quite possibly the single most important factor in attaining high performance with SIMD architectures is to achieve coalesced memory accesses. Loads and stores by threads within a vector can be coalesced into as few as a single memory transaction. This coalescing occurs when all memory references within a vector fall into a single cache line [61]

Each SM on the GPU has a limited amount of register space and shared memory/cache for threads operating within a block. L1 cache and shared memory share the same memory space, and the ratio of L1 cache to shared memory can be configured at compile time. Threads within a block can pass information through this shared memory space, but threads between blocks must pass information through global memory. Global memory is device memory shared between all SMs that can be accessed by any thread, albeit at a much higher latency than cache/shared memory. Synchronization can be achieved between threads within a logical block, but, in general, there is no way to synchronize threads across blocks. The low-level architectural model of the GPU presents a challenge in writing efficient programs. The Compute Unified Device Architecture (CUDA) programming model [61, 62] and the Open Computing Language (OpenCL) [41] have made strides towards lowering the barrier of programming GPUs. Significant work has been devoted to the goal of achieving high performance from vectorized operations on streaming SIMD architectures [37, 26, 47, 71].

Modern SIMD accelerator cards rely on a separate device memory pool that is located on the card for increased memory bandwidth. This memory sepearation has led to data transfer between the host and device to becoming the major bottleneck in heterogeneous systems. Although recent work has experimented with combining CPUs and GPUs on the same chip [25], the technology is not fully mature. Lower transistor overhead per core allows for greater efficiency in terms of area/performance and energy/performance. This core efficiency combined with higher memory bandwidth allows for much higher throughput than traditional latency-oriented chips.

2.2 GPU Programming Model

A major research focus has been parallelizing nongraphics applications on GPUs, known as GPGPU (general purpose GPU) computing [63]. Knowing the operational intensity of an algorithm will help one effectively parallelize it. Operational intensity is often defined by the FLOPS/byte ratio of an algorithm [95]. SIMD architectures achieve peak performance when the number of floating point operations per second (FLOPS) is high, and the number of read/write memory accesses is low, yielding a high FLOPS/byte ratio [31]. The FLOPS/byte ratio has a significant impact on performance when algorithms are parallelized on multicore/many-core architectures. The performance of algorithms on many-core architectures is greatly improved when there is a high FLOPS/byte ratio and low branch divergence in the algorithm. Two factors that lower SIMD efficiency are thread divergence in instructions and thread divergence in memory accesses, often termed control flow irregularity and memory access irregularity, respectively [18]. Minimizing these factors is key to achieving high performance on SIMD architectures.

The GPU is designed to provide parallelism through a heirachy of abstractions. Task level parallelism (kernels), coarse-grained parallelism (blocks), and fine-grained parallelism (threads). At each level there is potential to exploit parallelism. Mapping problems to the GPU involves a problem decomposition into computable elements. The size of the computable elements often determines a good mapping to the GPU. These elements are then mapped to threads, vectors (groups of multiple threads working together), or blocks.

The thread-based execution model (Figure 2.1(a)) implements each GPU thread as an execution unit for a single element computation. Because the number of threads equals the number of elements, the thread-based execution model scales well with the number of elements. However, as the resources per thread (e.g., number of registers, amount of shared memory) are typically small on most current GPUs, this strategy is suitable only to “tiny” problem sizes or with algorithms that are relatively memoryless (such as per thread reduction).

The vector-based execution model (Figure 2.1(b)) uses a group of threads within a block

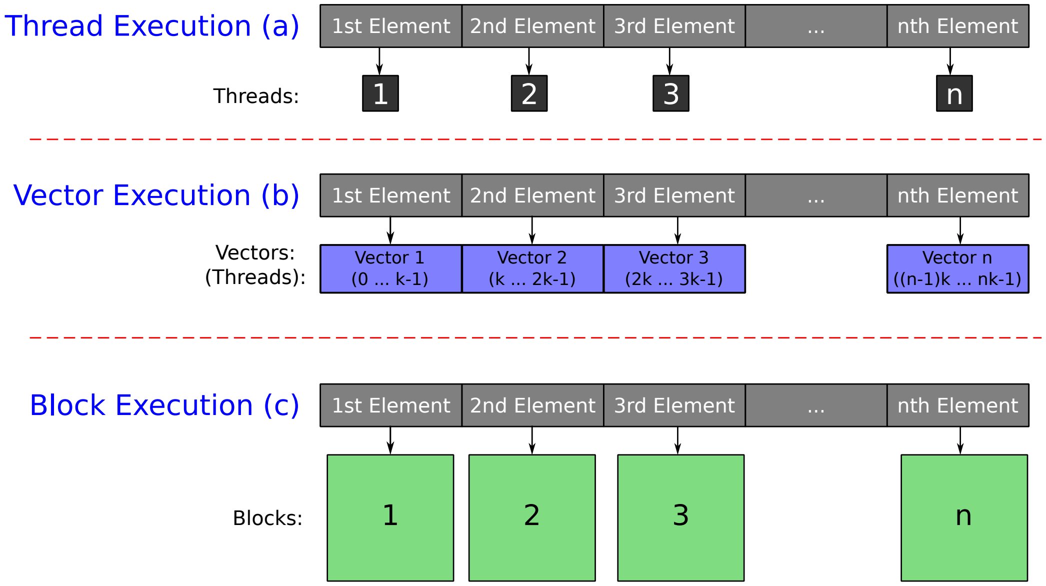


Figure 2.1: Parallel execution mapping for multielement processing using thread-based, vector-based, and block-based execution models.

as the execution unit. Each vector is composed of k threads. Multiple vectors (2 or more) of threads exist within a block. The vector size is often chosen to be between 2 and the wavefront size, to take advantage of implicit synchronization within a wavefront. Threads within a wavefront work in lock step and do not require explicit synchronization. However, if a vector size is chosen to be larger than the wavefront size, explicit synchronization will be required. Since threads within a vector will be part of the same block, they can share resources (i.e., shared memory). This strategy is suitable for small to medium size elements.

The block-based execution model (Figure 2.1(c)) employs each execution block as the execution unit. In this case, the number of execution blocks equals the number of elements. All threads of a single execution block work together to complete tasks for a single element. Barriers are normally required to synchronize between a block's threads, and a scratch memory space (i.e., GPU shared memory) is used to collaborate results. The granularity of the execution blocks controls the amount of resources available for each element's computation. As the resources are allocated per block, this strategy can handle a wide range of the inputs.

Figure 2.2 illustrates the process of mapping from the problem domain to the execution domain. When a vector-based execution model is used with the vector size being equal to the wavefront size (or SIMD-width), there is no synchronization required as all member threads execute in lock step. This is the most reasonable solution for the case when the element size is small to medium (i.e., able to fit within shared memory). When the execution model is a computational grid, synchronization is required to collaborate results between blocks.

2.2.1 Multielement data structures

Data structures are crucial for determining performance. Memory coalescing is one of the most important factors in attaining high performance. This condition depends solely on the access pattern of neighboring threads, encouraging neighbor threads to access continuous data. For the thread-based execution model, this turns out to be the interleaving data structure (see Figure 2.3(a)) where the first component of the data of the first element is laid out in the memory next to the first component of the second element and so on. For the block-based execution model, it becomes the concatenated data structure (Figure 2.3(b))

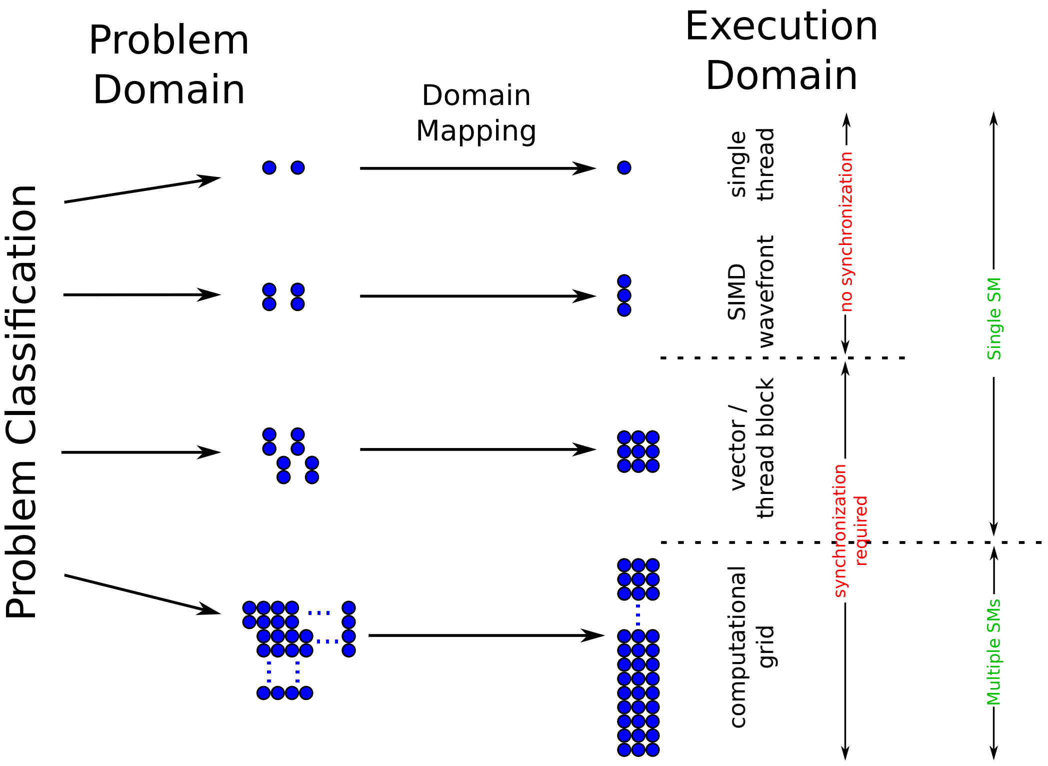


Figure 2.2: Problem classification for algorithm mapping

that lays out the data structure of each element sequentially in memory.

The interleaving and concatenating data structures are well known as the Structure of Arrays (SoA) and Array of Structures (AoS), respectively. SoA is generally preferred for single element GPU algorithms, whereas the AoS format is typically used for other mappings as it can handle a much wider data input range. Transpose operations can be used to switch between AoS and SoA data formats (Figure 2.3).

To achieve the highest bandwidth efficiency, data padding is sometimes used to guarantee the access of a thread/block starting at an aligned memory address. Data alignment can be employed per element with the AoS and data alignment per array with the SoA (see Figure 2.3). This strategy has proven to be simple yet effective and compact. This alignment strategy increases the storage by approximately 10% in contrast to the alignment per data dimension strategy, which can be very expensive with high-dimensional input data.

An additional benefit of data padding is that even though data might have odd size numbers in terms of number of elements and/or element size, the transpose function always achieves the highest memory bandwidth efficiency because execution blocks always access

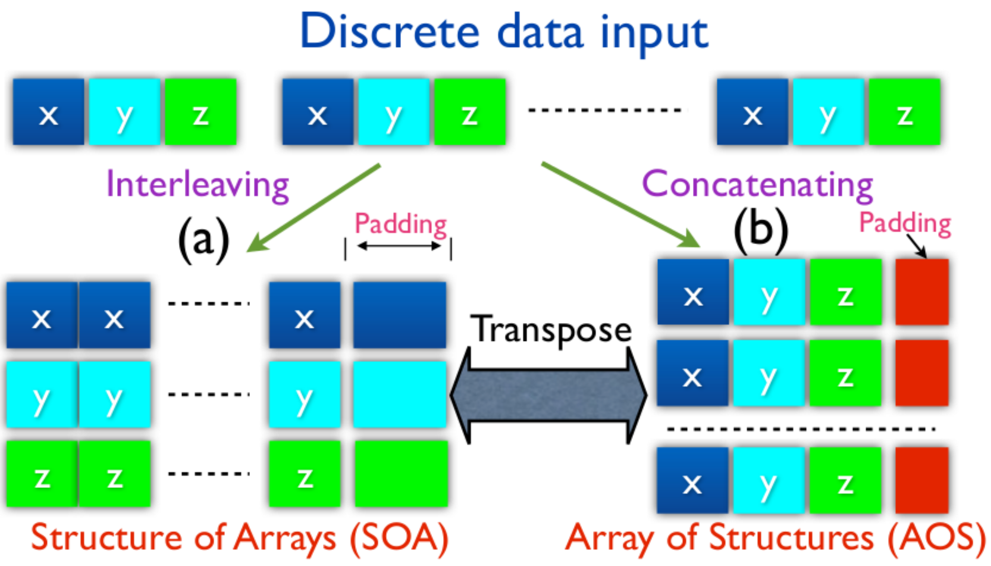


Figure 2.3: Common multielement data formats. Examples of (a) structure of arrays and (b) array of structures.

aligned memory for both data loading and storing. Hence, the added overhead due to the changing of data structures is minimized.

2.2.2 Block/Thread Configuration

Kernel configuration (number of threads, dimensions of thread blocks, number of blocks, and dimensions of compute grid) determines the parallelism granularity and has significant influence on performance, and a tuning strategy is often required to maximize performance [90]. It is important to start with a good estimation as the configuration space is large and displays nonlinear behavior. A good estimator must satisfy two conditions: it scales well across platforms and it is adapted to the problem size. The estimator has to take into account the hardware configuration of the running system (i.e., number of registers, size of the shared memory) and the kernel information (i.e., number of threads, shared memory usage and problem size). Even though high occupancy indicates good parallelism efficiency, it does not necessarily directly correspond to high performance [90].

A common configuration approach is to start with the minimal block size of one wavefront and increase in increments of a wavefront size each time until the occupancy requirement is met. This method is the reverse of the maximal block size strategy employed by

CUDA Thrust, which starts the search with the maximum block size. This strategy prevents idle threads from being generated which frees up more computational resources for working threads.

2.2.3 Batched Operations

Batch processing is the act of grouping some number of like tasks and computing them as a “batch” in parallel, which generally involves a large set of data whose elements can be processed independently of each other. Batch processing eliminates much of the overhead of iterative nonbatched operations. “Batch” processing is well suited to GPUs due to the SIMD architecture, which allows for high parallelization of large streams of data. Basic linear algebra subprograms (BLAS) are a common example of large-scale operations that benefit significantly from batch processing. The HDG method specifically benefits from batched BLAS Level 2 (matrix-vector multiplication) and BLAS Level 3 (matrix-matrix multiplication) operations.

Finding efficient implementations for solving linear algebra problems is one of the most active areas of research in GPU computing. The NVIDIA CUBLAS [62] and AMD APPML [24] are well-known solutions for BLAS functions on GPUs. CUBLAS is specifically designed for the NVIDA GPU architecture based on CUDA [61], and the AMD solution using OpenCL [7] is a more general cross platform solution for both GPU and multiCPU architectures. CUBLAS has constantly improved based on a successive number of research attempts by Volkov [90], Dongarra [78, 1], and others. This research has led to a speed improvement of one to two orders of magnitude for many functions from the first release version till now. In recent releases, CUBLAS and other similar packages have been providing batch processing support to improve processing efficiency on multielement processing tasks. The support is, however, not complete as currently CUBLAS supports batch mode processing only for BLAS Level 3, but not for functions within BLAS Level 1 and BLAS Level 2.

2.3 Analysis of Parallel Algorithms

The work by Pingali et al. provides a fundamental framework for the analysis of parallel algorithms [67]. It classifies parallel algorithms by their data topology, location, and ordering, of active data nodes, and the type of operations that are performed on the data. Figure 2.4 illustrates the structural analysis of algorithms using this methodology. Structural analysis of algorithms classifies them based on three defining characteristics, topology, location and ordering of active nodes, and the type of operator being applied to the data.

- The topology is defined as the relational structure of the data. The relationship of the problem data can be converted to a graph. A set or multiset is isomorphic to a graph with no edges. An ordered set could be defined as a graph with edges defining the ordering. Similarly matrices can be viewed as graphs. With this graph formulation, the topology of the problem can be viewed in terms of its structure.

  Structured: Topologies can be defined by a regular pattern. If the items are ordered, they are isomorphic to a sequence or stream. Dense matrices are defined by their width and height and have a rectangular pattern.

  semistructured: Topologies, like trees, exhibit some structural invariants (like the acyclic nature of trees), but do not follow a regular pattern.

  Unstructured: Topologies, such as general graphs, have no defining patterns.

- Active nodes are classified by the order and location within the computation.

  Ordering: The active nodes can be either ordered or unordered.

  Location: Active nodes can be determined based on the topology of the data (topology-driven) or based upon the values of the data (data-driven). An example of topology

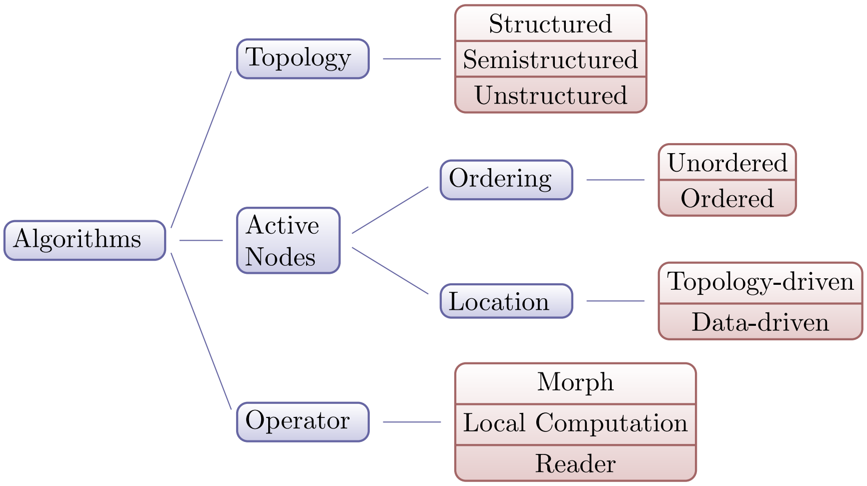


Figure 2.4: Structural analysis of algorithms.

driven locality is BLAS operations, such as matrix-matrix multiplication. In this case, the active nodes used in computing an output datum are explicitly defined based on their position in the matrix. An example of data-driven locality is ray-tracing or maxflow computations. In those cases, nodes become active based on the values of previous data computations. Naturally, data-driven locality is more unpredictable in its active node patterns since it is data dependent.

- Operators are classified based on how they modify the data.

  Morph: A morph operator alters the topology of the data structure. It may also alter the values of the nodes or edges as well, which allows for the insertion or deletion of nodes within the graph.

  Local: Operators that alter the data values of nodes but do not alter the topology are considered local operators. An example is iterative operations that update the value of the solution at each step and use that newly computed value in the next step.

  Reader: An operator is a reader if it only reads the data structure without altering topology or values. Reader operations are idempotent.

Certain characteristics of this structural analysis of algorithms lead to high irregularities. Regular algorithms typically have a structured topology, and the active nodes are defined in a topology-driven manner. Unstructured topologies lead to greater irregularities than in the semistructured or structured cases. Data-driven locations for active nodes also commonly lead to irregular control flow and memory access patterns. The operator being performed has less impact on regularity than topology and location/ordering of active nodes, although morph operators tend to have higher irregularity than that of local and reader operators. This increased irregularity is because morph operators alter the topology of the data while local computation operators alter only the data and reader operators do not modify either.

2.3.1 Regular Versus Irregular Algorithms

Compiler literature often references the terms “regular” and “irregular” when classifying types of code. Regular algorithms have no data-dependent control flow or memory accesses. Dense BLAS operations are an example of regular computation. The number and order of operations have no data dependencies, and the size and dimensions of the resulting output are defined by the width and height of the operands. Burtcher et al. provide an overview of an analysis of control flow and memory access irregularity for a variety of applications on the GPU [18].

The behavior of irregular code cannot be statically predicted a priori. Irregular code contains control flow and memory references that may be data-dependent. This degree to which the amount of control flow and memory references out of the total are data-dependent defines the degree of irregularity. The data-dependent nature of computations in irregular algorithms makes it difficult to predict the number, order, and location of operations. Irregular algorithms include a broad range of problems, such as n-body simulations [8], data mining [85], Boolean satisfiability [17], social networks [36], system modeling [66], compilers [2], meshing [20], and discrete-event simulation [54]. Graph applications, in particular, are often irregular. Graph traversal is generally highly irregular due to the connectivity of the graph and the fact that many operations are based on the values of the nodes and edges.

2.3.2 Metrics of Irregularity

Data locality plays an important role in performance of both the CPU and GPU. The GPU, in particular, is affected by divergent branches within a SIMD vector and memory stores/loads that are uncoalesced or cause bank conflicts within shared memory. The following two metrics to define the level of irregularity in a program on a SIMD architecture provided by [18].

- control-flow irregularity (CFI) $= \dfrac{\text{divergent-branches}}{\text{executed-instructions}}$
- memory-access irregularity (MAI) $= \dfrac{\text{replayed-instructions}}{\text{issued-instructions}}$

Control flow irregularity defines the degree to which vectorized threads diverge at branches. It is calculated as the ratio of divergent branches to total number of executed instructions. This value is typically low in most programs, since the number of branches is a small fraction of the total instructions, and the number of divergent branches is a subset of the number of branches. Memory access irregularity defines the amount of issues that are replayed out of the total issued. This metric corresponds to the percent of noncoalesced memory accesses that occur in the code. With fully coalesced memory accesses, the number of replayed instructions will be zero (i.e., only one memory transaction will be issued per warp/vector for a given store/load). Each metric ranges between 0% and 100%. SIMD architectures can achieve near peak performance when the CFI and MAI values are near 0%. Problems such as these are considered to be "embarrassingly parallel." Irregular applications are generally data-dependent, and these metrics can have high variability depending on program input.

2.3.3 Data-Driven Approaches

- **Worklist:** Worklists are used in data-driven problems to track active elements. The worklist is initialized with a starting set of graph elements. Each thread will extract an element from the worklist and process it. From the algorithm specific processing, a new set of graph elements will be produced. This new set of elements will be appended to the current worklist. This step is often implemented using two buffer sets, an input set and an output set. Elements are read in a processed from the input set, and the new graph elements are written to the output buffer [59].

- **Hierarchical Worklist:** A common optimization on GPUs is to use hierarchical worklists. Hierarchical worklists takes advantage of the memory hierarchy on the GPU by using shared memory to store local worklists. Threads read from and write to the local worklists. The local worklists can be partitioned across threads to allow for lock-free updates. The use of on-chip shared memory generally provides significant performance improvements.

- **Work Chunking:** Work chunking is used in combination with hierarchical worklists to improve memory efficiency. Threads batch read and write operations between the local worklists and global memory, which reduces overall memory bandwidth. Updating a list in global memory requires atomic operations to ensure synchronization. Batching updates allows a single atomic operation to update the global worklist for one set of batch elements.

- **Atomic-Free Worklist Update:** Prefix-sum computations can be used to calculate direct offsets into the worklist where results should be written. Each thread records the number of elements it needs to write to the global worklist. A hierarchical scan operation can be performed over these batch sizes in $\log n$ steps (where n is the number of threads). This operation will produce a set of indices where i^{th} index denotes the starting index where thread i can start writing its elements. A global barrier is required after all threads record their batch sizes, before the prefix sum operation can be computed. This approach can provide significant performance improvement through replacing fine-grained synchronization with atomics by coarse-grained synchronization using barriers [58].

- **Work Donating/Stealing:** Load imbalance is a common problem when parallelizing irregular algorithms. When the number of new work elements generated at each step is data-dependent, it cannot be statically predicted. In this case, dynamic load balancing is required. The two most common methods for load balancing are work stealing and work donating. In work stealing, idle threads look for work to take from threads that have excess amounts. In work donating, threads with excess amounts of work give work to idle threads. Work donating has a better memory footprint on GPUs, so it is generally favored.

- **Variable Kernel Configuration:** Worklist sizes often vary considerably during the coarse of an irregular algorithm, which naturally leads to adapting kernel configurations to match the amount of work. Smaller worklists can be assigned fewer threads and increasing the amount of threads for larger worklists. Modern GPU architectures support dynamic parallelism which allows kernels to instantiate other kernel calls. This feature can be used to call kernels that are specifically configured to match certain parameters, such as worklist size and memory footprint.

2.3.4 Topology-Driven Approaches

In topology-driven algorithms, the threads are parallelized across the nodes of the data structure and not the data itself. In these algorithms, activities do not create new nodes. In cases such as breadth first search (BFS) and single source shortest path (SSSP), information can be freely updated without the need for atomic operations.

- **Kernel Unrolling:** Kernel unrolling is a method that combines multiple iterations of an activity together. This unrolling reduces the per-iteration overhead and allows for information to quickly propagate across the graph.

- **Exploiting Shared Memory:** Storing data locally with shared memory can greatly speed up computations on the GPU. The thread block size plays an important role in the amount of available shared memory per thread. The more threads per block, the less shared memory will be available to each thread.

- **Optimized Memory Layout:** Optimizing the layout of nodes within the data structure can improve memory locality. This optimization is sometimes performed by scanning over the nodes and ensuring that nodes that are logical neighbors within the computation lie next to each other within memory. Other algorithm-specific data reordering methods have also seen use [99].

2.3.5 Hybrid Approaches

- **Temporal Hybrid:** Temporal hybrid methods combine data-driven and topology-driven approaches, using each at different stages of the computation. For example, in BFS a data-driven approach will be more efficient when the active working set is small, which will typically be the beginning and ending iterations. When the working set of active nodes grows to a certain threshold (the middle iterations), it will become more efficient to use a topological-driven approach, which avoids atomic updates.

- **Spatial Hybrid:** Spatial hybrid methods partition the nodes within the graph into patches. Patches are assigned representative nodes. These nodes are inserted into a worklist when there is work to be done for that patch. When a patch is processed, all nodes within that patch will process any required work. The patches are managed in a data-driven manner, and nodes within a patch are managed in a topological-driven manner.

2.3.6 Software Solutions

There have been a number of proposed software solutions for reducing control flow and memory access irregularities within SIMD programs [99]. Within parallel vectors of threads, control flow and memory access patterns are defined by functions of thread and block identifiers. By altering these functions, different mappings can be chosen, which will potentially reduce irregularities. Two prominent methods for reducing irregularities are data reordering and job swapping.

Data reordering is a method in which the memory references are reordered to improve memory coalescing, while maintaining the original mapping between threads and data values. This reordering can be done through the use of a redirection array $A[tid] \rightarrow A'[P[tid]]$. This optimization is sometimes advantageous, as it can be implemented to improve memory accesses within a data structure, while preserving the original logic of the algorithm. Job swapping is a method by which threads compute alternate tasks than in the original mapping. Job swapping can be accomplished through reference redirection $A[tid] \rightarrow A[P[tid]]$ or through data relocation $A[tid] \rightarrow A'[tid]$. Data relocation permutes the values in memory such that the mapping between threads and data values is not preserved. Optimizations of this sort require altering the logic of the algorithm to account for the permutation in the data. A combination of data reordering and job swapping can potentially reduce the irregularities in control flow and memory access more than either one alone.

Other research has shown that control flow and memory access divergence can be reduced by partitioning GPU kernels into sections designed to be executed by differing warps using fine grain synchronization within a block [10]. This warp specialization allows for interwarp divergence where each warp can dynamically execute different code, while preserving intrawarp coalescing. This method can be useful when kernels cannot be easily broken up into separate components due having large numbers of temporary calculations that must be stored between phases.

2.3.7 Hardware Solutions

Hardware solutions have also been proposed to help reduce control flow and memory access irregularities. One such hardware mechanism, *diverge on miss*, allows threads within a warp to execute out of synchronization when divergent memory accesses occur. Tarjan et al. demonstrated diverge on miss through SIMD architecture simulation [87]. Prefetching is a common technique for reducing memory access latency by queuing up memory fetches before the data values are required rather than at instruction execution time. Recent work has proposed hardware additions that dynamically track memory references and automatically prefetch based on detected patterns [76].

2.4 Discontinuous Galerkin Postprocessing

The discontinuous Galerkin (dG) method has quickly found utility in such diverse applications as computational solid mechanics, fluid mechanics, acoustics, and electromagnetics. It allows for a dual path to convergence through both elemental h and polynomial p refinement. Moreover, unlike classic continuous Galerkin FEM, which seeks approximations that are piecewise continuous, the dG methodology merely requires weak constraints on the fluxes between elements. This feature provides a flexibility that is difficult to match with conventional continuous Galerkin methods. However, discontinuity between element interfaces can be problematic during postprocessing, where there is often an implicit assumption that the field upon which the postprocessing methodology is acting is smooth. A class of postprocessing techniques was introduced in [22, 23], with an application to uniform quadrilateral meshes, as a means of gaining increased accuracy from dG solutions by performing convolution of a spline-based kernel against the dG field. As a natural consequence of convolution, these filters also increased the smoothness of the output solution. Building upon these concepts, smoothness-increasing accuracy-conserving (SIAC) filters were proposed in [79, 92] as a means of introducing continuity at element interfaces while maintaining the order of accuracy of the original input dG solution.

The postprocessor itself is simply the discontinuous Galerkin solution u convolved against a linear combination of B-splines. That is, in one-dimension,

$$u^{\star}(x) = \frac{1}{h}\int_{-\infty}^{\infty} K^{r+1,k+1}\left(\frac{y-x}{h}\right) u(y)dy,$$

where $u^{\star}$ is the postprocessed solution, h is the characteristic element length (elements are line segments in 1D) and

$$K^{r+1,k+1}(x) = \sum_{\gamma=0}^{r} c_{\gamma}^{r+1,k+1}\psi^{(k+1)}(x-x_{\gamma}),$$

is the convolution kernel, which is referred to as the convolution stencil. $\psi^{(k+1)}$ is the B-spline of order $k+1$ and $c_{\gamma}^{r+1,k+1}$ represent the stencil coefficients. The term r is the upper bound on the polynomial degree that the B-splines are capable of reproducing through convolution. The stencil width increases proportionately with r. x_{γ} represent the positions of the stencil nodes and are defined by $x_{\gamma} = -\frac{r}{2} + \gamma, \;\; \gamma = 0, \cdots, r$, where $r = 2k$. This pattern will form a line and a square lattice of regularly spaced stencil nodes in 1D and 2D, respectively.

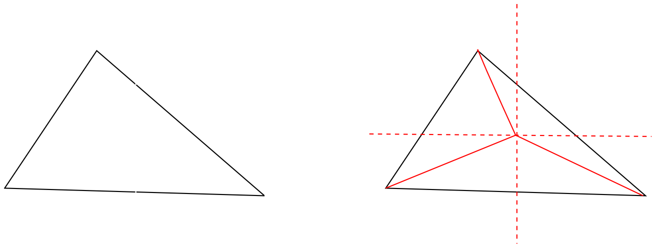
The postprocessor takes as input an array of the polynomial modes used in the discontinuous Galerkin method and produces the values of the postprocessed solution at a set of specified grid points. These grid points are chosen to correspond with specific quadrature points that can be used at the end of the simulation for error calculations. Postprocessing of the entire domain is obtained by repeating the same procedure for all the grid points. In two dimensions, the convolution stencil is the tensor product of 1D kernels. Therefore, the postprocessed solution at $(x, y) \in T_i$, becomes

$$u^\star(x, y) = \frac{1}{h^2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} K\left(\frac{x_1 - x}{h}\right) K\left(\frac{x_2 - y}{h}\right) u(x_1, x_2) dx_1 dx_2 \tag{2.1}$$

where T_i is a triangular element, u is the approximate dG solution, and the 2D coordinate system is denoted as (x_1, x_2).

To calculate the integral involved in the postprocessed solution in Equation (2.1) exactly, the triangular elements that intersect the stencil support are decomposed into subelements that respect the stencil nodes. The resulting integral is calculated as the summation of the integrals over each subelement. Figure 2.5 depicts a possible decomposition of a triangular element based on the stencil-mesh intersection.

As demonstrated in Figure 2.5(b), the triangular region is divided into subregions over which there is no break in regularity. These subregions are then triangulated for ease of implementation. The infinite integrals in Equation (2.1) may be transformed to finite local sums over elements, using the compact support property of the stencil ($T_j \in Supp\{K\}$). The extent of the stencil or $Supp\{K\}$ is given by $(3k + 1)h$ in each direction, where k is the degree of the polynomial approximation. Each of the integrals over a triangle T_j then becomes



(a) Triangular element (b) Integration regions

Figure 2.5: Demonstration of integration regions resulting from the stencil/mesh intersection. Dashed lines represent the breaks between stencil nodes. Solid red lines represent a triangulation of the integration regions.

$$\int\int_{T_j} K\left(\frac{x_1 - x}{h}\right) K\left(\frac{x_2 - y}{h}\right) u(x_1, x_2) dx_1 dx_2$$
$$= \sum_{n=0}^{N} \int\int_{\tau_n} K\left(\frac{x_1 - x}{h}\right) K\left(\frac{x_2 - y}{h}\right) u(x_1, x_2) dx_1 dx_2 \qquad (2.2)$$

where N is the total number of triangular subregions formed in the triangular element T_j as the result of stencil/mesh intersection, and τ_n is the n^{th} triangular subregion of the intersection. In the case that the stencil intersects a boundary of the domain, the stencil either wraps around the domain for periodic solutions, or an asymmetric (one-sided) stencil is used [72]. For further details on the discontinuous Galerkin method and postprocessing, see [50, 51, 52, 23, 73].

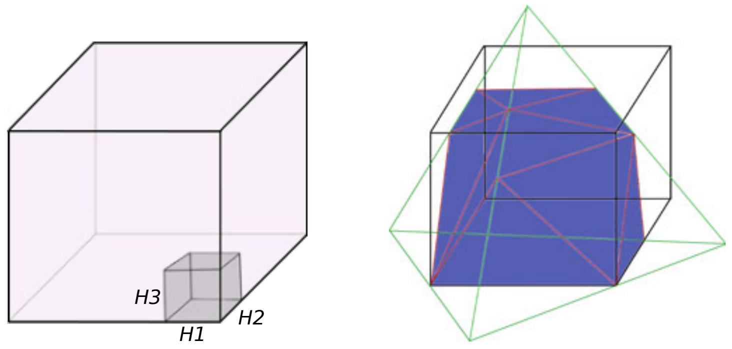
The postprocessor in 3D has the following form:

$$u(x, y, z) = \tfrac{1}{H_1 H_2 H_3} \times \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} K(\tfrac{x_1 - x}{H_1}) K(\tfrac{x_2 - y}{H_2}) K(\tfrac{x_3 - z}{H_3}) u_h(x_1, x_2, x_3) dx_1 dx_2 dx_3 \qquad (2.3)$$

where u_h is the approximate dG solution of the numerical simulation and H_i, where $i = 1, 2, 3$ are the kernel scaling parameters in each direction. The convolution kernel as presented in Equation (2.3) and the dG approximation u_h are piecewise polynomials. Therefore, to numerically evaluate the integral exactly to machine precision, the integration domain must be subdivided into regions of sufficient continuity, where the integrand does not have any break in regularity. In three dimensions, the footprint of the kernel is contained in a cube that is further subdivided by the kernel knots into smaller cubes of $H_1 \times H_2 \times H_3$ dimensions. A polyhedron clipping algorithm is applied to find the geometric intersection between a tetrahedral mesh element and a cube element of the B-spline, as demonstrated in [53].

To calculate the integral involved in the postprocessed solution in Equation (2.3) exactly, similar to the 2D case, the tetrahedral elements covered by the stencil support are decomposed into subelements that respect the stencil nodes. The resulting integral is calculated as the summation of the integrals over each subelement. Figure 2.6 depicts an example decomposition of a tetrahedron element and a hexahedron based on the stencil-mesh intersection.

To evaluate the postprocessed solution at a point denoted by (x, y, z), the numerical kernel is centered at that point, and the intersecting regions are subdivided and triangulated. After discretizing Equation (2.3) becomes



(a) B-spline stencil support

(b) Intersection between stencil hexahedron and tetrahedron

Figure 2.6: An illustration of an intersection between a hexahedral numerical B-spline kernel and a tetrahedral element.

$$u^{\star}(x,y,z) = \frac{1}{H_1 H_2 H_3} \sum_{T_j \in Supp\{\hat{K}\} T_j}$$

$$\int \bar{K}(x_1)\bar{K}(x_2)\bar{K}(x_3)u_h(x_1,x_2,x_3)dx_1dx_2dx_3 \qquad (2.4)$$

where $\bar{K}(x_i) = K\frac{(x_i-k)}{H_i}, k = x,y,z$ is denoted for simplicity, and $Supp\{K\}$ contains all the tetrahedral elements T_j that intersect with the numerical kernel footprint. This integration is performed for each triangulated subregion and summed together.

CHAPTER 3

DATA REUSE THROUGH ASSOCIATIVE REORDERING

3.1 Background

The SIMD architecture of the GPU fits well with stencil computations due to the inherent data-level parallelism. High performance of stencil computations on GPUs has been demonstrated using techniques such as autotuning and automatic generation of code [100]. Techniques such as data layout transformation and dynamic tiling at the thread level have been demonstrated in [19]. Various frameworks have been developed for increased data reuse with stencil operations [81, 9].

3.2 Algorithm

In previous work, stencil computations have often been defined as a method that updates each point in a structured grid according to an expression that depends upon the values of neighboring points in a fixed geometric pattern. For the case of discontinuous Galerkin (dG) postprocessing, we use a more general definition of stencil computations, which is the localized sampling area centered around a grid point that intersects with the mesh geometry. We now define the *key concepts* used in the context of stencil computations over unstructured meshes: computation grids, stencil operations, spatial data structures, and buffered vs. in-place stencils. All of our tests were conducted over 2D unstructured triangular meshes, and therefore we use the terms element and triangle interchangeably.

When evaluating stencil operations over a mesh, a set of evaluation points must be derived in relation to the underlying geometry. This set of points over which the stencil computations are evaluated is denoted as the computation grid. In the case of postprocessing of dG solutions, the evaluation points are the quadrature points of the polynomial interpolant defined over each element. Figure 3.1 illustrates an example of structured and unstructured 2D triangular meshes along with the set of grid points derived from them.

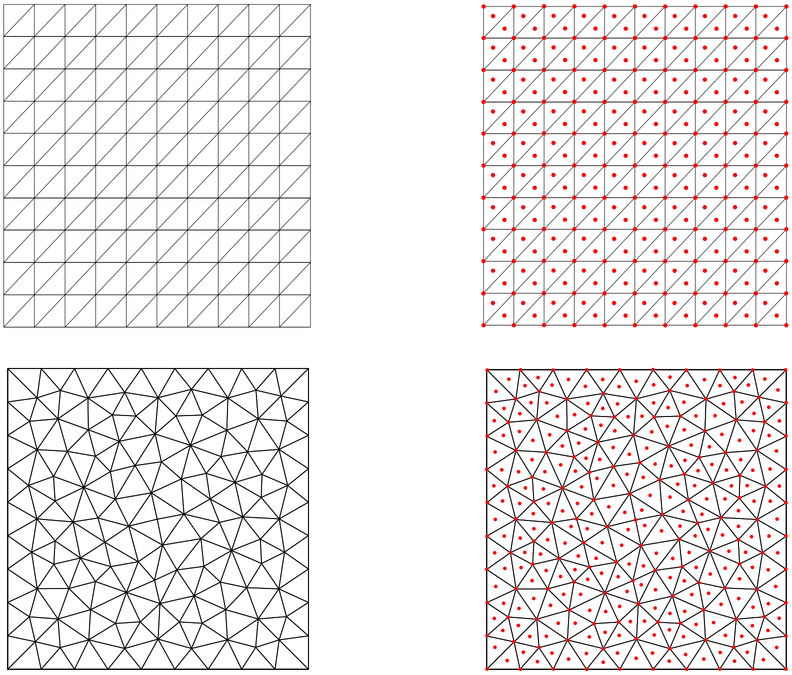


Figure 3.1: Structured and unstructured meshes and their respective structured and unstructured grids.

In the case of structured meshes, the layout of the quadrature points will follow a regular pattern. For unstructured meshes, the layout of the quadrature points will depend on the size, shape, and orientation of the elements. Postprocessing of dG solutions requires sampling the discontinuous piecewise functions that exist over the elements of the mesh.

We define stencil operations to be computations performed that update the value of a grid point at which the stencil is centered, using information within the localized sampling region. The computations depend upon function values of sampled points that lie within the stencil. The stencil may differ for each grid point when computing stencil operations over unstructured grids because the set of sample points within the stencil depends upon the intersection between the stencil and the underlying geometry. The varying intersection spaces between grid points will lead to a nonregular sampling pattern that must be calculated independently for each grid point.

As stencil operations rely on local neighborhood relationships between evaluation points, it is a common operation to query all elements within some distance of a given point. Therefore, an efficient method for accessing elements or points within some spatial region is required. There exist a number of data structures used for spatially decomposing an unstructured grid or mesh in an efficient manner, such as k-d trees, uniform hash grids, quad/oct trees, and bounding volume hierarchies [75]. Given that the stencils, in this case, are square and grid points are roughly uniformly distributed, a uniform hash grid was the most applicable choice [15].

We differentiate between stencil types based on how they operate over their solution memory space. In-place stencils sample from the same memory locations where the solutions are written, which is often the case with time-dependent iterative stencil computations. In-place stencils must be tiled in some fashion to avoid race conditions. Buffered stencils write the solution to a separate memory space from the space that is sampled to compute the stencil. As such, buffered stencil operations can be processed independently of each other without concern for race conditions. Postprocessing of dG solutions is a buffered stencil operation.

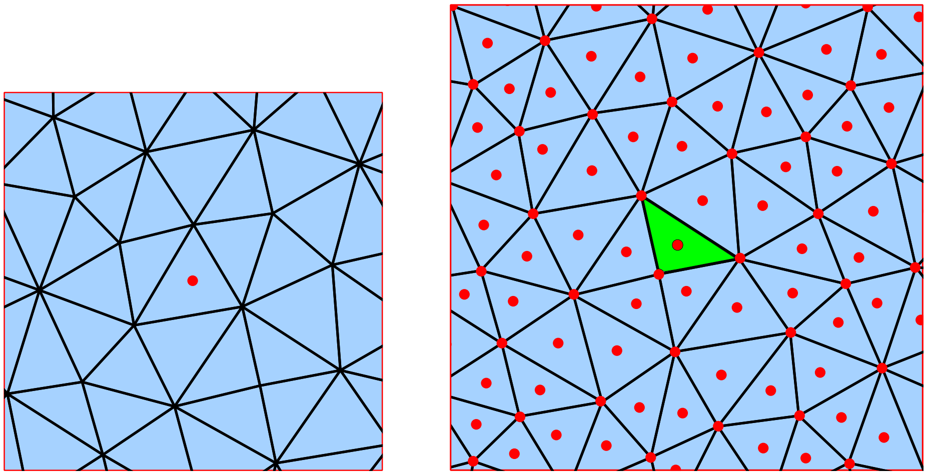
Algorithm 1: SutherlandHodgman (SH) Algorithm

```
input  : clipPolygon, subjectPolygon
output: intersectionPolygon
List outputList = subjectPolygon;
for Edge clipEdge in clipPolygon do
    List inputList = outputList;
    outputList.clear();
    Point S = inputList.last;
    for Point E in inputList do
        if E inside clipEdge then
            if S not inside clipEdge then
                outputList.add(Intersection(S,E,clipEdge));
            outputList.add(E);
        else if S inside clipEdge then
            outputList.add(Intersection(S,E,clipEdge));
        S ← E;
intersectionPolygon ← outputList;
```

3.2.1 Stencil Evaluation

The most straightforward method for postprocessing is a per-point evaluation method that iterates over the grid of points, and for each point finds all elements that intersect with the stencil centered around that point. Those intersected regions are then integrated and the values summed to produce the value of postprocessed solution at that grid point. We propose an alternate method that is a per-element evaluation method that iterates over each element, and for every element finds all of the points whose stencil intersects with that element. Each individual intersection is then integrated, which produces a number of partial solutions that are scattered to multiple grid points. Figure 3.2 illustrates these two methods. In per-point evaluation, integrations are all partial sums of the same grid point. In per-element evaluation, every grid point whose stencil intersects with the given element will have its value updated with a partial solution.

Postprocessing of dG solutions over unstructured meshes requires finding the intersections between the B-spline stencil and the mesh geometry. We use the Sutherland-Hodgman algorithm [84] to find and triangulate these intersections. This clipping algorithm finds the polygon that is the intersection between two given arbitrary convex polygons and divides the intersection into triangular subregions. Figure 3.3 illustrates this triangulation process. The convolution stencil used in the postprocessing algorithm is broken down into an array of squares as depicted by the red dashed lines. Consequently, the problem of finding the integration regions becomes the problem of finding the intersection areas between each square of the stencil array and the triangular elements covered by the stencil support.



(a) Per-Point

(b) Per-Element

Figure 3.2: Per-point versus per-element evaluation. Red points indicate grid points that will updated by this evaluation. The bounds indicate the area covered by the stencil. In the per-point case, the red dot indicates the point whose solution is being evaluated. In the per-element case, the partial solutions are evaluated with respect to the green highlighted element.

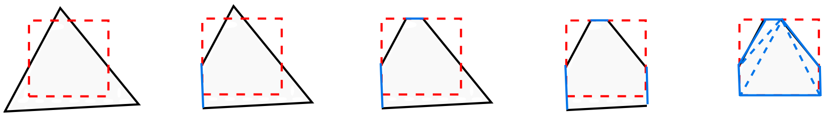


Figure 3.3: A sample triangulation of an intersection region by the Sutherland-Hodgman algorithm.

Figure 3.4 depicts a sample stencil/mesh overlap.

DG postprocessing consists of two main steps. The first is finding and triangulating the stencil/mesh intersections, which will create a set of triangulated subregions. The second is integrating those subregions according to Equation (2.2) and summing the results. The resulting sum is the postprocessed value of the solution u^* at that point.

3.2.2 Grid Construction

A spatial data structure is needed to efficiently search the elements of an unstructured mesh in order to determine in which element a given point lies. We perform a uniform subdivision of the mesh and each element/point is stored in a hash grid cell based on its spatial coordinates. For per-point sampling, the hash grid stores the centroid location of each element. On unstructured meshes, the centroid of a triangle may be located in a cell while sections extend into neighboring cells. To ensure enclosure (i.e., no triangle spans

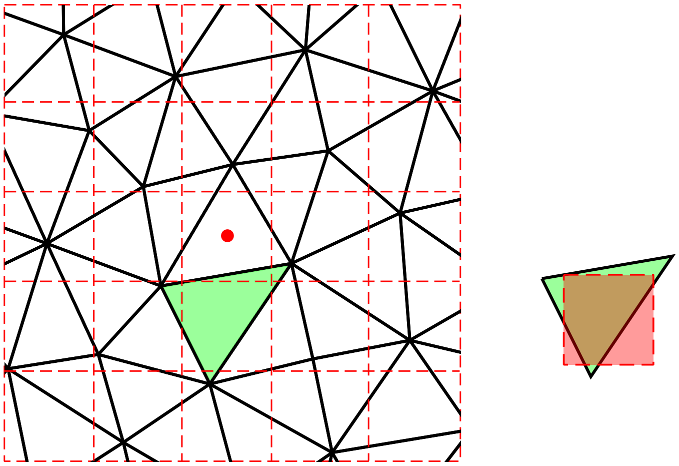


Figure 3.4: A sample stencil/mesh overlap. Dashed lines represent the 2D stencil as an array of squares. The intersections of the dashed lines are stencil node locations. The subfigure on the right illustrates the intersection of the green highlighted element and the overlapping stencil square.

more than two cells in any one dimension), a minimum size on the cells of the hash grid is imposed. The minimum size used in our computation to guarantee enclosure is the length of the longest edge among all triangles in the mesh. In the per-element case, the hash grid stores the grid points instead of the triangular elements of the underlying mesh. The decomposition in this case has no minimum size restriction on the cells of the grid.

When evaluating the intersection of a stencil and the triangular mesh, we first evaluate the intersection of the stencil and the uniform hash grid. The intersected cells store indices of the elements/points that must be tested for intersections with the given element/point being evaluated. We choose the domain of the hash grid to be $[0,1]$ in both dimensions, with per-point and per-element cell spacings c_p and c_e, respectively. We set c_p and c_e to be some factor of s, which represents the longest side amongst all triangles in the mesh. For our tests, we used $c_p = s$ and $c_e = \frac{s}{2}$. Construction of the uniform hash grid follows from dividing the mesh into $\lceil \frac{1}{c_p} \rceil$ and $\lceil \frac{1}{c_e} \rceil$ cells in each dimension.

Given an element with vertices (A, B, C) and a grid point (x, y), we construct a bounding box around the element with corners being defined as

$$\begin{aligned} min_x &= \min(A_x, B_x, C_x) & min_y &= \min(A_y, B_y, C_y) \\ max_x &= \max(A_x, B_x, C_x) & max_y &= \max(A_y, B_y, C_y). \end{aligned}$$

The bounds are extended by half of the stencil width, which is defined to be $w = s(3P+1)$, where P is the polynomial order. The bounds of the per-element and per-point stencils (e, p) are defined as

$$\begin{aligned} left_e &= \lfloor \frac{min_x - \frac{w}{2}}{c_e} \rfloor & left_p &= \lfloor \frac{x - \frac{w}{2}}{c_p} \rfloor - 1 \\ right_e &= \lfloor \frac{max_x + \frac{w}{2}}{c_e} \rfloor & right_p &= \lfloor \frac{x + \frac{w}{2}}{c_p} \rfloor + 1 \\ top_e &= \lfloor \frac{max_y + \frac{w}{2}}{c_e} \rfloor & top_p &= \lfloor \frac{y + \frac{w}{2}}{c_p} \rfloor + 1. \\ bottom_e &= \lfloor \frac{min_y - \frac{w}{2}}{c_e} \rfloor & bottom_p &= \lfloor \frac{y - \frac{w}{2}}{c_p} \rfloor - 1 \end{aligned} \tag{3.1}$$

The hash grid is constructed in a similar manner for both methods, with the per-point hash grid storing the triangle elements and the per-element hash grid storing the grid points.

The size of the intersection search space, in each dimension, for the per-point method is the sum of the stencil width and the width of the cells surrounding the stencil, known as the *halo* region [37]. The size of the intersection space for the per-element scheme is the sum of the width of element bounding box and the stencil width. The resulting size of the intersection search space has an upper bound of $2s + w$ for the per-point scheme, and $s + w$ for the per-element scheme. Figure 3.5 illustrates the difference in the intersection search spaces between the two methods. Elements that lie within the *halo* cells around the stencil but do not intersect the stencil are also tested, which results in additional unnecessary stencil/triangle intersection tests in the per-point case. Data about the number of intersection tests performed with the per-point and per-element hash grids are detailed in Table 3.1.

A single point cannot span more than one cell, which allows for smaller cells that form a tighter bound around the stencil, and additionally, the elimination of the *halo* region. We found that setting the cell size equal to half the maximum triangle edge size produced good results. This method makes a tradeoff by reducing uncoalesced reads from sampling the unstructured mesh and increasing coalesced writes by splitting the solution in parts. Note that not every triangle tested will intersect with the stencil around the grid point. Only true positive intersections will be integrated.

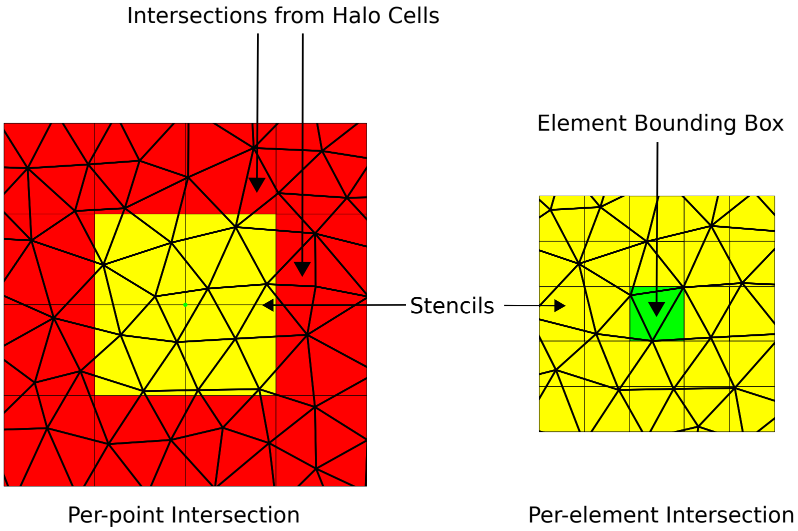


Figure 3.5: Per-point versus per-element mesh intersections on hash grid. The yellow areas denote the stencil regions, the red area denotes the halo region, and the green area is the element bounding box.

Table 3.1: Number of intersection tests performed with the per-point and per-element methods using linear polynomials.

Mesh Size	# of Per-Point Intersection Tests	# of Per-Element Intersection Tests
4k	6647394	3525297
16k	26492809	14235618
64k	110778427	59277119
256k	455614318	243245703
1024k	1919070326	1017924543

Algorithm 2: Per-Point Post Processing

```
foreach Point $p$ do
    // Compute hash grid bounds
    L,R,T,B $\leftarrow$ PointHashGridBounds(p);
    foreach Cell $j$ within bounds $L,R,T,B$ do
        foreach Element $e$ in Cell $j$ do
            // Compute and store per-element data
            ED $\leftarrow$ ElementData();
            // Compute and triangulate stencil/element intersections
            Regions $\leftarrow$ SH(Stencil(p), e);
            // Integrate triangulated regions
            Solution[p] $\leftarrow$ Solution[p] + Integrate(Regions, ED);
```

3.2.3 Per-Point Evaluation

To evaluate a stencil computation with the per-point method, a stencil is centered around each grid point and the intersections between that stencil and the underlying mesh geometry are found. When determining the mesh/stencil intersection, we first determine the intersection between the hash grid and the stencil. A bounded region on the hash grid is determined by centering the stencil at the grid point and expanding the borders to the nearest cell boundary in each dimension, as denoted in Equation (3.1). Next, each element within the bounded cells is tested for intersections. Intersected regions are then triangulated with the Sutherland-Hodgman algorithm and integrated. The set of *halo* cells around the bounded region must be included to ensure that all intersecting triangles are tested. Algorithm 2 provides psuedocode for the per-point evaluation method. The element data requires a minimum of $\frac{(P+1)(P+2)}{2} + 3$ values to be read from memory per integration, where P is the polynomial order.

Algorithm 3: Per-Element Post Processing

```
foreach Element $e$ do
    // Compute hash grid bounds
    L,R,T,B ← ElementHashGridBounds(e);
    // Compute and store element data in Shared Memory
    ED ← ElementData();
    foreach Cell $j$ within bounds $L,R,T,B$ do
        foreach Point $p$ in Cell $j$ do
            // Compute and triangulate stencil/element intersections
            Regions ← SH(Stencil(p), e);
            // Integrate triangulated regions
            PSolution[patch(e), p] ← PSolution[patch(e), p] + Integrate(Regions, ED);
// Perform reduction on solution by patch
Solution ← Reduction(PSolution)
```

3.2.4 Per-Element Evaluation

The per-element evaluation scheme groups sample points by the underlying geometric element in which they happen to fall. The per-element stencil bounds, denoted in Equation (3.1), enclose an area that includes all grid points that have stencil intersections with the bounding box of the triangle. From this bounded area, the set of grid points whose stencils intersect the triangle is determined. Each grid point that falls within this region is tested for a stencil/triangle intersection using the given triangle element. The evaluation points within the triangle are then processed concurrently. The per-element scheme breaks up Equation (2.2) into partial solutions. The partial solutions are grouped together by triangular element, and each element will contribute partial solutions to every grid point whose stencil intersects that triangle. We divide the mesh into patches, the details of which are described in the next section, with the solution of each patch being accumulated into a separate memory space. Algorithm 3 provides the psuedocode for the per-element evaluation method.

Data associated with the given element, such as the elemental coefficients and the bounds of the triangle, can be stored and reused for all evaluations, which takes advantage of data locality and leads to more coalesced memory accesses than in the per-point scheme. In the per-element case, only the spatial offset of the grid point (two values in 2D) is required to be read per integration, since the ($\frac{(P+1)(P+2)}{2} + 3$) values associated with the triangle are reused for all integrations over that element.

3.3 Implementation

The Sutherland-Hodgman algorithm presents a challenge in efficient postprocessing on many-core architectures. The highly divergent nature of the intersection processing, caused by branching logic, may lead to suboptimal performance on streaming SIMD architectures. The polygon clipping that takes place within the Sutherland-Hodgman algorithm occurs at irregularly-spaced intervals on an unstructured mesh. The GPU architecture relies on SIMD parallelism to gain efficiency, and this irregularity causes divergence between threads that are operating synchronously, which leads to noticeably poorer performance for unstructured meshes vs. that of structured meshes, due to noncontiguous memory accesses and thread divergence. Minimizing the total number of intersection tests is key to achieving high performance with stencil computations over unstructured meshes on SIMD architectures.

In the per-point method, we assign a block to compute the solution for a given grid point. On the GPU, we use a number of blocks equal to the SM count on the GPU (N_{SM}). The blocks then iterate over the points in a strided fashion (i.e., P_{i+k*N_B}, where P_i is the i^{th} point, N_B is the number of concurrent blocks, and k is an incrementing integer). Within a block, we assign threads to iterate over the element indices that lie within intersected cells of the hash grid in a similar strided fashion. The stencil/element intersections are then tested and integrated. There is no contention between stencils, as each stencil updates a discrete grid point. In this case, it is trivial to achieve perfect load balancing between all processing groups. In the per-element method, we assign a block to each patch. The threads within the blocks iterate over the points stored within the intersected cells of the hash grid in a strided manner. To maximize parallelism, we choose a number of blocks equal to the number of SMs per card. For multi-GPU decomposition, we divide the mesh into $N_{GPU} \times N_{SM}$ patches, where N_{GPU} is the number of GPUs. In the multi-GPU implementation, we use a two stage reduction. In the first stage, each GPU computes a reduction on the patches that it processed, which is followed by a final reduction of the resulting solutions from the second stage.

The per-element evaluation scheme requires that concurrent execution of stencil tiles acting on the same memory space do not overlap. Overlapping stencils may introduce race conditions where the value of a grid point is being updated by multiple stencils. To solve this problem, we assign a separate scratch pad memory space to each concurrent stencil tile where the partial solutions are accumulated. After all the stencils have finished their computations, all the partial solutions are summed to form the the final solution, which requires additional memory space, but allows for maximum parallelism without the need for pipe-lining of the stencils.

We implemented a spatially overlapped tiling scheme, introduced in [45], where each tile uses a disjoint memory working set. Each logical block is assigned to process stencils in a localized patch of the mesh. The partial solutions of each patch are stored in a separate scratch pad memory space. This requires that grid points lying along the borders of patches have multiple partial solutions. Grid points that fall within the intersection of stencils from multiple patches will have a partial solution stored in each memory set of these patches. Grid points that lie in the interior of a patch and only fall within stencils from that patch will have a single solution in memory. Figure 3.6 illustrates an example patch division and the partial solutions formed from the patches. The overlapped regions that lie within the intersection of stencils from multiple patches are summed together to produce the final result for those respective grid points, which leads to a relatively low amount of storage overhead. The memory overhead, relative to the memory requirement for the total solution, decreases as the mesh size increases.

Patch construction follows from simple recursive bisection of the mesh elements until there are k patches of roughly equal size, with k being the number of concurrently executing blocks. This method easily scales with the mesh size. As the domain size increases, the number of concurrent stencils can be increased. Patch perimeter distance should be minimized in order to minimize the overall memory overhead. Increasing the number of

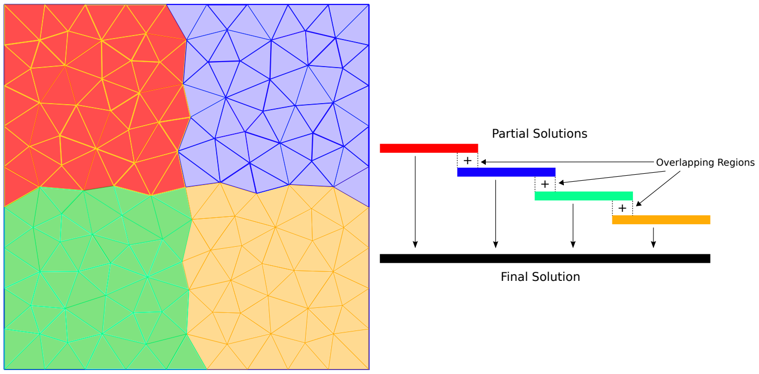


Figure 3.6: Example of mesh division into four patches.

tiles while decreasing the tile size has the effect of increasing overall memory overhead, but allows for higher parallelism. The number of concurrent executing tiles has a maximum upper bound equal to the number of geometric elements in the mesh. As the surface area of a patch grows at a faster rate than the perimeter, the memory overhead tends to be relatively low for large meshes, which also naturally extends to 3D with the memory overhead determined by the surface area to volume ratios of the patches.

The baseline memory consumption is the minimum amount of memory required to store the solution at all the evaluation grid points. The patch-based tiling method adds additional memory overhead based on the number of grid points that fall within the intersection of stencils from multiple patches. Each patch stores partial solutions for every grid point that falls within the union of intersections spaces of the elements contained within the patch. Only points near the boundaries of patches will require storing multiple partial solutions. The ratio of boundary length to patch area decreases inversely proportional to mesh size for a fixed number of patches. Figure 3.7 illustrates the scaling of memory overhead across the range of test meshes. The perimeter of a patch grows linearly and the surface area grows quadratically. The results demonstrate that relatively little overhead memory consumption is required for larger meshes.

The final summation of the partial solutions requires only a linear reduction based on the memory offset of each patch solution. In the reduction phase, we divide the grid points based on the patch in which they fall. We then assign a block to each patch, which performs the reduction on the partial solutions for those grid points. This process eliminates write contention to the final solution space. The process contributes a minimal amount of time to the overall process. We also explored a pipe-lined tiling method, but this introduces additional synchronizations between pipeline stages. There is no additional memory overhead introduced by pipe-lining, but there is reduction in overall performance.

3.4 Experimental Results

In this section we evaluate the performance of GPU implementations of the per-point and per-element methods. In addition, we demonstrate the scalability of our approach on 1, 2, 4, and 8 GPUs. We ran our tests on a node with two Intel Xeon E5630 processors (4 cores each) running at 2.53GHz, 128GB of memory, and eight NVIDIA Telsa M2090

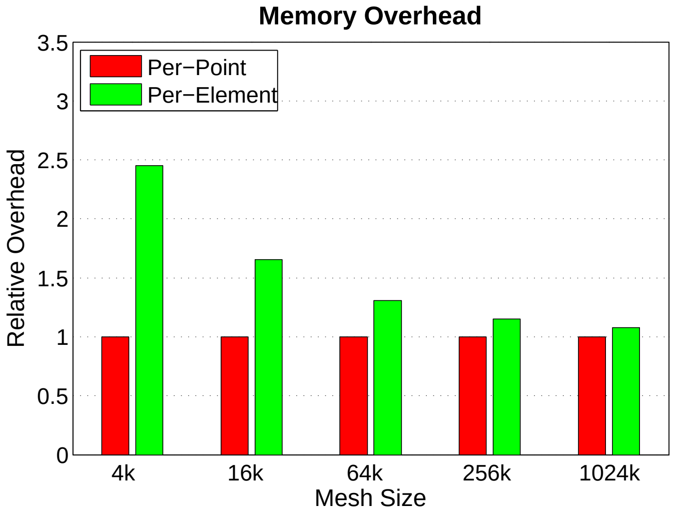


Figure 3.7: Memory overhead of per-element method using 16 patches with linear polynomials.

GPUs using CUDA 5.0. We executed the tests across a series of 2D unstructured triangular meshes created using Delaunay triangulation. We tested our implementations on two types of meshes. The first was an unstructured mesh with roughly uniform sized triangles, shown in Figure 3.8. The second type was an unstructured mesh with highly varying element sizes, shown in Figure 3.9. We tested each of these mesh types across mesh sizes on the order of 4k, 16k, 64k, 256k, and 1024k triangles. We used periodic boundary conditions with linear, quadratic, and cubic polynomials, which have 3, 6, and 10 coefficients, respectively, for triangular elements. All tests were conducted with double precision floating point values.

The postprocessing is divided into two main components, the first finds the intersections between the stencils and the underlying mesh geometry, and the second integrates those subregions and accumulates the results. The intersection finding has linear complexity with respect to the number of intersection tests performed, and the integral calculation has a computational complexity on the order of $O((P+1)^d)$, where P is the polynomial order used in the postprocessing of the finite element solution and d is the dimension. The higher computational complexity of integration calculation dominates the overall runtime as the polynomial order increases, which is demonstrated by the smaller performance increase between the the per-point and per-element evaluation scheme for quadratic and cubic polynomials.

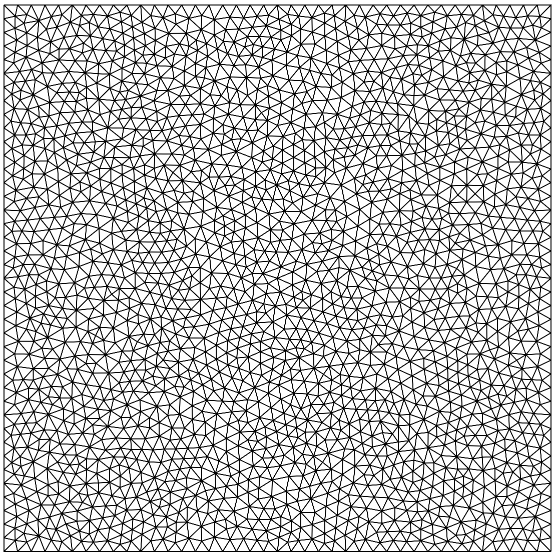


Figure 3.8: Unstructured mesh with low variance.

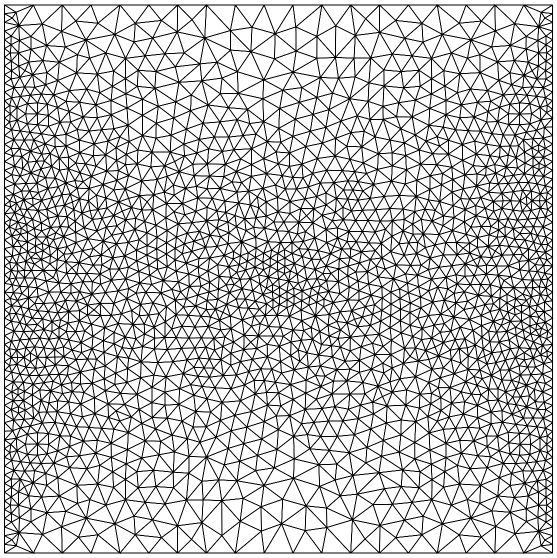


Figure 3.9: Unstructured mesh with high variance.

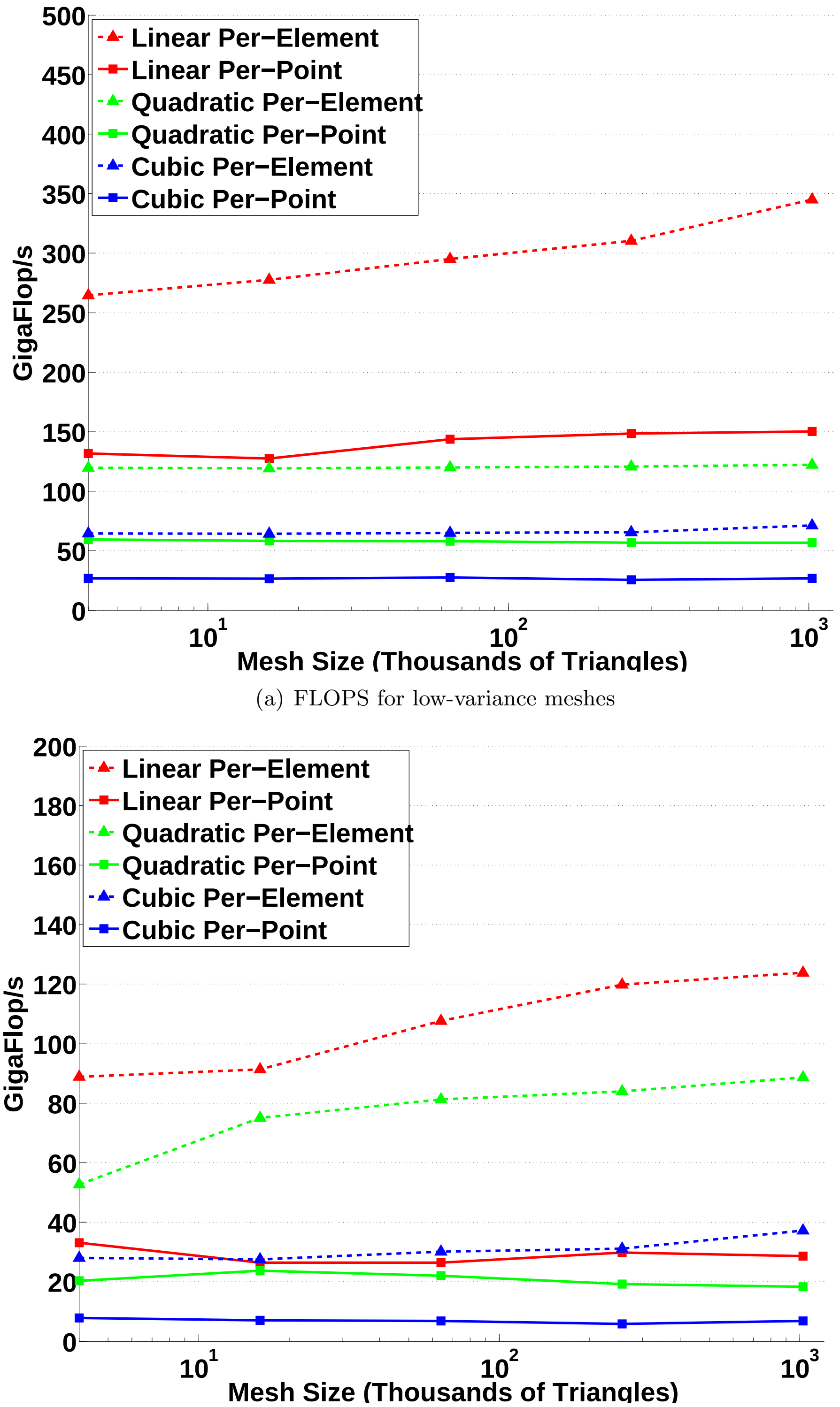
3.4.1 Metrics

Figure 3.10 (a) provides FLOP metrics for the GPU over low-variance meshes. The per-element method achieves a peak FLOP rating of 345 GFLOP/s for linear polynomials on the 1024k mesh. For quadratic and cubic polynomials, the FLOP ratings are lower, but the relative difference between the methods is larger. For quadratic polynomials, the methods achieve a peak FLOP rating between 50 - 120 GFLOP/s, and for cubic polynomials a peak rating of 30 - 60 GFLOP/s is seen. The computational complexity of the integral kernel grows quadratically with respect to the polynomial order. As polynomial order grows, the integral kernel occupies a larger percent of the total run-time, and the ratio of time spent computing intersections to time spent performing integrations decreases. In addition, the integration kernel requires storage of a large number of intermediate values that grow on the order of $O((P+1)^2)$. These constraints lead to a lower FLOP performance at higher polynomial orders.

Figure 3.10 (b) provides GPU flop ratings for high-variance meshes. The difference in FLOP performance between the two methods is more noticeable on meshes with high variance in element size, in part because the search area for the per-point method includes a halo region that has a cell width equal to the largest element size. This difference in search area has significantly more impact on performance than in the case of meshes with low variance in element size.

The results in Figure 3.11 illustrate the relative performance of the per-point and per-element method for low- and high-variance meshes. The timings of the per-point methods have been normalized. The performance difference between the per-element and per-point methods is greater on meshes with high variance in element sizes. The per-element method achieves over a 2× speedup for the low-variance mesh with cubic polynomials, and over a 3× speedup for the high-variance mesh.

The results demonstrate a significant performance improvement of the per-element evaluation scheme over the per-point scheme for many-core architectures. Local data associated with each element is accessed only once and reused for all evaluations within the element. The heterogeneity of the unstructured mesh leads to irregular memory access patterns and uncoalesced memory accesses. Fewer intersection tests combined with increased data reuse contribute to increased performance. The results provide insight into the performance of each evaluation method on many-core architectures. The streaming many-core architecture of the GPU benefits greatly from reduced intersection tests and increased data reuse of the local element information, in part due to the relatively low amount of cache per core. We also implemented a single-threaded CPU version of the methods. We noticed that implementations with low levels of concurrency see less benefit from data reuse. The improvement of per-element evaluation over per-point evaluation is less significant, and in a few cases even worse due to the increased overhead.



(a) FLOPS for low-variance meshes

(b) FLOPS for high-variance meshes

Figure 3.10: GPU Flop/s.

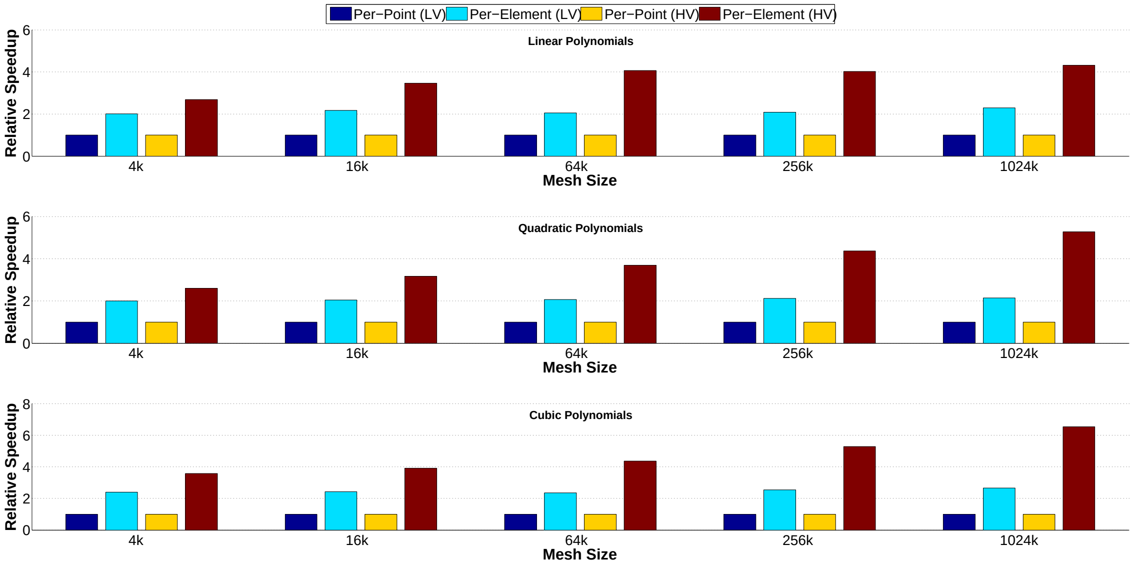


Figure 3.11: Relative speedup compared to a normalized per-point methods for low-variance (LV) and high-variance (HV) meshes.

3.4.2 Scaling

To demonstrate the scaling of the per-element method, we tested the per-element evaluation method on 1, 2, 4, and 8 GPUs across the entire range of our test meshes. The results demonstrate that the method has perfect linear scaling with respect to increased mesh size. This scaling is to be expected for a problem with outer parallelism where there are no inherent dependencies between grid points. Figure 3.12 illustrates the scaling of the GPU per-element method across the range of test meshes for linear polynomials. Parallelization across GPUs was achieved by subdividing the mesh into the $N_{GPU} \times N_{SM}$ patches and evenly distributing them between the GPUs.

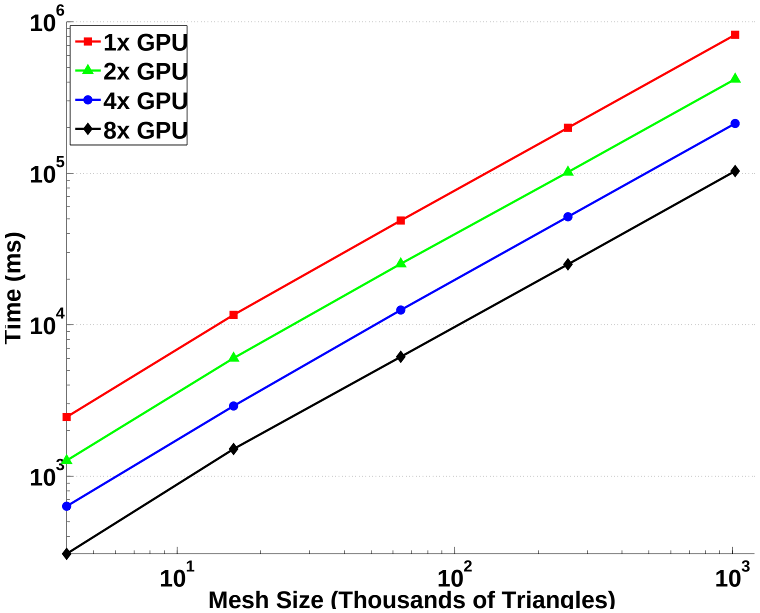


Figure 3.12: Scaling of the per-element method on 1, 2, 4, and 8 GPUs with linear polynomials.

CHAPTER 4

TUNING VECTORIZATION WIDTHS

4.1 Vectorization

Vectorization is the process by which a scalar operation that operates over a pair of operands is converted to operate over a vector of pairs of operands (a series of adjacent values). Vectorized instructions operate over multiple pairs of data in parallel. Performance is improved through this added level of parallelism and increased memory efficiency. Vectorization is one of the primary design aspects used in SIMD architectures that has allowed modern GPUs to achieve beyond teraflop level performance.

Vector instruction widths typically range from 2 to 64 (2, 4, 8, 16, 32, or 64 adjacent data elements). Vectorization of an algorithm can lead to significant performance improvements in some cases. Streaming SIMD extensions (SSE) are an example of a vector instruction set designed for the x86 architecture that has seen wide adoption. Modern GPUs group cores into work units (32 or 64 cores typically) that operate in lock step like a vector processor.

Although the vector width of the hardware is generally fixed, the logical vector width used by the programmer within a computation can range from 1 up to the width of the computation, which is done by decreasing the vector width and increasing the number of concurrent vectors. For example, with a hardware vector width of 8, one could compute 1 operation of width 8, 2 operations of width 4, 4 operations of width 2, or 8 operations of width 1. Adjusting this logical vector width in conjunction with the number of concurrent vectors can have a significant impact on overall performance and throughput.

Achieving high performance with complex code bases that involve numerous architecture specific parameters remains a difficult task. Often, tedious manual optimizations are required. This process is time consuming for the programmer, and manual tuning rarely yields the optimal parameter configuration. Autotuning is a method by which optimal or near optimal run-time configuration parameters are selected through an automated testing process [65]. This technique has proven to be a valuable tool for improving performance and increasing programmer efficiency.

4.2 Autotuning

The goal of autotuning is often to optimize execution time, memory usage, or energy consumption. It has been demonstrated that this goal can be achieved by exploiting domain-specific knowledge [93, 57], or through tuning application-independent parameters [86]. Autotuning has proven to be a valuable technique for exploring the configuration search space within applications on multi-core and many-core architectures. It can be used to optimize parallel applications in an automated fashion, which lifts the burden from the programmer of having to tediously test a large number of run-time configurations. On GPUs, it has been noted that a number of application-independent parameters, such as block size, number of blocks, and loop unrolling factors, can be tuned [88].

Empirical autotuning seeks to find the optimal runtime configuration by searching through a combinatorial set of parameter configurations [98]. This automated process recompiles/retests the code with different parameter configurations and compares performance against a set of benchmarks. Some autotuning frameworks adjust runtime parameters which allows the code to be retested without the need for recompilation [70]. However, this process often requires an expensive combinatorial search process to exhaustively examine the entire parameter domain, which is often prohibitively slow. Typically a domain informed search space representation is required for good performance because no single set of application independent parameters works well for all problems. A number of frameworks have been developed which use heuristics to prune the search space and tune the program based on a set of configurable parameters [3, 35].

Model-driven autotuning has been used in various applications to decrease the time for exploring a large search space. Model driven approaches attempt to predict the performance behavior of configurations. Often the model is fitted with a set of reference points that are generated randomly via Monte Carlo sampling. One distinct advantage of the model-driven approach is that good code can be produced from an accurate analytical model.

Often model-driven and empirical autotuning are combined. First, the search space is reduced to a few promising areas based on the analytical model, and then those areas are searched empirically to find the optimal configurations. These methods are often nonlinear, and there is no guarantee that the autotuning will find the global minimum as it may get stuck within a local minimum.

Predictive autotuning goes even further than the model-driven approach and attempts to train the analytical model through statistical and machine learning techniques. These approaches have seen some success with nonlinear regression modeling techniques such as regression trees [16]. Various other machine learning and statistical prediction techniques have also been successfully applied [56, 80, 29, 38, 35].

4.3 Application

This work analyzes the impact of tuning vectorization widths with the application of 3D dG postprocessing over tetrahedral meshes. 3D dG postprocessing involves computing a series of dense integral computations which are reduced to produce a final value for each postprocessed point.

4.3.1 DG Postprocessing

From a computational perspective, postprocessing over tetrahedral meshes is a challenging task. Computing the postprocessed value at a single point (x, y, z) requires evaluating the formula given in Equation (4.1).

$$u^{\star}(x,y,z) = \frac{1}{H_1H_2H_3} \sum_{T_j \in Supp\{\hat{K}\} T_j} \int \bar{K}(x_1)\bar{K}(x_2)\bar{K}(x_3)u_h(x_1,x_2,x_3)dx_1dx_2dx_3 \tag{4.1}$$

Evaluation can be divided into two distinct steps:

1. Identifying the support of the numerical B-spline kernel, centered at (x, y, z), over the dG mesh, followed by calculating the geometric intersection to obtain the integration regions.

2. Numerically evaluating the integrals by means of quadrature rules.

In the case of structured tetrahedral meshes, it suffices to find the extent of the numerical kernel on the basic hexahedral mesh that has a uniform structure. The footprint of the numerical kernel over such a uniform mesh can be found in constant computational time.

When dealing with unstructured meshes, the elements can be grouped within cells of a hash grid. To find the integration regions, we use a modified version of the Sutherland-Hodgman clipping algorithm. In this algorithm, we loop through the faces of one polyhedron and clip it against the second polyhedron. The computational complexity of this algorithm is $O(n)$, where $n = f_1 \times f_2$ and $f_i,\ i = 1, 2$ indicates the number of faces in each polyhedron. For structured tetrahedral meshes, the support of the numerical kernel spans $3k+1$ hexahedral elements in each direction, with k being the degree of the approximation. Each evaluation point requires processing $6 \times (3k+1)^3$ tetrahedra for the configuration we chose in our tests. As each tetrahedral element intersects with at most eight cubes of the numerical kernel, the cost of finding all the intersection regions will therefore be $8 \times 6 \times 24(3k + 1)^3$ or $O(k^3)$.

To evaluate a stencil computation around a given point, a stencil is centered at that point, and the intersections between that stencil and the underlying mesh geometry are found. There are two general strategies for this evaluation [42]: per-point and per-element evaluation. The main difference between these strategies is that the per-point method iterates over points while computing stencil intersections with a list of elements, whereas the per-element method iterates over elements while computing stencil intersections with a list of points. First, the intersection between the hash grid and the stencil is determined. A bounded region on the hash grid is determined by centering the stencil at the grid point and expanding the borders to the nearest cell boundary in each dimension. Next, each element/point within the bounded cells is tested for intersections. Intersected regions are then tetrahedralized and integrated.

The algorithm performs a series of tetrahedron/plane clipping tests, with the intersection of each test being divided into zero to three tetrahedrons, which are then stored in a list to be tested against the subsequent planes. These tests are performed for the six planes that coincide with the faces of the hexahedron. The result of this clipping algorithm will be a list of zero or more tetrahedrons that are subregions of the mesh/stencil intersection. The convolution stencil used in the postprocessing algorithm is broken down into a lattice of cubes (a 2D illustration of this is depicted in Figure 4.1). This intersection finding in 3D is much more computationally demanding than in the 2D case. In 2D, a triangle/quadrilateral intersection can form at most five triangles, whereas in 3D, a tetrahedron/hexahedron intersection can form up to 12 tetrahedrons. The 3D computation requires a significant

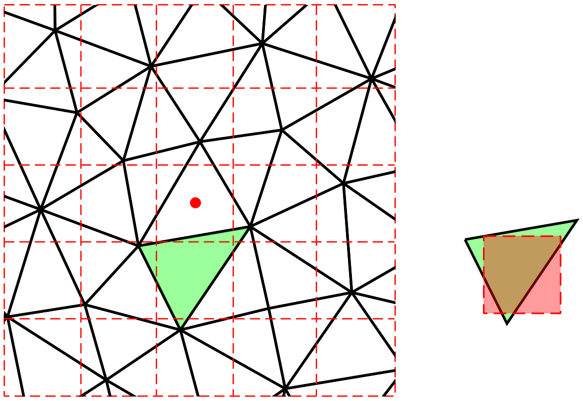


Figure 4.1: A sample stencil/mesh overlap depicted in 2D. Dashed lines represent the 2D stencil as an array of squares. The region on the right illustrates the intersection of the green highlighted element and the overlapping stencil square.

amount of additional branching logic and dynamically allocated memory, both of which lower SIMD efficiency.

DG postprocessing consists of two main steps. Step 1 finds and tetrahedralizes the stencil/mesh intersections. Step 2 integrates those tetrahedralized subregions according to Equation (4.1) and sums the results. The resulting sum is the postprocessed value of the solution u^* at that point. In our tests, we used the per-element method [42]. This method processes each mesh element individually and stores partial solutions for each point whose stencil intersects with the given element. A reduction performed at the end sums the values of the partial solutions.

4.3.2 Grid Construction

As in the 2D case, we use a uniform hash grid to organize the point/elements of the mesh for easy searching. When evaluating the intersection of a stencil and a mesh, first the intersection of the stencil and the uniform hash grid is evaluated. The intersected cells store the indices of the elements/points that must be tested for intersections with the given element/point being evaluated. We choose the domain of the hash grid to be $[0, 1]^3$, with per-point and per-element cell spacings c_p and c_e, respectively. In our tests, since we used a tetrahedral decomposition from a structured grid of hexahedrons, we set c_p and c_e to be the length of the hexahedral cells. Construction of the uniform hash grid then matches the exact hexahedral cell formation of the original mesh.

The hash grid is constructed in a similar manner for both methods, with the per-point hash grid storing the tetrahedron elements and the per-element hash grid storing the grid points. The size of the intersection search space, in each dimension, for the per-point method is the sum of the stencil width and the width of the cells surrounding the stencil, known as the *halo* region [37]. The size of the intersection space for the per-element scheme is the sum of the width of element bounding box and the stencil width. The resulting size of the intersection search space is larger for the per-point scheme than for the per-element scheme.

The geometric intersection is computed through a simple convex polygon clipping algorithm. Given a hexahedron and a tetrahedron, the algorithm tests each tetrahedron against each face of the hexahedron using a plane intersection test. The part of the tetrahedron that lies on the inside of a given plane, based on the face normal, is kept and tetrahedralized. This intersection computation will generate a new set of tetrahedrons that are put into a list and tested against the next face of the hexahedron. The final result will be a list of tetrahedrons that lie within the intersection of the original tetrahedron and the hexahedron. There are six possible intersection scenarios for a tetrahedron and a plane. The variability in the intersection tests, combined with the fact that the number of tetrahedral regions generated is data dependent and can range from 0 to 12, yields a highly branch-divergent algorithm. The geometric intersection computation is executed a minimum of six times and has a branch for each of the six intersection scenarios. Thread divergence on the GPU can have a significant impact on performance. In the worst case, it can cause all the threads in the warp to execute in a serialized fashion.

The integration computation is computed for each quadrature point within the tetrahedron. The number of quadrature points N_{qp} depends upon the degree of the polynomial over the element, and is defined as

$$Q_0 = \lceil \frac{4p+3}{2} \rceil \quad Q_1 = 2p+1 \quad Q_2 = 2p+1$$
$$N_{qp} = Q_0 Q_1 Q_2$$

where p is the polynomial order. Each subregion integration requires 16 values to be transferred to the GPU: 1 for the point index, 3 for the (x, y, z) coordinates of the point, and 12 for the coordinates of the four vertices of the tetrahedral subregion. A set of quadrature points is mapped to the tetrahedron in order to integrate over the subregion.

This integration requires evaluating the function at 36 quadrature points for $p = 1$ (linear polynomials), and at 150 quadrature points for $p = 2$ (quadratic polynomials).

At the core of the postprocessing algorithm is an integration operation, illustrated by Equation (4.1). This integration is computed via quadrature using Jacobi polynomials. A series of vector dot products must be computed to evaluate $u_h(x_1, x_2, x_3)$. The integration operation involves successively evaluating Jacobi polynomials at quadrature points. These values are stored temporarily, and then later a reduction is performed that accumulates into the resulting integration value. The vector width parameter influences data-level parallelism within the computation and the amount of shared memory required.

4.3.3 Loop Restructuring

Full vectorization requires larger memory footprints per operation, which increases the likelihood of having larger gaps of unused memory in shared memory. Partially vectorized operations allow for more operations with smaller memory footprints to be allocated to the shared memory space, reducing the likelihood of large memory gaps. A nonvectorized computation assigns each thread to a different operation and allows for perfect utilization of shared memory. Such a method requires more per-instance memory overhead and also decreases the opportunities for coalesced memory accesses, which is a significant source of performance on SIMD architectures. Mapping threads with partial vectorization compared to full vectorization is illustrated in Figure 4.2.

The vectorization can be varied by decreasing the vector width and increasing number of concurrent vectors or vice versa. A partially vectorized computation has some number of threads equal to the stride length (logical vector width) assigned to each operation. The

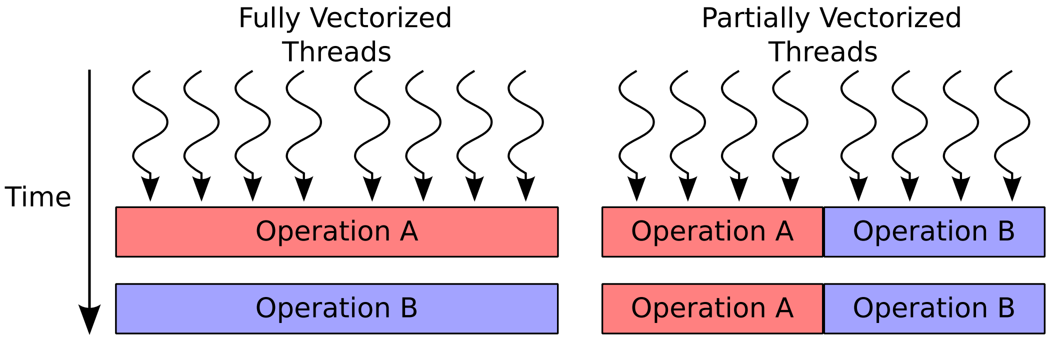


Figure 4.2: Example of varying thread vectorization width.

threads iterate over the vector by shifting their memory offsets by the stride length in each iteration. A fully vectorized computation corresponds with a stride that is equal to the vector width. In that case, a thread is assigned to each element in the vector for each iteration.

Listing 4.1 provides simple pseudocode of how a computation could be nonvectorized, partially vectorized, and fully vectorized. In this example, we use component-wise vector addition $A_i = B_i + C_i$, where the arrays A, B, and C are each an array of structures. Within a nonvectorized computation, each local operation has a single thread assigned to it. All the data associated with that operation are accessed and stored locally by that thread. Within a partially vectorized operation, the number of threads assigned is equal to the stride length; a value greater than 1 and less than the total width of the computation.

```
Vectors A[WIDTH], B[WIDTH], C[WIDTH];
int vID; //vector ID
int vLane = threadID % THREADS_PER_VECTOR;
const int stride = THREADS_PER_VECTOR;
const int RPB = REGIONS_PER_BLOCK;
const int TPB = THREADS_PER_BLOCK;

//No Vectorization
vID = threadID;
for(vID; vID < NUM_REGIONS; vID += TPB) {
  int offset = vID*WIDTH;
  for(int i=0; i < WIDTH; i++) {
    A[offset + i] = B[offset + i] + C[offset + i];
  }
}

//Partial Vectorization
vID = threadID / THREADS_PER_VECTOR;
for(vID; vID < NUM_REGIONS; vID += RPB) {
  int offset = vID*WIDTH;
  for(int i=vLane; i < WIDTH; i += stride) {
    A[offset + i] = B[offset + i] + C[offset + i];
  }
}

//Full Vectorization
vID = 0;
for(vID; vID < NUM_REGIONS; vID++) {
  int offset = vID*WIDTH;
  A[offset + vLane] = B[offset + vLane] + C[offset + vLane];
}
```

Listing 4.1: Vectorization examples.

These threads iterate over the vector until all elements have been computed. With full vectorization, a thread is assigned to the operation for each element in the vector. In this case, the ‘for’ loop is unnecessary and is omitted from the code.

Figure 4.3 illustrates the progress of four operations that have been parallelized with four threads using no vectorization, partial vectorization, and full vectorization. In theory, each of these mappings should be equivalent in terms of work and efficiency. However, when these approaches are mapped to the GPU, there are significant differences in terms of thread/shared memory occupancy and warp scheduling. These differences can have a major impact on performance.

4.3.4 Tuning Parameters

Within a filtering operation, for a given polynomial and quadrature order, we adjusted three autotuning parameters: vector width, regions per block, and block count. We used 2D

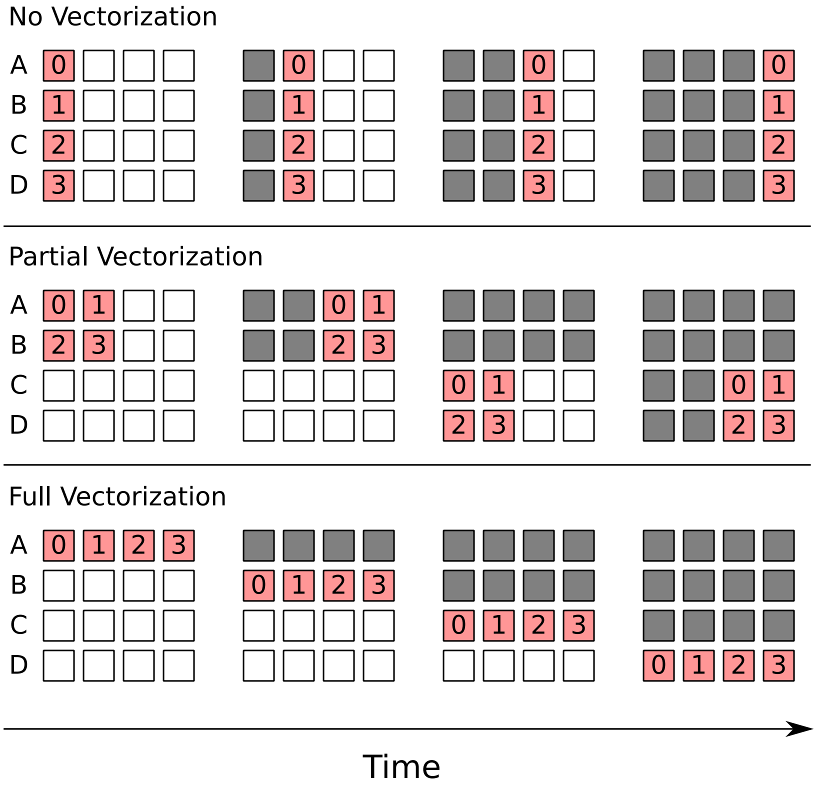


Figure 4.3: Illustration of vectorization with none, partial, and full. Given four operations (A, B, C, D) with four suboperations each and four threads (0, 1, 2, 3), the work can be divided up as shown. Red squares indicate suboperations that will be completed in the next iteration, and gray squares indicate previously completed suboperations.

thread blocks where the x dimension was set to the vector width and the y dimension set to the number of integration instances on which block will operate (regions per block). We autotuned the methods using C++ templates combined with scripts that alter the template values, recompile, and test the new configuration.

- **Parameter 1 - Vector Width:** The vector width is the number of threads assigned to a process in a particular region. We chose values for VECTOR_WIDTH ranging between 1 and the max vector width for each test. We favored numbers that were multiples of 16 or 32, as they divide easily into warps of size 32 which Nvidia GPUs use. VECTOR_WIDTH1 represents the values used in the linear tests, and VECTOR_WIDTH2 represents the values used in the quadratic tests.

$$\begin{aligned} &\text{VECTOR\_WIDTH1} \in \{1, 2, 4, 8, 12, 16, 18, 24, 32, 36\} \\ &\text{VECTOR\_WIDTH2} \in \{16, 32, 48, 64, 80, 96, 112, 128, 144, 150\} \end{aligned}$$

- **Parameter 2 - Regions Per Block:** One of the most important tuning parameters in a GPU kernel is often the block size (number of threads per block). We calculate this based on two independent parameters, the vector width and the region count. The regions per block is the number of independent regions that a block will operate over (i.e., the number of concurrent vectors). BLOCK_SIZE is calculated by multiplying VECTOR_WIDTH and REGIONS_PER_BLOCK.

$$\begin{aligned} &\text{REGIONS\_PER\_BLOCK} \in \{2, 4, 6, 8, 10, 12, 14, 16, 18, 20\} \\ &\text{BLOCK\_SIZE} = \text{VECTOR\_WIDTH} \times \text{REGIONS\_PER\_BLOCK} \end{aligned}$$

- **Parameter 3 - Block Count:** Another important parameter is the number of blocks launched per GPU kernel. The optimal number of blocks is often a multiple of the number of SMs on the card. The GPU we used in our tests has 13 SMs, so we chose multiples of 13 for BLOCK_COUNT.

$$\text{BLOCK\_COUNT} \in \{13, 26, 39, 52, 65, 78, 91, 104\}$$

4.3.5 Implementation

In our implementation, we pair a CPU core with a GPU. The CPU iterates over each element, using the per-element method, and computes the set of points whose stencil intersects that element. This set of subregions is then sent to the GPU while the CPU continues with the next set of intersection tests. On the GPU, the thread blocks iterate over the subregions in a strided fashion (i.e., $\mathcal{P}_{i+k*N_B}$, where $\mathcal{P}_i$ is the i^{th} point, N_B is the number of concurrent blocks, and k is an incrementing integer). Within a block, N_{pq} threads are assigned to each subregion, with each thread computing the value at a specific quadrature point. A reduction is performed over the values produced from each quadrature point, which yields the integrated value over the subregion.

We use a mesh division algorithm that partitions the mesh into nonoverlapping patches [42]. A 3D uniform hash grid is created and each point/element is distributed into the appropriate bins based on its spatial position. The grid is then divided into patches (eight in our tests). A separate memory space is assigned to each patch where the partial solutions are accumulated. A reduction is computed at the end to sum the values and produce the final postprocessed solution for each point.

This tiling approach incurs additional memory overhead based on the number of grid points that fall within the intersection of stencils from multiple patches. The ratio of memory overhead to base memory consumption needed to store the solution is proportional to the ratio of the patch surface area to the patch volume. Each patch stores partial solutions for every grid point that falls within the union of intersections spaces of the elements contained within the patch. Thus, only points near the boundaries of patches will require storing multiple partial solutions.

The final summation of the partial solutions requires only a linear reduction based on the memory offset of each patch solution. In the reduction phase, we divide the grid points based upon which patch they lie in. We then assign a block to each patch that performs the reduction on the partial solutions for those grid points. This patch division eliminates write contention to the final solution space. The reduction contributes a minimal amount of time to the overall process.

4.4 Experimental Results

In this section, we evaluate the performance of our method for dG postprocessing. We ran our tests on a node with an Intel Xeon E5-2640 processor running at 2.5GHz, 128GB of memory, and a NVIDIA Tesla K20c GPU using CUDA 5.0. We compiled our code with *g++*

4.7.2 and *nvcc* 5.5. The dG postprocessing tests were executed on structured tetrahedral meshes consisting of 10^3 hexahedrons, each being subdivided into six tetrahedrons. We used periodic boundary conditions with linear and quadratic polynomials. All tests were conducted with double precision floating point values.

Table 4.1 provides metrics for the GPU and CPU architectures used in our tests. The Intel Xeon E5-2640 has 2.5MB of cache per core, wheras the Nvidia K20c has 0.85kB of cache per core. This small amount of chache on the K20c is somewhat offset having roughly 5× higher memory bandwidth than the Intel Xeon. GPUs generally perform better with algorithms that have high FLOPS/byte ratios due to their limited cache size and streaming SIMD architecture.

In our autotuning method, we used three adjustable parameters, vector width, regions per block, and block count. We set the x dimension of the thread blocks to the vector width and the y dimension to the number of integration instances (regions per block) that a block will compute. Figure 4.4 illustrates our autotuning results across varying vector widths and regions per block. Each cell represents the best results for a given vector width and region per block size across the range of block counts. Figure 4.5 provides all the runtime results for the autotuning tests, ordered by vector width.

A full integration computation requires $IW * (QX + QY * QZ + QZ + 1) + 21$ doubles (8 bytes) of memory in cache/shared memory. For linear polynomials, $IW = 36, QX = 3, QY = 2, QZ = 2$, which yields 3048 bytes per integration operation. Quadratic polynomials require significantly more memory with $IW = 150, QX = 4, QY = 3, QZ = 3$, yielding 20568 bytes of cache/shared memory per integration. This computation consumes a significant portion of the 48k shared memory per SM on the NVIDIA K20c. Up to two

Table 4.1: CPU/GPU comparison

	Intel Xeon E5-2640	Nvidia Tesla K20c
Number of Cores	6	2496 (13 × 192)
Clock Speed	2.50 GHz	0.71 GHz
Cache	15 MB	2.06 MB
Cache Per Core	2.5 MB	.85 kB
Peak FLOPS (Double Precision)	120 GFLOPS	1.17 TFLOPS
Memory Bandwidth	42.6 GB/s	208 GB/s

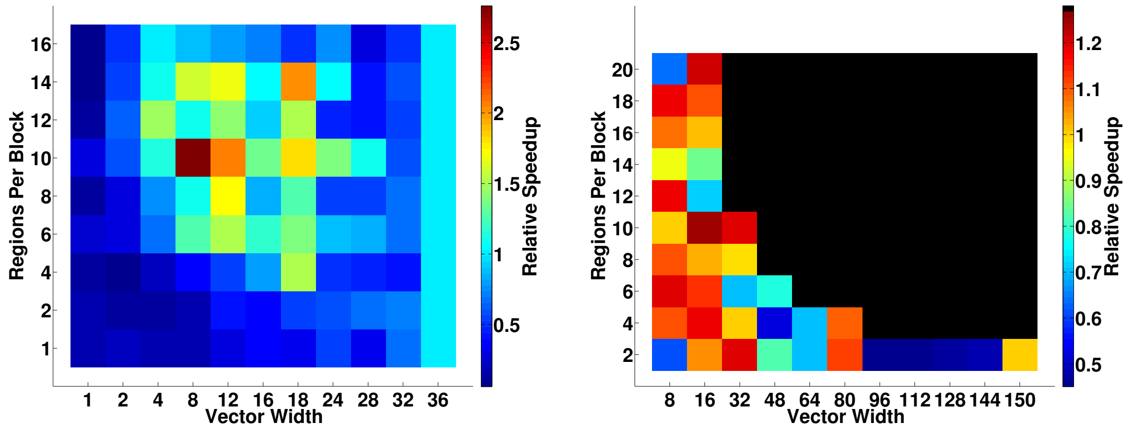


Figure 4.4: Autotuning results (left: linear, right: quadratic): Each square represents the best result from varying the block count for a given combination of vector width and regions per block. In the right figure, the black cells represent combinations that could not be tested without exceeding the shared memory resource limit on the GPU.

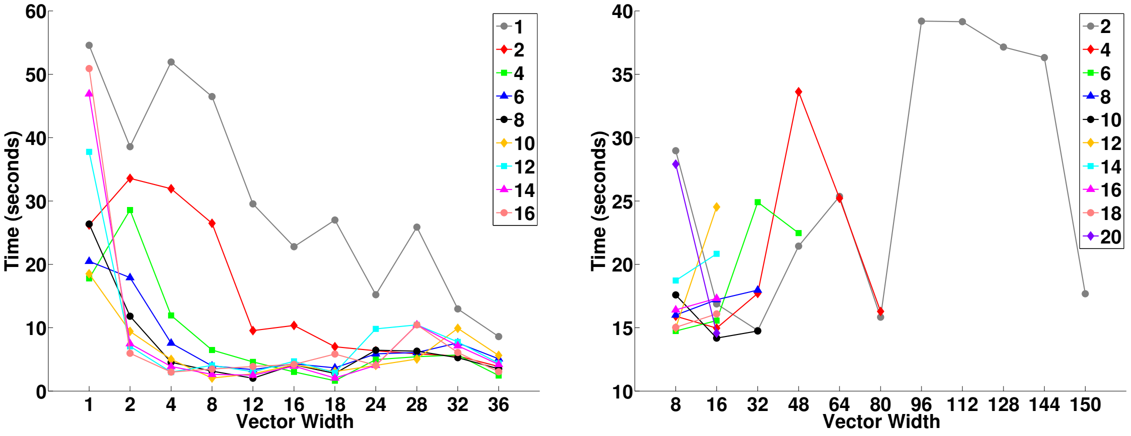


Figure 4.5: Autotuning results (left: linear, right: quadratic) illustrating all results with times, organized by vector width. Colors correspond to number of vectors per block.

integration operations could be stored, leaving close to 8k shared memory still available. Allowing for partial vectorization increases the cache utilization, which can offer significant boosts in performance.

Since each quadrature point on the mesh can be processed independently, the complexity of the postprocessing grows linearly with respect to the number of mesh elements. Using linear polynomials on structured tetrahedral meshes, each quadrature point will have to check for intersections with a 5 × 5 region of hexahedral elements around it. There are approximately 750 possible intersections (with each intersection being subdivided into up to

12 smaller tetrahedrons) per point, followed by the subsequent integration of those regions. For even a small mesh ($10^3 \sim 6000$ Tetrahedrons) the number of intersection/integrations equates to $\sim 18 \times 10^6$. This process will be even more computationally intense for meshes with higher variance in element size.

Figure 4.6 provides the results for relative speed-ups of integration over linear and quadratic polynomials with varying vector widths compared to a full vectorization baseline. Each individual bar represents the minimum runtime of a set of results that were autotuned for the given vector width. We used an exhaustive search method to find the optimal configuration among our set of parameters. The results have been normalized with respect to the time taken to integrate with full vectorization (black bar). Green bars indicate results that are faster than full vectorization and red bars indicate results that are slower. For linear polynomials, we found that vector widths of 8, 12, and 18 provide a speed-up of 19%, 20%, and 50%, respectively. For quadratic polynomials, we found that vector widths of 8, 16, 32, and 80 provide a speed-up of 20%, 25%, 20%, and 12%, respectively.

On one end of the spectrum there are nonvectorized computations with one thread per operation. Nonvectorized operations generally incur significant loop overhead and reduce the likelihood of coalesced memory accesses. On the other end are fully vectorized computations, which often do not have the highest shared memory utilization. Partial vectorization widths in the middle of the spectrum show significant performance improvements for linear polynomials. Vectorization widths close to half of the total vector width demonstrate the

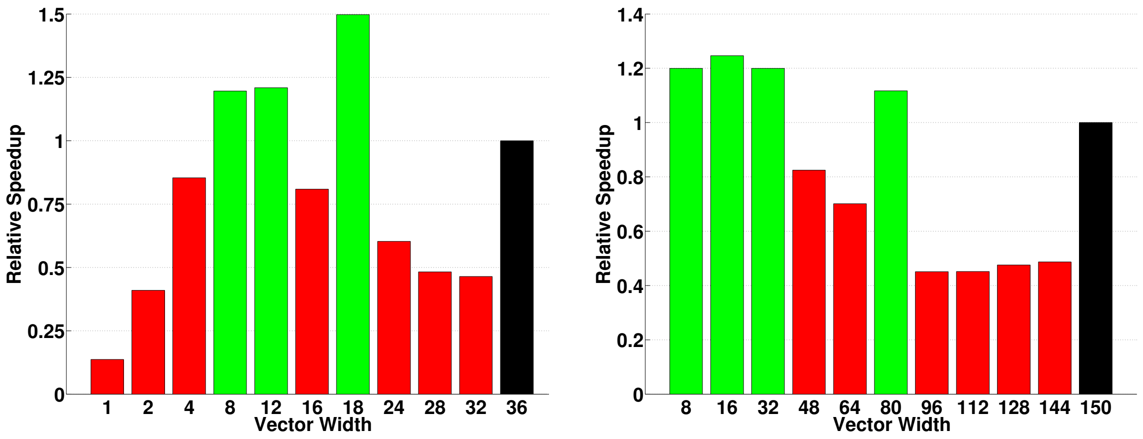


Figure 4.6: Results based on varying vector widths. Values have been normalized with respect to full vectorization (black bar). Green > 1.0, Red < 1.0, Black = 1.0.

best performance with quadratic polynomials.

These autotuning results indicate that determining the optimal combination of parameters is nonobvious and best suited for an automated search of some sort. In the linear polynomial integration, the vector width of 18 exhibits a 50% performance improvement whereas a vector width of 16 incurs a 20% performance degradation. This pattern is also witnessed in the quadratic integration results, where vector widths of 8, 16, 32, and 80 see improvements, but 48 and 64 do not. Typically vector widths are chosen to be multiples of 32 and 64 to conform with the size of work groups used on the GPUs. It is generally not clear to the programmer which combination will yield the best result without extensive testing.

Figure 4.7 provides the FLOPS measurements for the linear and quadratic tests. We see significant gains in terms of FLOPS over full vectorization in some cases. Analysis through use of the Nvidia CUDA visual profiler indicates these gains in performance are due to better use of cache/shared memory. We also find that vector widths close to full vectorization often incur significant performance degradation. This degradation is likely due to the lack of any extra instances of integration operations fitting in memory while requiring loop overhead and additional passes of the integration operation to span the whole vector.

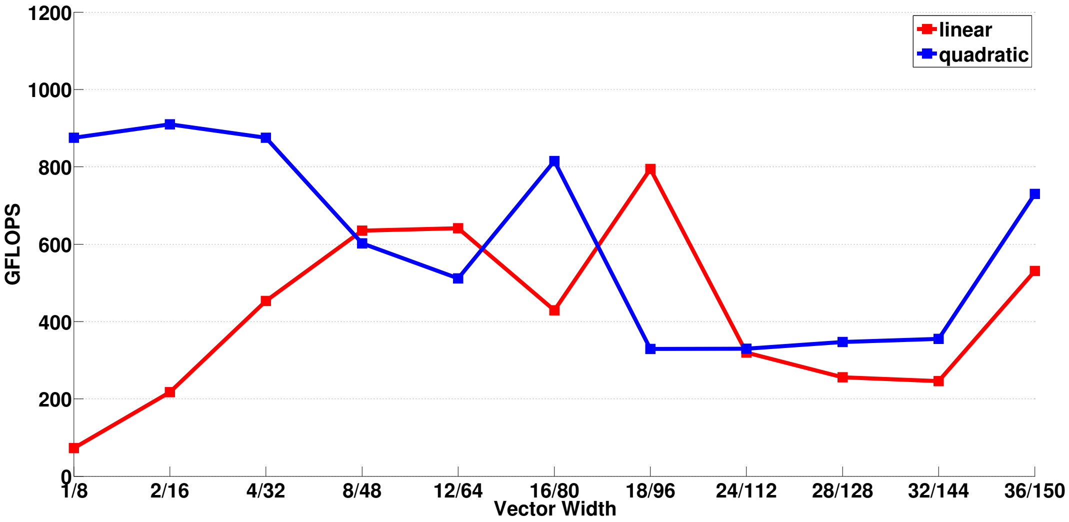


Figure 4.7: FLOPS measurements for linear and quadratic polynomials. (Vector width: linear/quadratic)

CHAPTER 5

DYNAMIC COMPRESSED SPARSE ROW

5.1 Sparse Matrices

5.1.1 Sparse Matrix Formats

The *coordinate* (COO) format is the simplest sparse-matrix format. It represents a matrix with three vectors holding the row indices, column indices, and values for all nonzero entries in the matrix. The entries within a COO format must be sorted by row in order to efficiently perform an SpMV operation. SpMV operations are conducted in parallel through segmented reductions over the length of the arrays. Tracking which thread has processed the final entry in a row requires explicit interthread communication.

The *compressed sparse row/column* (CSR/CSC) formats have arrays that fully store two of the three sets, either the column indices or the row indices in addition to the values. Either the rows or columns (in CSR or CSC, respectively) are compressed to store only the offsets into the other two arrays. For CSR, entry i and $i + 1$ in the row offsets array will store the starting and ending offsets for row i. CSR has been shown to be one of the best formats in terms of memory usage and SpMV efficiency due to its fully compressed nature and has become widely used [34]. CSR has a greater memory efficiency than COO, which is a significant factor in speeding up SpMV operations due to decreased memory bandwidth usage.

The *ellpack* (ELL) format uses two arrays, each of size $m \times k$ (where m is the number of rows and k is a fixed width), to store the column indices and the values of the matrix [30, 31]. These arrays are stored in column-major order to allow for efficient parallel access across rows. This format is best suited for matrices that have a fixed number of entries per row. Allocating enough memory in each row to store the entire matrix is prohibitively expensive for ELL when a matrix contains even one long row.

The *hybrid-ellpack* (HYB) format offers a compromise by using a combination of ELL and COO. It stores as many entries as possible in an ELL portion, and the overflow from rows with a number of entries greater than the fixed ELL width is stored in a COO portion. ELL and HYB have become popular on SIMD architectures due to the ability of thread warps to look through consecutive rows in an efficient parallel manner [12].

The diagonal format (DIA) is best suited for banded matrices. It is formed by two arrays which store the nonzero data and the offsets from the main diagonal. The nonzero values are stored in an $m \times k$ array where m is the number of rows in the matrix and k is the maximum number of nonzeros out of any row in the matrix. The offsets are stored with respect to the main diagonal, with positive offsets being to the right and negative offsets being to the left. The SpMV parallelization of this format is similar to that of ELL with one thread/vector being assigned to each row in the matrix. The values array is statically sized, similar to ELL, which restricts its ability to handle insertions.

A number of other specialized sparse-matrix formats have been developed, including jagged diagonal storage (JDS), block diagonal (BDIA), skyline storage (SKS), tiled COO (TCOO), block ELL (BELL), and sliced-ELL (SELL) [55], all of which offer improved performance for specific matrix types. Blocked variants of these and other formats work by storing localized entries in blocks for better data locality and a reduction in index storage. “Cocktail” frameworks that mix and match matrix formats to fit specific subsets of the matrix have been developed, but they require significant preprocessing and are not easily modified dynamically [83]. Garland et al. have provided detailed reviews of the most common sparse matrix formats [30, 31, 91], as well as an analysis of their performance on throughput-oriented many-core processors [13].

Block formats such as BRC [6] and BCCOO [96] have limited ability to add in additional entries. BRC can add new entries only if those entries correspond to zeros within blocks that have been stored. BCCOO can handle the addition of new entries, but it suffers from many of the same problems as COO. Also, new insertions will not always follow a blocked structure, so additional blocks may be sparse, which lowers memory efficiency.

Many sparse matrix formats are fully compressed and do not allow additional entries to be added to the matrix dynamically. Adding additional entries to a CSR matrix requires rebuilding the entire matrix, since there is no free space between entries. Of existing formats, COO is the most amenable to dynamic updates because new entries can be placed at the end of the data structure. However, updating a COO matrix in parallel requires atomic operations to keep track of currently available memory locations. The ELL/HYB formats allow for some additional entries to be added in a limited fashion. ELL cannot add in more entries per row than the given width of the matrix, and while the HYB format has a COO matrix to handle overflow from the ELL portion, it cannot be efficiently updated in parallel since atomic operations are required and the COO portion must maintain the sorted property.

5.1.2 Sparse Matrix Algorithms on the GPU

A great deal of research has been devoted to improving the efficiency of SpMV, which has been studied on both multicore and many-core architectures. Williams et al. demonstrated the efficacy of using architecture-specific data structures to optimize performance [94, 46]. Since SpMV is a bandwidth-limited operation, research has also produced other methods, such as automatic tuning, blocking, and tiling, to increase cache hit rates and decrease bandwidth usage [97, 21, 69].

The two most common CSR SpMV algorithms are CSR-scalar and CSR-vector. CSR-scalar assigns one thread per row and CSR-vector assigns a vector of threads to each row. On SIMD architectures the vector size generally never exceeds a full warp (to avoid explicit synchronization between threads). A vectorized approach allows for more efficient coalesced memory accesses. A hybrid approach has been shown to be effective. This method selectively picks between CSR-scalar and CSR-vector based on the row length [34]. Adaptive algorithms that group rows together by length and assign separate kernels to each group have also been explored [5].

Graph applications often use sparse binary adjacency matrices to represent graphs and translate graph operations to linear algebraic operations [40]. Finding the transitive closure of a graph can be done through repeated multiplication of its adjacency matrix. The transitive closure of an adjacency matrix R calculates the union of successive powers (R^i) of the matrix. The result is R^i having a nonzero between any pair of nodes connected by a path of length i. Thus, the union (addition/binary-or) of all $R, \dots R^n$ will have a nonzero entry for every pair of nodes that are connected by a path of length $\leq n$. This process of unioning successive powers of R can be continued until a fixed point is reached. All nodes that are connected by a path of any length will be marked in the matrix.

Bandwidth limited sparse matrix-matrix operations such as sparse matrix-matrix addition $A + B = C$ and sparse matrix-matrix multiplication $AB = C$ remain difficult to compute efficiently. These operations require creating a new sparse matrix C whose entries and sparsity will depend on the sparsity patterns of A and B, and often will have a differing number of elements than either. Current implementations generally look globally at both matrices and find the intersection patterns using temporary workspace memory, after which the new matrix C can be generated [14, 41], which often involves format conversions that consume additional time and memory.

5.2 Dynamic Compressed Sparse Row (DCSR)

Dynamic compressed sparse row (DCSR) uses a method of dynamic allocation to add additional entries without rebuilding the matrix. DCSR uses a *row offset array*, representing a dense array of ordered rows, and for each a fixed number of *segment* offsets. The column indices and values are stored in arrays that are logically divided into these data segments in the same way that CSR row offsets partition the column indices and values. Each such segment is a contiguous portion of memory that stores entries within a row. Segments may contain more space than entries to allow for future insertions. The contiguous layout of entries within the set of segments for a given row is equivalent to the corresponding row in CSR format.

Initializing the matrix can be done in one of two ways. Either a matrix can be loaded from another format (e.g., COO or CSR) or the matrix can be initialized as blank. In the latter case, each row is assigned an initial number of entries (an initial segment size) in the column indices and values arrays. The row offset array is initialized with space for k segment offset pairs, with either no allocated segments or a single allocated segment of size μ per row. The latter case will consume the same amount of memory as an ELL matrix with a row width of μ, except in row-major order instead of column-major order. To allow for dynamic allocation, we maintain a larger memory buffer than needed and use simple bump-pointer allocation to add new segments. This allocation pointer is set to the end of the currently used space ($rows \times \mu$ in the case of a new matrix). A maximum size of memory buffer for the columns and values arrays is specified by the user. Figure 5.1 provides an illustrative comparison of CSR, HYB, and DCSR formats.

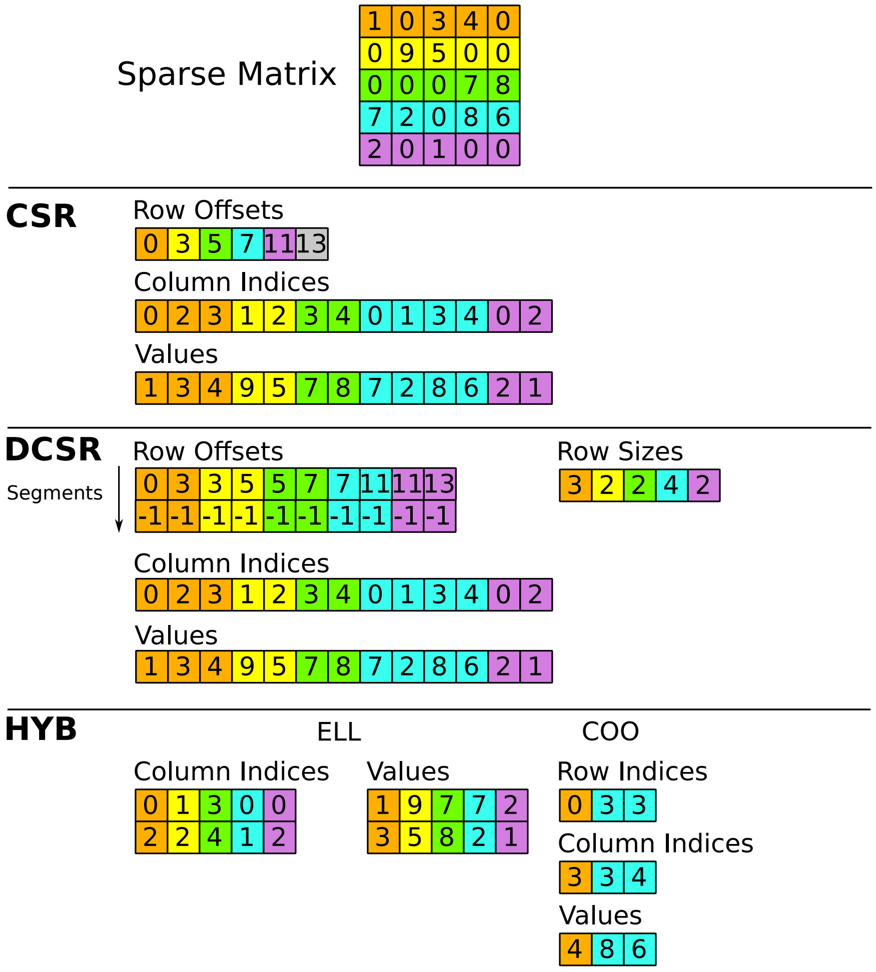


Figure 5.1: Comparison of CSR, DCSR, and HYB formats.

The format consists of four arrays for column indices, values, row offsets, and row sizes, in addition to a memory allocation pointer. The row offsets array functions in a similar manner to that of its CSR counterpart, except that both a beginning and ending offset are stored and space exists for up to k such pairs per row. This table is encoded as a strided array where the starting and ending offsets of segment k in row i are indexed by $(i * 2 + k * pitch)$ and $(i * 2 + k * pitch + 1)$, respectively. The $pitch$ may be defined as a value convenient for cache performance such that $pitch \geq 2 * rows$. Each set of offsets for a given segment lies within a different cache line, which serves to increase memory aligned accesses. The number of memory segment offset pairs (the max k) is an adjustable parameter specified at matrix construction. The column indices and values correspond 1:1, just as in CSR. Unlike CSR, however, there may be more than one memory segment assigned to a given row, and the segments need not be contiguous. As the last segment for a row may not be full, the actual row sizes are maintained so the used portion of each segment is known.

Explicitly storing row sizes allows for optimization techniques such as the adaptive binning strategy used in *adaptive CSR* (ACSR)[5]. This optimization implements customized kernels to process bins of specified row-lengths. We make use of this optimization by binning rows together based on row size before SpMV or SpMM operations. Each row is given a bin label based on its size (1, 2-3, 4-8, 9-16, 17-32, ...). A permuted set of row indices is created by sorting according to these bin labels. Bin-specific kernels are launched with these permuted indices on separate streams, which allows each kernel to easily access the rows that it needs to process without scanning over the matrix.

When inserting new elements within a row, the last allocated segment for that row is located, and if space is available the new elements are inserted in a contiguous fashion just after current entries. If that segment does not have enough room, a new segment will be allocated with the appropriate size plus an additional amount α. The α value represents additional "slack space" and allows for a greater number of entries to be inserted without the creation of a new segment. If dynamic updates follow a power-law distribution, there will be a higher probability of additional entries being inserted into longer rows. Although we experimented with setting α to be a factor of the previous segment size, for our tests we settled on a value of μ (average row size of matrix). When a new segment is allocated, the memory allocation pointer is atomically increased by the size of the new segment. A hard limit on these additions, before defragmentation is required, is fixed by the number of segments k. The defragmentation operation always reduces the number of segments in each row to one, which allows the format to scale to an arbitrary number of allocations.

Algorithm 4 provides pseudocode illustrating new segment allocation. This allocation function can be parallelized across rows, as each vector of threads will execute this function on a different row. Within a row, a vector of threads operate together to add new elements into matrix A from an array of values B (B_offsets, B_cols, B_vals). The segments could be of variable length, so the total size is computed by looping over the segments and summing the differences of the starting and ending offsets (*A_start, A_end*). The current available memory is calculated by computing the difference of the final segment ending offset and index of the last element (*A_end* − *A_start*). If there is enough room, the elements

Algorithm 4: Allocate Segments

```
Input: sizes, offsets, Aj, Ax, B_offsets, B_cols, B_vals
Output: sizes, offsets, Aj, Ax
$row \leftarrow vid$ ; // vector ID
while $row < n\_rows$ do
    $sid \leftarrow 0$ ; // segment index
    $rl \leftarrow$ sizes[$row$] ; // row length
    $idx \leftarrow 0$ ; // thread row index
    $start \leftarrow$ offsets[$row * 2$] ; // starting segment offset
    $end \leftarrow$ offsets[$row * 2 + 1$] ; // ending segment offset
    $free\_mem \leftarrow 0$;
    $B\_start \leftarrow$B_offsets[$row * 2$];
    $B\_end \leftarrow$B_offsets[$row * 2 + 1$];
    $rlB \leftarrow B\_row\_end - B\_row\_start$;
    if $rlA \geq 0$ then
        while $A\_idx < rlA$ do
            $idx \leftarrow idx + (A\_end - A\_start)$;
            if $idx < rlA$ then
                $sid \leftarrow sid + 1$;
                $A\_start \leftarrow$ offsets[$sid$*pitch+$row * 2$];
                $A\_end \leftarrow$ offsets[$sid$*pitch+$row * 2 + 1$];
        $idx \leftarrow A\_end + rlA - idx$;
    else
        $idx \leftarrow A\_start$;
    $free\_mem \leftarrow A\_end - A\_start$;
    if $lane = 0$ AND $free\_mem < rlB$ AND $rlB > 0$ then
        // allocate new space
        $size \leftarrow rlB - free\_mem + \alpha$;
        $addr \leftarrow$ atomicAdd(sizes[n_rows], $size$);
        // allocate new row segment
        offsets[($sid + 1$)*pitch + $row * 2$] $\leftarrow addr$;
        offsets[($sid + 1$)*pitch + $row * 2 + 1$] $\leftarrow addr + size$;
    // Allocate new entries (Algorithm 2)
    Insert_Elements();
    $row \leftarrow row$ + num_vectors;
```

are inserted into the remaining space; otherwise a new segment must be allocated. This allocation is performed by atomically incrementing the memory offset pointer to allocate a new segment of memory of size equal to new elements minus the remaining free space plus an α value. The returned address *addr* is the beginning offset of the new segment of size *size*. Afterward, the new elements are inserted via Algorithm 5.

Algorithm 5: Insert Elements

```
Input: sizes, offsets, Aj, Ax, B_cols, B_vals
Output: sizes, Aj, Ax
$B\_idx \leftarrow B\_start$+lane ; // add thread lane
while $B\_idx < B\_end$ do
    if $idx \geq A\_end$ then
        $pos \leftarrow idx - A\_end$;
        $sid \leftarrow sid + 1$;
        $A\_start \leftarrow$ offsets[$sid$*pitch+$row * 2$];
        $A\_end \leftarrow$ offsets[$sid$*pitch+$row * 2 + 1$];
        $idx \leftarrow A\_start + pos$;
    Aj[$idx$] ← B_cols[B_idx];
    Ax[$idx$] ← B_vals[B_idx];
    $B\_idx \leftarrow B\_idx$+ VECTOR_SIZE;
    $idx \leftarrow idx$+ VECTOR_SIZE;
if $lane = 0$ then
    sizes[$row$] ← sizes[$row$] + $rlB$;
```

When inserting new elements into the matrix, it is possible that duplicate nonzero entries (i.e., two or more entries with the same row and column index) will be added. Duplicate entries are handled in one of two ways. The first method is to simply let them accumulate, which does not pose a problem for many operations. SpMV operations are tolerant of duplicate entries due to the distributive property of the inner product and will yield the same result to within floating point tolerance. For binary matrices, the row-vector inner products will produce the same result irrespective of duplicate nonzeros. A second solution is to perform a segmented reduction on the entries after sorting by row and column, which combines all entries with matching row and column indices into a single entry. This full reduction is generally not needed when performing only SpMV and addition operations. Sparse matrix-matrix multiplication (SpMM) operations may cause significant fill-in, which would require such a reduction to be performed. In our SpMV tests, we let the values accumulate for all formats as they do not hinder the SpMV operations that are performed.

Algorithm 5 provides pseudocode for the insertion operation. A vector of threads will operate together to add the elements into the segments. After a segment is full, the next segment indices are retrieved from the offsets table whose starting and ending offsets are *A_start* and *A_end*, respectively. Column indices and values are copied from *B_cols* and *B_vals* to their respective locations in the A matrix. After this is complete, a single thread

Algorithm 6: DCSR SpMV

```
Input: sizes, offsets, Aj, Ax, x, y
Output: y
$tid \leftarrow$ thread index ; // thread ID
$lane \leftarrow tid \,\%\, Vec\_Size$ ; // lane ID
$vid \leftarrow tid \,/\, Vec\_Size$ ; // vector ID
for $row \leftarrow vid$ to $num\_rows$, $row$ += $num\_vecs$ do
    $idx \leftarrow 0$ ; // thread row index
    $rl \leftarrow$ sizes[$row$] ; // row length
    $sid \leftarrow 0$ ; // segment index
    while $idx < rl$ do
        $start \leftarrow$ offsets[$sid$*pitch + $row * 2$];
        $end \leftarrow$ offsets[$sid$*pitch + $row * 2 + 1$];
        /* accumulate local sums                                  */
        for $j \leftarrow start$ to $end$, $j$ += $Vec\_Size$ do
            sum += Ax[$j$] * x[Aj[$j$]];
        idx += ($end$ - $start$);
    y[row] = sum;
```

will update the row sizes array to reflect the new size.

An SpMV operation works as follows. The first pair of segment offsets is fetched. The entries within the corresponding segment are multiplied by the appropriate values in x according to the algorithm being used (CSR-scalar, CSR-vector, etc.). If the row size is greater than the capacity of the current memory segment, the next pair of offsets is fetched. If the size of the current segment plus the running sum of the previous segment sizes is greater than or equal to the row size, this is the final segment of the row. In case the final segment is not full, the location of the last entry can be determined by the difference of the row size and the running sum. This process continues until the entire row has been read.

As the matrix accumulates more segments, SpMV performance decreases slightly. A fixed number of segments also means this process cannot continue forever. Our solution to both problems is to implement a defragmentation operation that compacts all the entries within the column indices and values arrays, eliminating empty space. This defragmentation step combines all the segments in a row into a single segment that compactly stores the entire row. This operation may be invoked periodically, or more conservatively when a row has reached its maximum capacity of segments. In practice we do the latter and set a flag when any row reaches its maximum segment count. At this point we consider defragmentation to

Algorithm 7: Defragment DCSR

```
Input: sizes, offsets, Aj, Ax
Output: offsets, Aj, Ax
/* prefix sum on row sizes                                  */
exclusive_scan(sizes, temp_offsets);
new T_cols(size(Aj)), new T_vals(size(Ax));
CompactIndices(T_cols, T_vals, temp_offsets, Aj, Ax, offsets, sizes);
/* shallow copy, old arrays deleted                         */
Aj = &T_cols, Ax = &T_vals;
SetRowOffsets(offsets, sizes, temp_offsets);
```

be required. Algorithm 6 illustrates the SpMV operation, which is performed in a similar fashion to CSR-vector, except that there is an outer loop over the segments.

Defragmentation performs the equivalent to a sort by row operation on the entries of the matrix; we formulated a method that does not require an actual sort and is significantly faster than doing so. Since we explicitly store row sizes, we perform a prefix-sum operation on them to calculate the new row offsets in a compacted CSR form. The entries are then shuffled from their current indices to their new indices in newly allocated column indices and values buffers, after which we set a pointer in our data structure to these new arrays and free the old buffers (shallow copy). By using the knowledge of the row sizes to compute resulting offsets and indices, we eliminate the need to do any comparisons in this operation, which greatly improves performance. The defragmentation process is described by Algorithm 7.

Figure 5.2 illustrates an example of inserting new elements into a DCSR matrix. Initially, the matrix has four populated rows with the memory allocation pointer being 16. Row 0 can insert 1 additional entry in its current segment before a new segment would need to be allocated. Rows 1 and 2 have enough room for two additional entries, but row 3 is full. Figure 5.2 shows a set of new entries that are inserted into rows 0, 2, and 3. In this case, a new segment of size 4 is allocated for row 0 and row 3. The additional segments need not be consecutive nor in order of row since the exact offsets are stored for each segment. Finally, the defragmentation operation computes new segment offsets from the row sizes. The entries are shuffled to their new indices, which results in a single compacted segment for each row.

As CSR is the most commonly used sparse matrix format, we designed DCSR to be compatible with CSR algorithms and to allow for easy conversion between the formats. Minimal

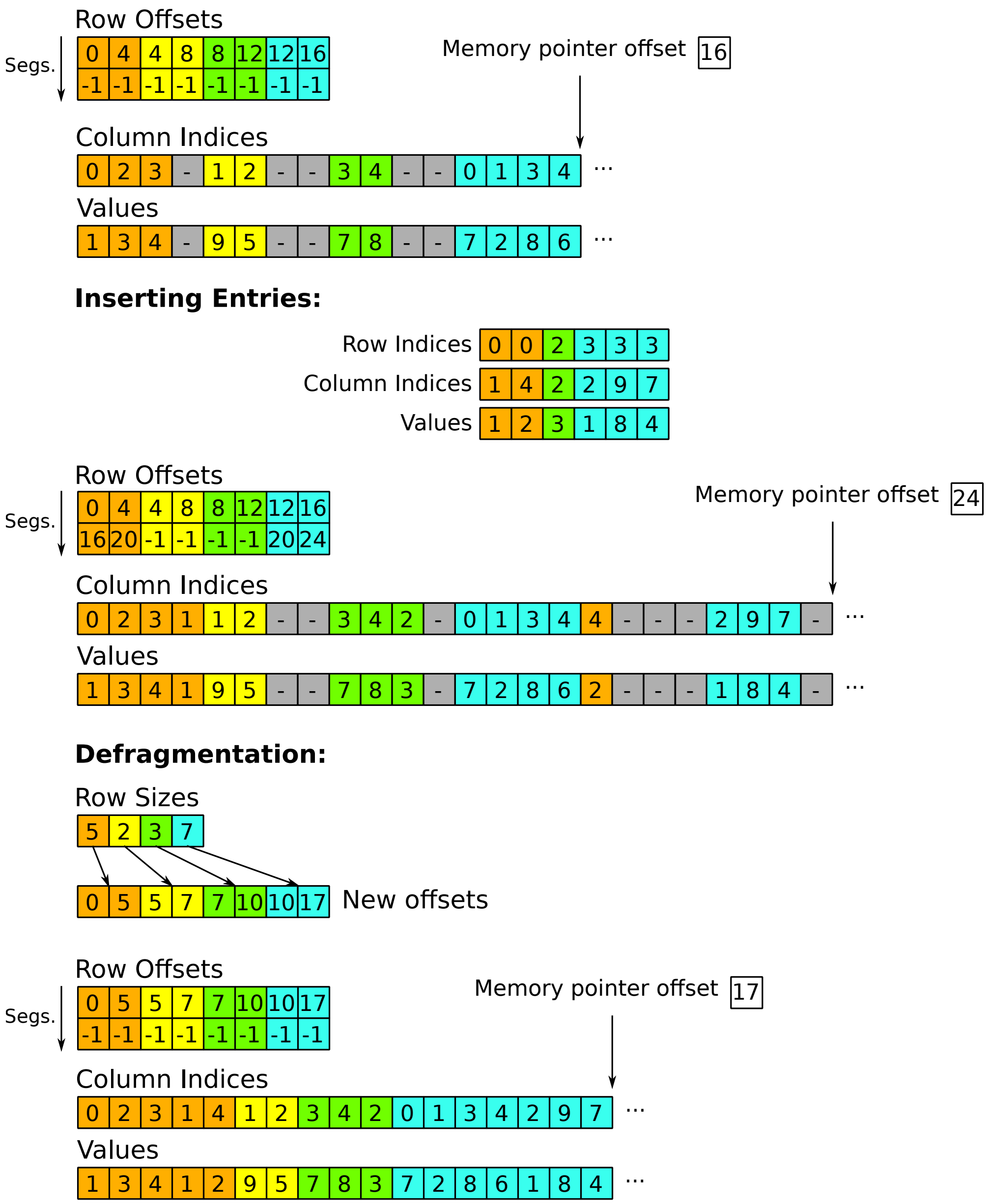


Figure 5.2: Illustration of insertion and defragmentation operations with DCSR.

overhead is required to convert from CSR to DCSR and vice versa. When converting from CSR to DCSR, the column indices and values arrays are copied directly. For the row offsets array, the i^{th} element is copied to indices $i * 2 - 1$ and $i * 2$ for all elements except the first and last one. A simple subtraction must be performed to calculate the row sizes from the row offsets. Converting back is equally simple, assuming the matrix is first defragmented; the column indices and values arrays are copied back, and the starting segment offset from each row is copied into the row offsets array.

5.3 Experimental Results

To benchmark SpMV, SpMM, update, and conversion performance, we used a node with an Intel Xeon E5-2640 processor running at 2.50GHz, 128GB of memory, and a NVIDIA Tesla K20c GPU. For additional scaling tests, we used an Intel Xeon E5630 processor running at 2.53GHz, 128GB of memory, and 8 NVIDIA Tesla M2090 GPUs. We compiled using *g++* 4.7.2, CUDA 7.5, CUSP 0.5.1, and Thrust 1.8.1, comparing our method against modern implementations in CUSP [14]. Table 5.1 provides a list of the matrices that we used in our tests as well as their sizes, number of nonzeros, and row entry distributions. All the matrices can be found in the University of Florida sparse-matrix database [27].

Memory consumption is a major concern for sparse matrix formats, as one of the primary reasons for eliminating the storage of zeros is to reduce the memory footprint. The ELL component of HYB is best suited to store rows with an equal number of entries. If there is a large variance in row size, much of the ELL portion may end up storing zeros, which is inefficient. We provide a comparison of memory consumption for HYB, DCSR (using 2, 3, and 4 segments), and CSR formats in Table 5.2. We compute the storage size of the HYB format using an ELL width equal to the average number of nonzeros per row (μ) for the given matrix. CSR has the smallest memory footprint since its row indices have been

Table 5.1: Matrices used in tests. NNZ: total number of nonzeros, μ: average row size, σ: standard deviation of row sizes, Max: maximum row size.

Matrix	Abbr.	NNZ	Rows \ Cols	$\mu \setminus \sigma \setminus$ Max
amazon-2008	AMA	5M	735K	7 \ 4 \ 10
cnr-2000	CNR	3M	325K	9 \ 21 \ 2716
dblp-2010	DBL	807K	326K	2 \ 4 \ 154
enron	ENR	276K	69K	3 \ 28 \ 1392
eu-2005	EU2	19M	862K	22 \ 29 \ 6985
flickr	FLI	9M	820K	11 \ 87 \ 10K
hollywood-2009	HOL	57M	1139K	50 \ 160 \ 6689
in-2004	IN2	16M	1382K	12 \ 37 \ 7753
indochina-2004	IND	194M	7414K	26 \ 216 \ 6985
internet	INT	207K	124K	1 \ 4 \ 138
kron-18	KRO	10M	262K	40 \ 261 \ 29K
ljournal-2008	LJO	79M	5363K	14 \ 37 \ 2469
rail4284	RAL	11M	4K \ 1M	2633 \ 4K \ 56K
soc-LiveJournal1	SOC	68M	4847K	14 \ 35 \ 20K
webbase-1M	WEB	3M	1000K	3 \ 25 \ 4700
wikipedia-2005	WIK	19M	1634K	12 \ 31 \ 4970

Table 5.2: Comparison of memory consumption between HYB, CSR, and DCSR formats. Size of HYB is listed in bytes (using ELL width of μ), and sizes for DCSR and CSR are listed as a percent of the HYB size.

Matrix	HYB size	DCSR 2 segs.	DCSR 3 segs.	DCSR 4 segs.	CSR
AMA	54M	0.924	1.026	1.128	0.77
CNR	47M	0.626	0.679	0.732	0.547
DBL	12M	0.86	1.052	1.245	0.572
ENR	4M	0.653	0.762	0.871	0.489
EU2	236M	0.675	0.703	0.731	0.633
FLI	160M	0.546	0.585	0.624	0.487
HOL	859M	0.531	0.541	0.551	0.516
IN2	229M	0.654	0.7	0.746	0.585
IND	2791M	0.571	0.591	0.612	0.541
INT	4M	0.761	0.969	1.177	0.449
KRO	171M	0.493	0.505	0.516	0.475
LJO	1152M	0.594	0.63	0.665	0.541
RAL	149M	0.577	0.577	0.577	0.576
SOC	1009M	0.595	0.631	0.668	0.54
WEB	40M	0.966	1.155	1.344	0.682
WIK	276M	0.635	0.68	0.725	0.567

compressed to the number of rows in the matrix. We see that DCSR has a significantly smaller memory footprint in almost all test cases. Test cases such as AMA and DBL have lower memory consumption for HYB than for DCSR (with 3 and 4 segments), because these matrices have a low row size variance. DCSR with 4 segments uses 20% less memory on average than HYB.

Conversion times between formats are often a key factor when determining the efficacy of a particular format. High conversion times can be a significant hindrance to efficient performance. Architecture-specific formats may provide better performance, but unless the rest of the code base uses that format, the conversion time must be accounted for. We provide the overhead required to convert to and from CSR and COO matrices in Table 5.3. The conversion times have been normalized against the time required to copy CSR → CSR. The conversion times to DCSR are only slightly higher compared to that of CSR. HYB requires significant overhead as the entries must first be distributed throughout the ELL portion and the remaining overflow entries distributed into the COO portion.

Table 5.3: Comparison of relative conversion times. Conversions are normalized against time to copy CSR→CSR.

From To	COO CSR	COO DCSR	COO HYB	CSR DCSR	CSR HYB	DCSR CSR
AMA	2.93	3.03	9.22	1.06	9.25	0.9
CNR	2.24	2.62	14.84	1.04	13.62	0.87
DBL	4.34	5.74	18.07	1.17	16.83	1.1
ENR	5.56	5.95	27.15	1.29	26.95	1.14
EU2	2.1	2.29	16.08	1.06	15.67	0.99
FLI	2.13	2.5	23.29	1.06	19.74	0.96
HOL	1.82	1.9	20.37	1.01	20.3	0.99
IN2	2.15	2.42	18.12	1.06	18.15	0.98
IND	1.93	1.98	∞	1.03	∞	1.01
INT	12.07	13.74	21.38	1.3	15.12	1.0
KRO	1.78	2.09	24.01	1.0	20.14	0.91
LJO	2.09	2.19	19.96	1.02	19.97	0.98
RAL	1.73	2.03	20.67	1.0	17.97	0.91
SOC	2.22	2.35	20.47	1.06	20.41	1.01
WEB	2.89	3.19	11.45	1.16	11.56	0.86
WIK	2.18	2.42	20.13	1.07	20.11	0.98

5.3.1 Matrix Updates

To measure the speed of dynamic updates, we ran two series of tests that involved streaming updates and iterative updates. In the streaming updates test, the matrix was incrementally built up by continuously inserting new entries. The elements are first buffered into three arrays, representing the rows indices, column indices, and values. We initialize the matrix sizes according to the average number of nonzeros for the given input. The entries are then added in a streaming parallel fashion to the matrices.

Updating a HYB matrix first requires checking the ELL portion, and if the row in question is full, inserting the new entry into the COO portion. Any updates to the COO portion require atomic operations to ensure synchronous writes between multiple threads. These atomic updates are prohibitive for fast parallel updates as all threads are contending to insert entries onto the end of the COO matrix.

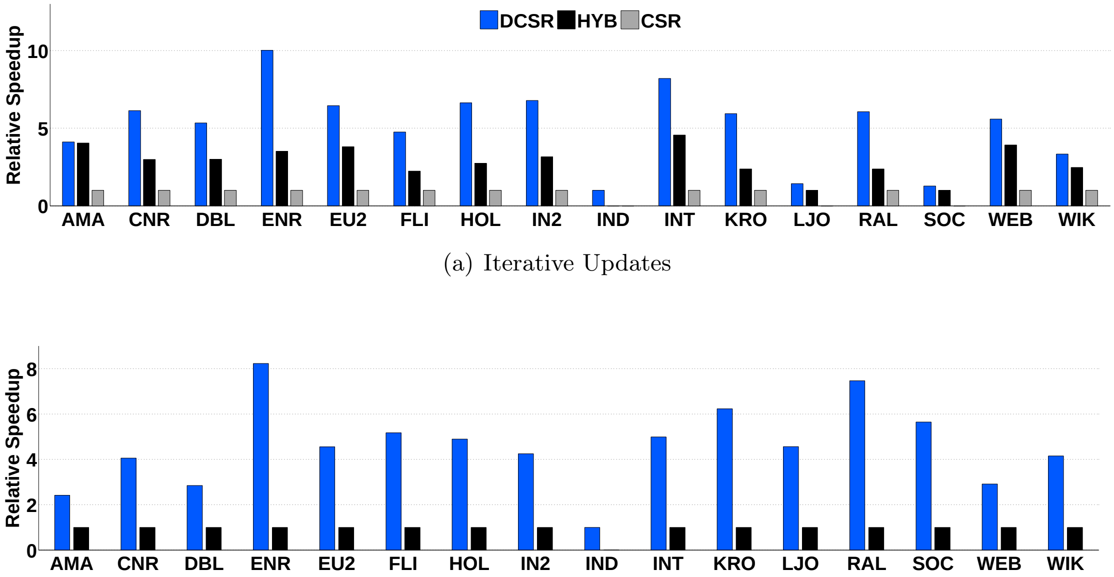
Updating a DCSR matrix requires finding the last occupied (current) segment within a row. If that segment is not full, the new entry is added into it and the row size is increased. When the current segment for a row fills up, a new segment is allocated dynamically. Since atomic operations are required only for the allocation of new segments, and not for each individual element, synchronization overhead is kept low. By allowing for dynamically sized slack space within a row, we dramatically reduce the number of atomic operations that are required to allocate new entries. In this way, DCSR was designed to be updated in an efficient parallel manner.

The number of segments, initial row width, and α value can be tuned for the problem to give a reasonable limit on updates. In our tests, we used four segments and an α value of μ (average row size of the matrix). When a row nears its limit, a defragmentation is required in order to reduce that row to a single segment.

Figure 5.3 provides the results of our iterative and streaming matrix update tests. We do not compare to CSR in the latter case, since it is not possible to dynamically add entries without rebuilding the matrix. This operation only loads the matrix and does not perform any insertion checks. DCSR saw an average speed-up of 4.8× over HYB with streaming updates. In the case of IND, only DCSR was able to perform the operation within memory capacity.

We also executed an iterative update test to compare the ability of the formats to perform a combination of dynamic updates and SpMV operations. This test is analogous to what would be done in a graph application (such as CFA) where the graph is updated at periodic intervals. In the iterative updates tests, a series of iterations are performed consisting of a matrix addition operation ($A = A + B$) followed by several SpMV operations $Ax = y$. Part (a) of Figure 5.3 provides the results for our iterative updates. Within each iteration, the matrix is updated with an additional 0.2% random nonzeros followed by 5 SpMV operations, which is repeated 50 times. This process yields a total increase of 10% to the number of nonzeros. We compare the DCSR and HYB results to a normalized CSR baseline. In the CSR case a new matrix must be created to update the original matrix, which causes a significant amount of overhead (in terms of computation and memory). In the cases of LJO and SOC, CSR was not able to complete within memory capacity, so we normalized against HYB.

DCSR shows significant improvement over HYB on streaming updates in all test cases (in some by as much as 8×). DCSR also outperforms HYB in all test cases on iterative updates, and in some cases by as much as 2.5×. The Amazon-2008 matrix has a low standard deviation, and the majority of its entries fit nicely into the ELL portion, which



(a) Iterative Updates

(b) Streaming Updates

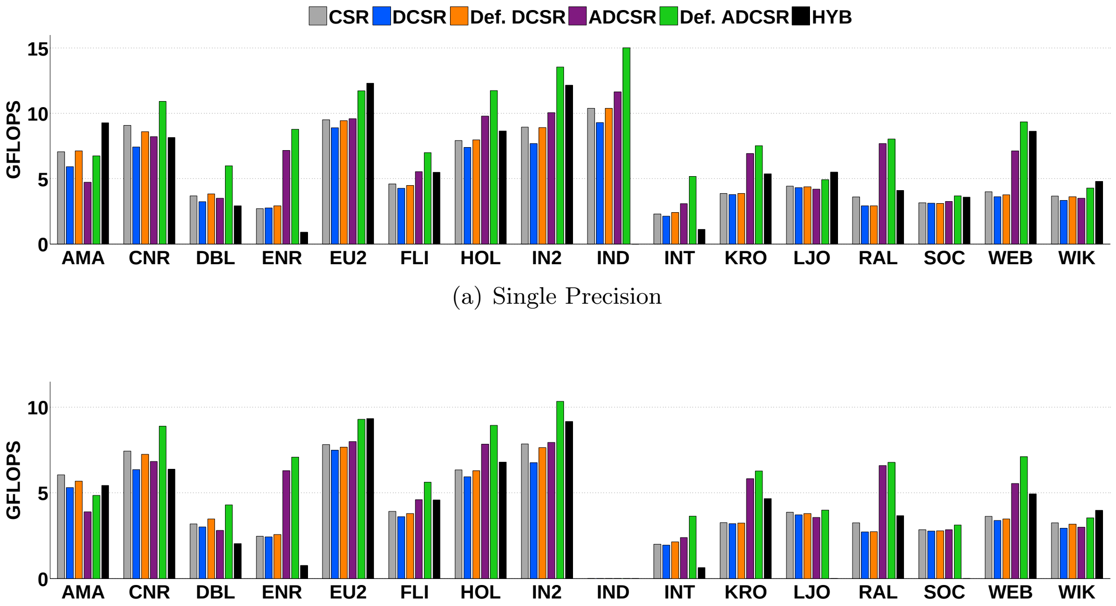
Figure 5.3: Relative speed comparisons. Top: Relative speed-up of DCSR compared to HYB for iterative updates with SpMV operations. The speed-up is compared to a normalized CSR baseline. Bottom: Relative speed-up of DCSR compared to HYB for matrix updates.

greatly speeds up SpMV operations. However, even in this case DCSR slightly outperforms HYB on iterative updates due to having lower overhead for defragmentation. In all other cases, DCSR exhibits noticeable performance improvements over HYB and CSR.

5.3.2 SpMV Results

In the SpMV tests, we take the same set of matrices and perform SpMV operations with randomly generated dense vectors. We performed each SpMV operation 100× times and averaged the results. Figure 5.4 provides the results for these SpMV tests run using both single- and double-precision floating-point arithmetic. We implemented the adaptive binning optimization (ACSR) outlined in [5], which we labeled ADCSR. This optimization requires relatively little overhead and provides noticeable speed improvements by using specialized kernels on bins of rows with similar row sizes. In these tests, we compare across several variants of our format, including DCSR, defragmented DCSR, ADCSR, and defragmented ADCSR, in addition to standard implementations of HYB and CSR.

The fragmented DCSR times are 8% slower than the defragmented DCSR times on average. When the DCSR format is defragmented, it sees SpMV times competitive with



(a) Single Precision

(b) Double Precision

Figure 5.4: FLOP ratings of SpMV operations for CSR, DCSR, and HYB.

those of CSR (1% slower on average). With the adaptive binning optimization applied, we see that ADCSR outperforms HYB in many cases. ADCSR performs 9% better on average than HYB across our benchmarks.

5.3.3 Postprocessing Overhead

Postprocessing overhead is a concern when dealing with dynamic matrix updates. Dynamic segmentation allows for DCSR to be updated with new entries without requiring the entries to be defragmented. SpMV operations can be performed on the DCSR format regardless of the number and order of segments, in contrast to HYB matrices where a sort is required anytime an entry is added that overflows into the COO portion. The SpMV operation for HYB matrices assumes the COO entries are sorted by row (without this property, the COO SpMV would be dramatically slower). Table 5.4 provides postprocessing times for HYB and DCSR formats relative to a single SpMV operation. In the case of IND, HYB was unable to sort and update due to insufficient memory (overhead represented as ∞).

The defragmentation operation gives us an opportunity to internally order rows by row-size at no additional cost. Our defragmentation algorithm is similar to the row sorting

Table 5.4: Overhead of DCSR defragmentation and HYB sorting is measured as the ratio of one operation against a single CSR SpMV. Update time is measured as the ratio of 1000 updates to a single CSR SpMV.

Matrix	DCSR defrag	HYB sort	DCSR update	HYB update
AMA	3.9	2.12	2.02	4.89
CNR	5.13	6.75	3.77	15.26
DBL	5.69	4.66	3.6	10.23
ENR	5.49	8.0	2.21	18.2
EU2	2.32	4.28	2.65	12.05
FLI	1.58	4.22	1.94	10.01
HOL	1.54	5.57	2.55	12.45
IN2	2.58	5.85	3.14	13.34
IND	2.15	∞	3.36	∞
INT	6.74	6.19	1.76	8.78
KRO	1.02	3.43	1.82	11.3
LJO	1.45	3.02	1.34	6.1
RAL	0.72	2.04	1.82	13.61
SOC	1.05	3.74	1.02	5.74
WEB	2.65	1.93	2.54	7.39
WIK	1.39	2.54	1.32	5.49

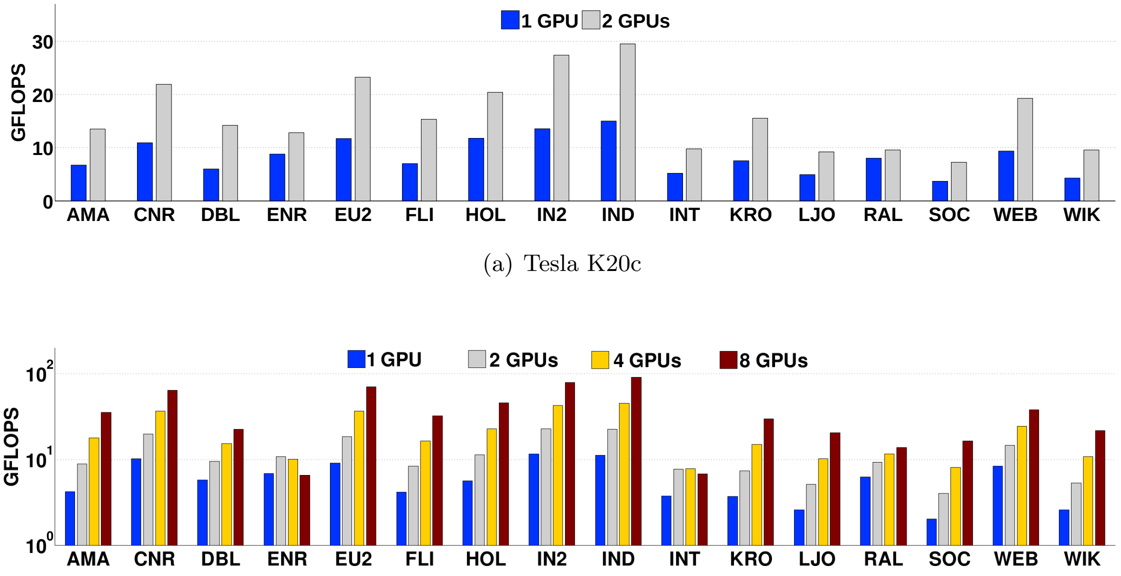
technique illustrated in [44], although we use a global sorting scope as opposed to a localized one. Because we explicitly manage segments within the columns and values arrays by both starting and ending index, the internal order of segments may be changed arbitrarily, and this permutation remains invisible from the outside. To accomplish this optimization, we permute row sizes according to the permuted row indices (which have already been binned and sorted by row size). The permuted row sizes can then be used to create new offsets for the monolithic segments produced by defragmentation. The column and value data can be internally reordered by row size at no additional cost. This internal reordering provides a noticeable SpMV performance improvement of 12%. This improvement is from an increased cache-hit rate via better correlation between bin-specific kernels and the memory they access.

The DCSR defragmentation incurs a lower overhead than HYB sort because entries can be shuffled to their new index without a sort operation. A DCSR defragmentation step is 2× faster on average than the HYB sorting step. More importantly, this process is required infrequently, whereas HYB sorting must be performed at every insertion, which means that DCSR requires significantly lower total postprocessing overhead.

5.3.4 Multi-GPU Implementation

DCSR can be effectively mapped to multiple GPUs. The matrix can be partitioned across n devices by dividing rows between them (modulo n) after sorting by row size, which provides a relatively even distribution of nonzeros between the devices. Figure 5.5 provides scaling results for DCSR across two Tesla K20c GPUs and up to eight Tesla M2090 GPUs. We see an average speed-up of 1.93× for the single precision and 1.97× across the set of test matrices. The RAL matrix sees a smaller performance gain due to our distribution strategy of dividing up the rows. The added parallelism is split across rows but, in this case, the matrix has few rows and many columns. We see nearly linear scaling for most test cases.

For the matrices INT and ENR, we see reduced scaling due to small matrix sizes. In these cases, the kernel launch times account for a significant portion of the total time due to a relatively small workload. The total compute time can be approximately represented as $c + \frac{x}{n}$, where c is the kernel launch overhead and the workload x is divided among n devices (assuming x can be fully parallelized). As the number of devices increases, the work per device decreases while the kernel launch time remains constant. In our tests, we perform 100× iterations of each kernel, which leads to poor scaling performance on small matrices.



(a) Tesla K20c

(b) Tesla M2090

Figure 5.5: Scaling results for SpMV with 1 and 2 K20 GPUs (upper) and 1, 2, 4, and 8 M2090 GPUs (lower).

We performed additional tests where we move the iterations into the kernel itself and call the kernel once, eliminating the additional kernel launch times. In this case, we see scaling for the INT matrix of 1.94×, 3.55×, and 6.03× and for the ENR matrix we see scaling of 1.80×, 2.70×, and 3.76× for 2, 4, and 8 GPUs, respectively. These results indicate that the poor performance was primarily due to the low amount of work done relative to the kernel launch overhead.

CHAPTER 6

SPARSE MATRIX-MATRIX MULTIPLICATION (SPMM)

6.1 Algorithm

It is a difficult task to efficiently compute $C = AB$ for sparse matrices in parallel. The sequential sparse matrix-matrix multiplication algorithm is not suitable for fine-grained parallelization. Sequential algorithms are efficient, but they rely on a large amount of (per thread) temporary storage. Specifically, to compute the sparse product $C = AB$, the sequential methods use $O(N)$ additional storage, where N is the number of columns in C. The parallel approach to sparse matrix-matrix multiplication is formulated in terms of highly scalable parallel primitives with no such limitations. As a result, a straightforward parallelization of the sequential scheme requires $O(n)$ storage per thread, which is not possible when using tens of thousands of independent threads of execution. Although it is possible to construct variations of the sequential method with lower per-thread storage requirements, any method that operates on the granularity of matrix rows (i.e., distributing matrix rows over threads), requires a nontrivial amount of per-thread state and suffers load imbalances for certain input [11].

The standard algorithm for parallel SpMM that exposes fine-grained parallelism is:

1. Expansion of $A * B$ into an intermediate coordinate format T.

2. Sorting of T by row and column indices to form $\hat{T}$.

3. Compression of $\hat{T}$ by summing duplicate values for each matrix entry.

Example 1: [h] T and $\hat{T}$ are given for $C = AB$, where

$$A = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 2 & 0 \\ 0 & 7 & 9 \end{bmatrix}, B = \begin{bmatrix} 4 & 3 & 7 \\ 0 & 5 & 0 \\ 2 & 0 & 8 \end{bmatrix}, C = \begin{bmatrix} 10 & 3 & 31 \\ 8 & 16 & 14 \\ 18 & 35 & 72 \end{bmatrix}$$

$$T = \begin{bmatrix} 0, & 0, & 4.0 \\ 0, & 1, & 3.0 \\ 0, & 2, & 7.0 \\ 0, & 0, & 6.0 \\ 0, & 2, & 24.0 \\ 1, & 0, & 8.0 \\ 1, & 1, & 6.0 \\ 1, & 2, & 14.0 \\ 1, & 1, & 10.0 \\ 2, & 1, & 35.0 \\ 2, & 0, & 18.0 \\ 2, & 2, & 72.0 \end{bmatrix} \quad \hat{T} = \begin{bmatrix} 0, & 0, & 4.0 \\ 0, & 0, & 6.0 \\ 0, & 1, & 3.0 \\ 0, & 2, & 7.0 \\ 0, & 2, & 24.0 \\ 1, & 0, & 8.0 \\ 1, & 1, & 6.0 \\ 1, & 1, & 10.0 \\ 1, & 2, & 14.0 \\ 2, & 0, & 18.0 \\ 2, & 1, & 35.0 \\ 2, & 2, & 72.0 \end{bmatrix}$$

All three stages of the algorithm expose fine-grained parallelism of which the GPU can take advantage of. The algorithm can be formulated in terms of efficient data-parallel computations — gather, scatter, scan, sort, etc. Like the sequential algorithm, this formulation is work efficient. It computes the exact number of partial products required for each nonzero without performing any additional operations with zero entries. It has the same computational complexity as the sequential method $O(nnz(T))$. The complexity is proportional to the size of the intermediate format T, and the work required at each stage is linear with respect to T. This process results in a relatively even load balancing across the GPU regardless of the sparsity patterns of the input matrices.

A limitation of this method is that the memory required to store the intermediate format is potentially large. If A and B are both square, $n \times n$ matrices with exactly K entries per row, then $O(nK^2)$ bytes of memory are needed to store T. Since the input matrices are generally large themselves ($O(nK)$ bytes), it is not always possible to store a K-times larger intermediate result in memory. In the limit, if A and B are dense matrices (stored in sparse format), then $O(n^3)$ storage is required. In such a case, the matrix-matrix multiplication $C = AB$ can be decomposed into several smaller operations that are computed in a workspace of bounded size. The resulting slices are then concatenated together to produce the final result. This technique introduces some overhead, but in practice it is relatively small as the workspace can be sized appropriately to saturate the device.

Our implementation of SpMM follows the same principles as the general algorithm, but we assign specialized kernels to process rows grouped by size. This algorithm allows for a more efficient use of shared memory when performing the sort and reduction operations. DCSR allows for asynchronous dynamic memory allocations when storing the rows products into C. This property of DCSR allows computation of the rows to be handled asynchronously. In the standard algorithm, the result of each previous row is required to know the offset when writing the final result into C. We precompute the number of partial products per row i following:

$$\sum_{k=1}^{ARS_i} BRS_j$$

where ARS_i is the number of entries in row i of matrix A, and j is the column index of element $a_{i,j}$. We then assign specific kernels, based on this row size, to process rows of length 1-32, 33-64, 65-128, 129-256, 257-512, 513-1024, 1025-2048, and 2049+.

The kernels process a row by computing the partial products, sorting them by column index, and reducing them before storing them in the resulting C matrix. Since this is done on a per row basis, the row is implicit and we need only store the column indices and values for the sorting and reduction phases. For all kernels except the 2049+ kernel, the operations are computed within shared memory on the GPU, which provides a significant performance improvement over global memory. For the 2049+ kernel, we use dynamic parallelism to assign a compute kernel to each row, which performs these operations using global memory.

SpMM was implemented using DCSR and its efficiency tested using algebraic multigrid. This improved method is compared to a similar version that computes SpMM using CSR and COO matrices. AMG can be formulated in terms of SpMM, SpMV, and primitive parallel operations. Algorithm 8 illustrates the structure of the AMG preconditioner setup phase of

Algorithm 8: AMG Setup

```
Input: $A$, $B$
Output: $A_0$,...,$A_M$, $P_0$,...,$P_M$
$A_0 \leftarrow A$, $B_0 \leftarrow B$;
for $k = 0, \ldots, M$ do
    $C_k \leftarrow$ strength($A_k$);
    $Agg_k \leftarrow$ aggregate($C_k$);
    $T_k, B_{k+1} \leftarrow$ tentative($Agg_k, B_k$);
    $P_k \leftarrow$ prolongator($A_k, T_k$);
    $R_k \leftarrow P_k^T$;
    $A_{k+1} \leftarrow (R_k A_k P_k)$;
```

AMG given a sparse matrix A and a set of vectors B. In our tests, we used a constant vector, which is a common default. The $(R_k A_k P_k)$ operation computes the Galerkin product of the three matrices using SpMM by first computing $A * P = AP$ followed by $R * AP = RAP$.

6.2 Experimental Results

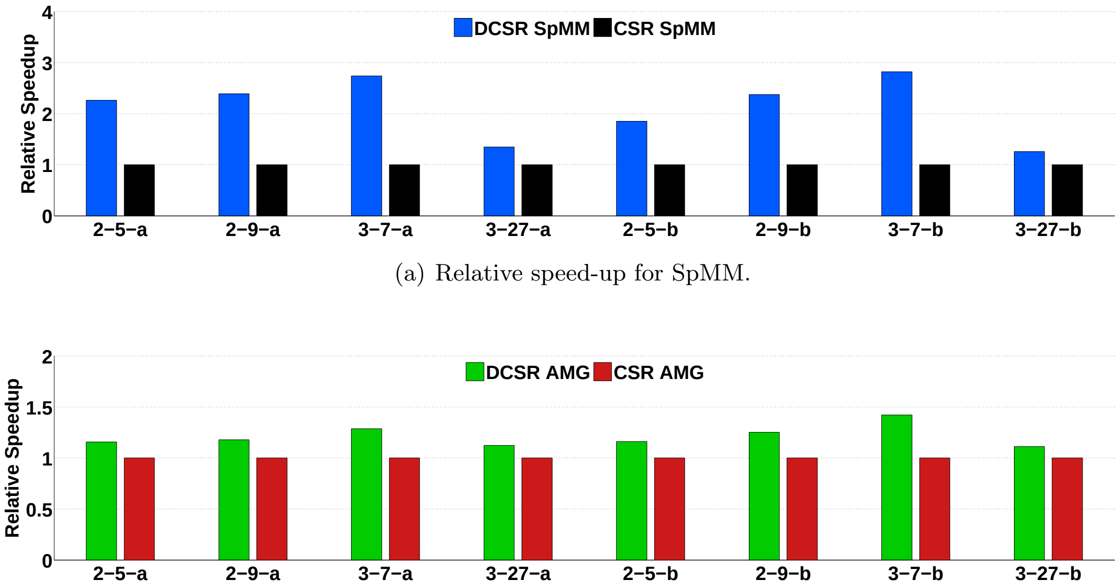
We compare the results for AMG on 2D and 3D Poisson problems with Dirichlet boundary conditions. It is known that AMG performs well as a preconditioner on such problems, which allows us to focus on the merits of the SpMM method rather than on whether AMG is suited for the problem. Table 6.1 lists the set of matrices used in our tests as well as the number of unknowns and nonzeros. These tests were all computed with double precision.

Figure 6.1 illustrates the results of our AMG tests with both the individual SpMM times and the overall AMG preconditioner times. Our method outperforms the baseline method by upwards of 3× in some cases. The Galerkin product represents 30% – 50% of total time required by the setup phase of the preconditioner. Results shown in [11] indicate that the Galerkin product occupies 50% – 60% of the run time on similar matrices using a Nvidia Tesla C2050 GPU, which seems to indicate that the underlying architecture plays a role in the relative processing times across stages. In the case of matrix 3-7-a, the Galerkin product occupies roughly half of the setup time, and our SpMM method is nearly 3× faster in that case, resulting in a speed-up of 40%. There is no guarantee what the resulting fill will be in the C matrix, but in practice the resulting fill is relatively sparse for multiplication with Poisson matrices.

By taking advantage of asynchronous updates enabled by DCSR, we are able to employ

Table 6.1: List of matrices used for AMG tests.

Matrix	Abbr.	Unknowns	Nonzeros
2D Poisson 5pt	2-5-a	262144	1310720
2D Poisson 9pt	2-9-a	262144	2359296
3D Poisson 7pt	3-7-a	262144	1810432
3D Poisson 27pt	3-27-a	262144	6859000
2D Poisson 5pt	2-5-b	1048576	5238784
2D Poisson 9pt	2-9-b	1048576	9424900
3D Poisson 7pt	3-7-b	2097152	14581760
3D Poisson 27pt	3-27-b	2097152	55742968



(a) Relative speed-up for SpMM.

(b) Relative speed-up for AMG.

Figure 6.1: Relative speed-up for SpMM and AMG using DCSR and CSR.

specialized kernels based on row lengths. These row length optimized kernels perform the sort and reduction operations within shared memory, which is notably faster than performing these operations within global memory. The efficient use of shared memory leads to significant performance gains for the overall SpMM operation. The Galerkin product is by far the largest single component of the setup phase, so improvements in this area will lead to the greatest gains.

Our tests indicate that this new SpMM method is faster for row sizes that can fit within shared memory. A 2× to 3× speed-up was observed on matrix multiplication operations that do not exceed 2048 partial products per row. When the number of the partial products in a row exceeds the shared memory limit, there is no way to efficiently sort and reduce the elements without tiling and excessive memory shuffling. In that case, it is more efficient to perform the standard SpMM computation for that row.

CHAPTER 7

FUTURE WORK

This work can be expanded upon in a number of ways. In addition, there are number of interesting applications that could be explored in relation to this work.

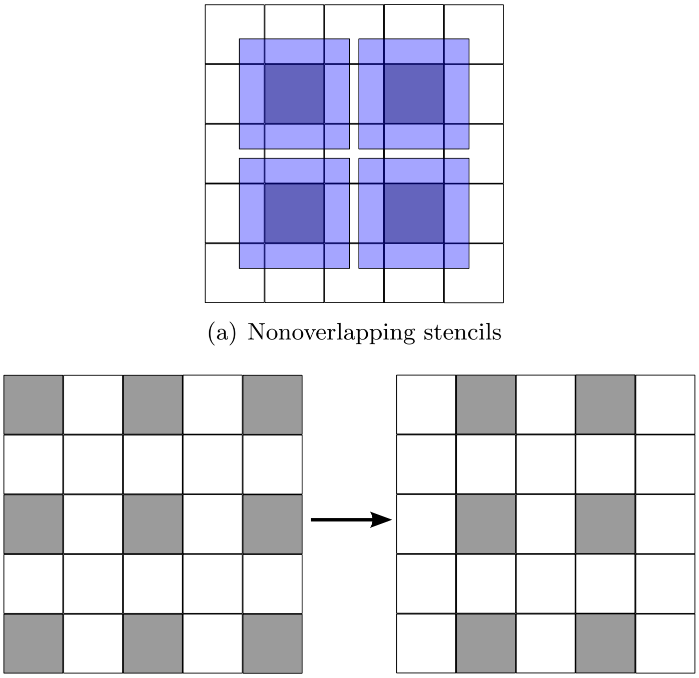
7.1 Nonoverlapping Stencil Parallelization

The stencil parallelization presented in Chapter 3 could be expanded upon. The method currently being employed uses temporary workspace arrays for each patch. The results from each patch are scattered to their respective locations in memory. Afterward, the results are scattered to the final solution space. This requires additional memory space based on the number of mesh division patches. Although the final reduction step between the patch memory spaces is relatively quick, it may be possible to avoid this step with a careful parallelization scheme.

An alternate method of parallelization could be implemented which relies on parallelizing according to nonoverlapping stencils. Figure 7.1 illustrates this process. This method could be combined with the CUDA unified memory abstraction which allows the CPU and GPU to access a shared pool of managed memory as opposed to accessing two separate memory spaces. This allows multiple GPUs on the same node to asynchronously update the CPU host memory without the need for specific GPU to CPU write calls. This would remove the need for additional patch memory spaces and eliminate the final reduction step.

7.2 Expanded Tuning Parameters

The tuning methodology used in this work is relatively simple and could be expanded upon. A number of other search space parameters could be explored such as kernel unrolling and code variants. Furthermore, the search space employed is exhaustively searched for the best configuration, and better heuristics and autotuning approaches could be employed to refine the search space. Methods presented in generalized frameworks like OpenTuner [3]



(a) Nonoverlapping stencils

(b) Stencil parallelization by patches

Figure 7.1: Illustration of patch parallelization using nonoverlapping stencils (grey patches are currently active).

and Orio [35] could be adapted.

7.3 Applications for Dynamic Compressed Sparse Row

There are a number of applications that could benefit from dynamic updates. Performing static control flow analysis for programs was a primary motivation for this work. A general approach to static program analysis of higher-order languages has been developed [77, 49]. These algorithms use an approximate interpretation of their target code to yield an upper bound on the propagation of data and control through a program across all possible actual executions. A CFA involves a series of increasing operations on a graph (extending it with nodes and edges), terminating when a fixed point is reached (a steady state where the analysis is self-consistent).

Recent work has shown how to implement this kind of static analysis as linear-algebraic operations on the sparse-matrix representation of a function [33, 68]. Other recent work shows how to implement an inclusion-based points-to analysis of C on the GPU by applying a set of semantic rules to the adjacency matrix of a sparse-graph [48]. These algorithms may be likened to finding the transitive closure of a graph encoded as an adjacency matrix. The matrix is repeatedly extended with new entries derived from SpMV until a fixed point is reached (no more edges need to be accumulated). These approaches to static analysis on the GPU are distinct; however, both require high performance sparse-matrix operations and dynamic insertion of new entries.

Graph problems are a primary problem type that may benefit from dynamic updates. Computing the transitive closure of a graph could be formulated in a way that dynamically updates a sparse matrix. This operation could also be performed through SpMM operations, which have been demonstrated to benefit from dynamic updates.

CHAPTER 8

CONCLUSIONS

This dissertation targeted serveral goals relating to irregularities in control flow and memory access. These goals are fufilled by the analysis and methods presented in this dissertation. The primary applications analyzed dealt with unstructured meshes and sparse matrices, both of which have high degrees of irregularity due to indirection within the data structures.

This dissertation presents a method for improved data reuse through associative reordering of operations for stencil compuations over unstructured meshes. Two general strategies are presented for evaluating stencil computations over unstructured meshes, per-point and per-element. In addition, a scalable overlapped tiling method is presented, which allows for concurrent execution of stencils.

Increased data-reuse and data locality have a significant impact on the performance of stencil computations over unstructured meshes with high levels of concurrency. On the GPU, the per-element method exhibits between 2× and 6× performance improvement over the per-point counterpart. The per-element method demonstrates perfect linear scaling as the number of computing cores increases. The overlapped tiling method allows for nearly perfect linear scaling with minimal synchronization overhead. The per-element method adds some memory overhead to the process, but significantly improves overall performance.

This dissertation presents results that demonstrate the impact that vector widths have on performance for postprocessing of 3D discontinuous Galerkin solutions. This application represents a dense numerical computation that benefits greatly from vectorization. Loops are restructured based on a tunable vector width parameter. An exhaustive search autotuning method is applied to test combinations of the vector width, regions per block (with block size being determined by the regions per block and vector width), and block count. By combining loop restructuring with autotuning, performance gains of up to

50% over a baseline using full vectorization were seen for postprocessing. Some level of vectorization almost always yields improved performance over nonvectorized operations on SIMD architectures. However, our results have demonstrated that full vectorization generally does not provide optimal performance for dense numerical computations. The optimal vector width is generally nonobvious and is best found with an automated method since the search space is large.

Vectorization tends to be more beneficial for dense operations. Partial vectorization can yield better cache/shared memory utilization while still maintaining a high number of coalesced memory accesses and low loop overhead. This vectorization can significantly boost performance for dense numerical computations. Autotuning has proven to be an effective method for finding the optimal balance point between nonvectorized and fully vectorized computations. This technique of adjusting vector widths is broadly applicable to many domains, and it could be applied to other types of numerical computations.

Lastly, this dissertation presents a fast, flexible, and memory-efficient strategy for dynamic sparse-matrix allocation. A new sparse matrix format (DCSR) is illustrated, which provides a robust method for allocating streaming updates while maintaining fast SpMV times on a par with that of CSR. The format gracefully degrades in performance upon dynamic extension, but does not require a sort to be performed after inserting new entries (as opposed to COO-based formats like HYB).

Without defragmentation, SpMV times are only marginally slower than that of a fully constructed CSR matrix, and after defragmentation they are roughly equal. With adaptive binning applied, DCSR gives faster overall SpMV times as compared to the HYB format. DCSR is significantly more efficient in terms of memory use as well. ELL must allocate enough room in every row for the longest row in a matrix. HYB is a vast improvement, by allowing long rows to overflow into its COO portion. However, DCSR exhibited lower memory consumption on every benchmark when set to allow 2 segments per row, and still used 20% less memory on average when allowing 4 segments per row.

A key advantage of DCSR in its design is compatibility with CSR-scalar, CSR-vector, and other CSR algorithms. Only minor modifications are required to account for a difference in the format of the row offsets array. CSR specific optimizations, such as adaptive binning, can be easily applied to DCSR. Other optimizations, such as tiling and blocking, could also be used. This compatibility also means that minimal overhead is required to convert to and from CSR. Numerous sparse-matrix formats have been developed that are specifically tailored to GPU architectures. These formats offer improved performance, but require converting from whatever previous format was being used. As CSR is the most commonly used sparse-matrix format, and large amounts of software already incorporate it into their code bases, it is often not worth the conversion cost to introduce another format. DCSR reduces this barrier to use with a low cost of conversion.

This dissertation also presents an improved algorithm for parallel sparse matrix-matrix multiplication. The standard method for parallel SpMM requires conversions to and from the COO format as well as a large workspace in global memory. By utilizing the dynamic updates with DCSR, the sort and reduction operations can be computed in shared memory which is significantly faster than global memory. This also avoids the need for costly format conversions of the matrix.

REFERENCES

[1] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. 2011. A hybridization methodology for high-performance linear algebra software for GPUs. In *GPU Computing Gems, Jade Edition*, Wen-mei W. Hwu (Ed.). Vol. 2. Elsevier Inc., 473–484.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques.* Edmonton, Canada.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (Oct. 2009), 56–67.

[5] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14).* IEEE Press, Piscataway, NJ, USA, 781–792.

[6] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. 2014. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14).* ACM, New York, NY, USA, 273–282.

[7] ATI. 2011. *AMD Accelerated Parallel Processing OpenGL Programming Guide.*

[8] J. Barnes and P. Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature* 324 (Dec. 1986), 446–449.

[9] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-directed transformation for higher-order stencils. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS '15).* IEEE Computer Society, Washington, DC, USA, 313–323.

[10] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: Leveraging warp specialization for high performance on GPUs. *SIGPLAN Not.* 49, 8 (Feb. 2014), 119–130.

[11] Nathan Bell, Steven Dalton, and Luke N. Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* (2012).

[12] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

[13] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, New York, NY, USA, 1–11.

[14] Nathan Bell and Michael Garland. 2012. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. (2012). Version 0.3.0.

[15] Jon Louis Bentley and Jerome H. Friedman. 1979. Data structures for range searching. *ACM Comput. Surv.* 11, 4 (Dec. 1979), 397–409.

[16] James Bergstra, Nicolas Pinto, and David Cox. 2012. Machine learning for predictive auto-tuning with boosted regression trees. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 1–9.

[17] A. Braunstein, M. Mézard, and R. Zecchina. 2005. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms* 27, 2 (2005), 201–226.

[18] M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 141–151.

[19] Li-Wen Chang, John A. Stratton, Hee-Seok Kim, and Wen-Mei W. Hwu. 2012. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 27, 11 pages.

[20] L. Paul Chew. 1993. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry (SCG '93)*. ACM, New York, NY, USA, 274–280.

[21] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *SIGPLAN Not.* 45, 5 (Jan. 2010), 115–126.

[22] B. Cockburn, M. Luskin, C.W. Shu, and E. Suli. 1999. Post-processing of Galerkin methods for hyperbolic problems. In *Proceedings of the International Symposium on Discontinuous Galerkin Methods*. Springer, 291–300.

[23] B. Cockburn, M. Luskin, C.W. Shu, and E. Suli. 2003. Enhanced accuracy by post-processing for finite element methods for hyperbolic equations. *Math. Comp.* 72 (2003), 577–606.

[24] AMD Corp. 2011. *AMD Accelerated Parallel Processing Math Libraries*.

[25] M. Daga, A. M. Aji, and W. C. Feng. 2011. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In *2011 Symposium on Application Accelerators in High-Performance Computing*. IEEE Press, 141–149.

[26] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 4, 12 pages.

[27] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages.

[28] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. 2008. Fast scan algorithms on graphics processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing (ICS '08)*. ACM, New York, NY, USA, 205–213.

[29] Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. 2005. Probabilistic source-level optimisation of embedded programs. *SIGPLAN Not.* 40, 7 (June 2005), 78–86.

[30] Michael Garland. 2008. Sparse matrix computations on manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08)*. ACM, New York, NY, USA, 2–6.

[31] Michael Garland and David B. Kirk. 2010. Understanding throughput-oriented architectures. *Commun. ACM* 53, 11 (Nov. 2010), 58–66.

[32] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2007. High-performance graph algorithms from parallel sparse matrices. In *Applied Parallel Computing. State of the Art in Scientific Computing*, Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski (Eds.). Lecture Notes in Computer Science, Vol. 4699. Springer Berlin Heidelberg, 260–269.

[33] Thomas Gilray, Jim King, and Matthew Might. 2014. Partitioning 0-CFA for the GPU. *Workshop on Functional and Constraint Logic Programming* (September 2014).

[34] Joseph L. Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 769–780.

[35] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*. IEEE Press, Rome, Italy, 1–11.

[36] K. Hildrum and P.S. Yu. 2005. Focused community discovery. In *Fifth IEEE International Conference on Data Mining*. IEEE Press, Newport Beach, CA, USA.

[37] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 311–320.

[38] Frank Hutter, Youssef Hamadi, HolgerH. Hoos, and Kevin Leyton-Brown. 2006. Performance prediction and automated tuning of randomized and parametric algorithms. In *Principles and Practice of Constraint Programming - CP 2006*, Frédéric Benhamou (Ed.). Lecture Notes in Computer Science, Vol. 4204. Springer Berlin Heidelberg, 213–228.

[39] Eun-jin Im and Katherine Yelick. 2000. Optimization of sparse matrix kernels for data mining. In *First SIAM Conference on Data Mining*. SIAM, Chicago, IL, USA, 1–16.

[40] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[41] Khronos Group. 2011. *The OpenCL specification*.

[42] J. King and R. M. Kirby. 2013. A scalable, efficient scheme for evaluation of stencil computations over unstructured meshes. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, Piscataway, NJ, USA, 1–12.

[43] David B. Kirk and Wen-mei W. Hwu. 2010. *Programming Massively Parallel Processors: A Hands-On Approach* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[44] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2013. A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR* abs/1307.6209 (2013).

[45] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*. ACM, New York, NY, USA, 235–244.

[46] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 273–282.

[47] Tareq Malas, Aron J. Ahmadia, Jed Brown, John A. Gunnels, and David E. Keyes. 2013. Optimizing the performance of streaming numerical kernels on the IBM Blue Gene/P PowerPC 450 processor. *International Journal of High Performance Computing Applications* 27, 2 (1 May 2013), 193–209.

[48] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 107–116.

[49] Jan Midtgaard. 2012. Control-flow analysis of functional programs. *Comput. Surveys* 44, 3 (June 2012), 10:1–10:33.

[50] H. Mirzaee, Liangyue Ji, J. K. Ryan, and R. M. Kirby. 2011. Smoothness-increasing accuracy-conserving (SIAC) post-processing for discontinuous Galerkin solutions over structured triangular meshes. *SIAM Journal of Numerical Analysis* 49 (2011), 1899–1920.

[51] H. Mirzaee, J. King, J. Ryan, and R. Kirby. 2013. Smoothness-increasing accuracy-conserving filters for discontinuous Galerkin solutions over unstructured triangular meshes. *SIAM Journal on Scientific Computing* 35, 1 (2013), A212–A230.

[52] H. Mirzaee, J. K. Ryan, and R. M. Kirby. 2011. Efficient implementation of smoothness-increasing accuracy-conserving (SIAC) filters for discontinuous Galerkin solutions. *Journal of Scientific Computing* (2011).

[53] Hanieh Mirzaee, Jennifer K. Ryan, and Robert M. Kirby. 2013. Smoothness-increasing accuracy-conserving (SIAC) filters for discontinuous Galerkin solutions: Application to structured tetrahedral meshes. *Journal of Scientific Computing* 58, 3 (2013), 690–704.

[54] Jayadev Misra. 1986. Distributed discrete-event simulation. *ACM Comput. Surv.* 18, 1 (March 1986), 39–65.

[55] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers*, YaleN. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Lecture Notes in Computer Science, Vol. 5952. Springer Berlin Heidelberg, 111–125.

[56] Antoine Monsifrot, François Bodin, and Rene Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA '02).* Springer-Verlag, London, UK, UK, 41–50.

[57] Anna Morajko, Tomas Margalef, and Emilio Luque. 2007. Design and implementation of a dynamic tuning environment. *J. Parallel and Distrib. Comput.* 67, 4 (2007), 474–490.

[58] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6).* ACM, New York, NY, USA, 96–107.

[59] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-driven versus topology-driven irregular computations on GPUs. In *IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS).* 463–474.

[60] NVIDIA 2010. *CUDA CUSPARSE Library.* NVIDIA.

[61] NVIDIA 2015. *CUDA C Best Practices Guide.* NVIDIA.

[62] NVIDIA. 2015. *CUDA C Programming Guide v7.5.* NVIDIA.

[63] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5 (May 2008), 879–899.

[64] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron Lefohn, and Timothy J. Purcell. 2007. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, Vol. 26. 80–113.

[65] David A. Padua. 2011. Autotuning for high performance computing. In *Proceedings of the Fourth Workshop on High Performance Computational Finance (WHPCF '11)*. ACM, New York, NY, USA, 19–20.

[66] James L. Peterson. 1977. Petri nets. *ACM Comput. Surv.* 9, 3 (Sept. 1977), 223–252.

[67] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 12–25.

[68] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. 2010. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the Symposium on the Principals of Programming Languages*. ACM, Piscataway, NJ, USA, 511–522.

[69] I. Reguly and M. Giles. 2012. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Innovative Parallel Computing (InPar)*. IEEE Press, Piscataway, NJ, USA, 1–12.

[70] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Parameterized tiled loops for free. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 405–414.

[71] Max Rietmann, Peter Messmer, Tarje Nissen-Meyer, Daniel Peter, Piero Basini, Dimitri Komatitsch, Olaf Schenk, Jeroen Tromp, Lapo Boschi, and Domenico Giardini. 2012. Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 38, 11 pages.

[72] J. K. Ryan and C.-W. Shu. 2003. On a one-sided post-processing technique for the discontinuous Galerkin methods. *Methods and Applications of Analysis* 10 (2003), 295–307.

[73] J. K. Ryan, C.-W. Shu, and H. L. Atkins. 2005. Extension of a post-processing technique for the discontinuous Galerkin method for hyperbolic equations with application to an aeroacoustic problem. *SIAM Journal on Scientific Computing* 26 (2005), 821–843.

[74] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[75] Hanan Samet. 2005. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[76] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. 2013. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Press, Piscataway, NJ, USA, 73–82.

[77] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. Carnegie-Mellon University, Pittsburgh, PA.

[78] F. Song, S. Tomov, and J. Dongarra. 2011. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 365–376.

[79] M. Steffen, S. Curtis, R. M. Kirby, and J. K. Ryan. 2008. Investigation of smoothness enhancing accuracy-conserving filters for improving streamline integration through discontinuous fields. *IEEE Transactions on Visualization and Computer Graphics* 14, 3 (2008), 680–692.

[80] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 77–90.

[81] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 65–76.

[82] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. 2015. TOP 500. (2015). http://www.top500.org.

[83] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 353–364.

[84] Ivan E. Sutherland and Gary W. Hodgman. 1974. Reentrant polygon clipping. *Commun. ACM* 17, 1 (1974), 32–42.

[85] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2005. *Introduction to Data Mining* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[86] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1–11.

[87] David Tarjan, Jiayuan Meng, and Kevin Skadron. 2009. Increasing memory miss tolerance for SIMD cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, New York, NY, USA, Article 22, 11 pages.

[88] Martin Tillmann, Thomas Karcher, Carsten Dachsbacher, and Walter F. Tichy. 2014. Application-independent autotuning for GPUs. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE) (Advances in Parallel Computing)*. IOS Press, Munich, Germany, 626–635.

[89] Oreste Villa, Daniel R. Johnson, Mike O'Connor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, Stephen W. Keckler, and William J. Dally. 2014. Scaling the power wall: A

path to exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 830–841.

[90] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 31, 11 pages.

[91] Richard Wilson Vuduc. 2003. *Automatic performance tuning of sparse matrix kernels.* Ph.D. Dissertation. University of California, Berkeley.

[92] David Walfisch, Jennifer K. Ryan, Robert M. Kirby, and Robert Haimes. 2009. One-sided smoothness-increasing accuracy-conserving filtering for enhanced streamline integration through discontinuous fields. *Journal of Scientific Computing* 38, 2 (2009), 164–184.

[93] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1–2 (2001), 3–35.

[94] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, New York, NY, USA, Article 38, 12 pages.

[95] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76.

[96] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet another SpMV framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 107–118.

[97] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. 2011. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *Proc. VLDB Endow.* 4, 4 (Jan. 2011), 231–242.

[98] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. 2003. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 63–76.

[99] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. *SIGPLAN Not.* 46, 3 (March 2011), 369–380.

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