Optimization Techniques for GPU Programming

PIETER HIJMA, Vrije Universiteit Amsterdam
STIJN HELDENS, ALESSIO SCLOCCO, and BEN VAN WERKHOVEN, Netherlands eScience Center
HENRI E. BAL, Vrije Universiteit Amsterdam

In the past decade, Graphics Processing Units have played an important role in the field of high-performance computing and they still advance new fields such as IoT, autonomous vehicles, and exascale computing. It is therefore important to understand how to extract performance from these processors, something that is not trivial. This survey discusses various optimization techniques found in 450 articles published in the last 14 years. We analyze the optimizations from different perspectives which shows that the various optimizations are highly interrelated, explaining the need for techniques such as auto-tuning.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **Software performance**; • **Computing methodologies** → **Graphics processors;**

Additional Key Words and Phrases: Survey, GPU, optimization, optimization techniques, performance bottleneck

ACM Reference format:
Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* 55, 11, Article 239 (March 2023), 81 pages.
https://doi.org/10.1145/3570638

1 INTRODUCTION

Graphics Processing Units (**GPUs**) have revolutionized the HPC landscape in the past decades [144] and are seen as an enabling factor in recent advances in **Artificial Intelligence** (**AI**) [199]. GPUs originated as processors for gaming and then were adapted to more general workloads as co-processors in HPC systems. Over the past decade, GPUs have started to again penetrate new markets such as IoT devices [260] and autonomous vehicles [228]. The first generation of exascale supercomputers is being deployed right now, most of which use GPUs as their main computing platform. However, in contrast to the pre-exascale era where NVIDIA dominated the GPU market, several of these systems contain GPUs from Intel [2] and AMD [1, 3] with different and relatively new programming models. It is therefore urgent that we understand the lessons

This project has received funding from the Dutch Research Council (NWO) in the framework of the NWA-ORC Call under file number NWA.1160.18.316 (CORTEX) and from the Netherlands eScience Center under file number 027.016.G06 (A methodology and ecosystem for many-core programming).
Authors' addresses: P. Hijma and H. E. Bal, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands; emails: {p.hijma, h.e.bal}@vu.nl; S. Heldens, A. Sclocco, and B. van Werkhoven, Netherlands eScience Center, Amsterdam, The Netherlands; emails: {s.heldens, a.sclocco, b.vanwerkhoven}@esciencecenter.nl.

0360-0300/2023/03-ART239 $15.00
https://doi.org/10.1145/3570638

learned in the past fourteen years of GPU optimization techniques now that the hardware, applications, and programming systems for GPU programming are rapidly diversifying.

With the introduction of the CUDA programming model in 2007, GPU programming became accessible and widespread quickly. The OpenCL standard was released in late 2008, closely following the CUDA programming model and enabling execution on a wider range of processors. Both CUDA and OpenCL remain the main programming languages for GPU applications, despite many higher-level languages and domain-specific languages that have been developed to simplify application development. GPU programming is often seen as a specialized field, in which *performance optimization* in particular is regarded as something that requires advanced knowledge of the intricacies of the hardware and finding the right balance in a large number of trade-offs. As such, many studies describe code transformations, programming techniques, or approaches that are collectively referred to in the literature as *optimizations* that aim at improving the performance of GPU applications.

In this survey, we provide the first comprehensive overview of such techniques and approaches proposed to improve GPU application performance. We focus on software techniques that can be applied by programmers to improve performance on existing hardware in the context of GPU programming languages, such as CUDA and OpenCL. Optimizations on the intermediate representation produced by GPU compilers or architectural techniques to improve performance are outside the scope of this work. In this article, we use CUDA terminology, but most optimizations are also applicable to OpenCL and non-NVIDIA hardware.

This survey aims at serving multiple audiences: First of all, GPU programmers can learn about the various techniques they might apply to improve performance; Second, researchers in programming languages and compilers can observe where GPU programming is challenging and where additional language support has or may further ease the process of performance optimization; Finally, researchers in computer architecture and hardware manufacturers can learn how the programmability of architectures has improved over time and might be enhanced further.

We first discuss studies related to this survey (Section 2), then discuss our method (Section 3) to process 450 articles from which we extracted optimizations and stored them in a database for analysis (Section 4). After we introduce GPU programming (Section 5), we describe various optimization techniques (Section 6) that we analyze from different viewpoints (Section 7), concluding that the various optimizations are highly interrelated, that many factors are dependent on each other, and that techniques such as auto-tuning are very helpful in striking a good balance for high utilization (Section 8).

2 RELATED WORK

In this section, we review earlier work that discusses optimization techniques for GPUs or, more generally, parallel architectures. Bacon et al. survey high-level program restructuring techniques [41] mainly for compilers, but they state that these techniques are also relevant for manual optimization. They define an optimization to be a shorthand for *optimizing transformation* and the overall goal of optimization is maximizing the use of computational resources while minimizing the number of operations, the use of memory bandwidth, and the size of total memory used. They acknowledge that optimization becomes increasingly more complex with more intricate architectures.

Kowarschik et al. supplement Bacon's loop-based optimizations with cache and data locality optimizations [190]. Their work focuses on CPUs but the optimizations they present are often important for GPU programming as well.

In 2008, Ryoo et al. published two articles reviewing optimizations explicitly for *GPUs* [318, 320]. These articles are relatively early compared to the first CUDA implementation (2007) and focus on

understanding which code is suitable to run on GPUs [318] and how to achieve high performance with these codes, stressing that a correct balance is often required to obtain high performance requiring auto-tuning [318, 320].

In 2012, Stratton et al. surveyed several GPU applications and kernels [337] and presented various *optimization patterns* and *performance issues*. They discuss for each pattern which performance issues they address. In the same year, Brodtkorb et al. discuss optimization strategies [56]. They base their guidelines on the most important architectural features of GPUs, namely memory latency with multithreading and the memory hierarchy. They advocate a profiler-driven methodology with three kernel versions: the original, one without the memory statements, and one with only the memory statements. The goal is to create a well-balanced kernel that can be auto-tuned when facing conflicting requirements to balance the kernels. These studies from 2008 and 2012 discuss general principles that are based on the authors' experiences in programming GPUs. Instead, our study is based on information on GPU optimizations from the articles we reviewed.

There are several related surveys that discuss GPU optimizations for a specific domain. Tran et al. discuss GPUs in the context of graph processing [357] and focus mainly on the importance of data layout and workload distribution. Al-Mouhamed et al. review optimizations and high-level compilers for structured grid computing [17]. They make a distinction between basic architectural optimizations, for example for memory bandwidth and locality, and domain-specific optimizations, for example targeting synchronization between iterations. Mittal et al. survey optimization techniques for Deep Learning on GPUs [261] and make a distinction between optimizations for *computer-related bottlenecks* and *memory-related bottlenecks*, presenting optimization schemes for both categories.

Although this survey focuses on optimizations manually applied by programmers, much work has been performed to improve *GPU architectures* to increase the performance. Khairy et al. performed a survey of architectural improvements of GPUs [181] and we consider this survey to be of interest to our readership as well as it provides context for the optimizations that we discuss here. In addition, it contains an excellent comparison between CPUs and GPUs.

3 METHODOLOGY

This section provides an overview of the approach for finding, selecting, and processing the articles for our study. To this end, we make use of the curated Scopus library by using the software package *litstudy* [145], which helps us to perform queries and organize the results.

Our method has several distinct phases designed to get a good coverage of many of the articles published on GPU optimizations. We used this extensive approach for several reasons: We noticed that many studies were referring to the same concepts using different terminology and this coverage allowed us to understand what names different authors were giving to sometimes similar or overlapping concepts. In addition, this approach allowed us to analyze which optimizations were more prevalent than others, see our data analysis in Section 4. Finally, we wanted to make sure that the generally underrepresented AMD GPUs and OpenCL code were taken into account as well.

Figure 1 shows the selection process that we describe in detail in Appendix B. In *phase 1* we perform a query on the Scopus database based on keywords in the context of GPU optimizations. We executed the query on 29 November 2019 and repeated it on 27 May 2021 to update the database with the newly published articles. The query resulted in 3,973 articles from which we first selected based on the title, venue, and keywords resulting in 1,120 articles. In *phase 2* we added abstracts and authors for the selection process and selected 532 articles and marked 202 as an *auxiliary* (i.e., articles that are not selected but contained other relevant information). From these auxiliary articles, we selected 10 resulting in 542 articles after phase 2. In *phase 3* we "scan" these articles and select articles if they contain GPU optimizations and satisfy several selection criteria (see

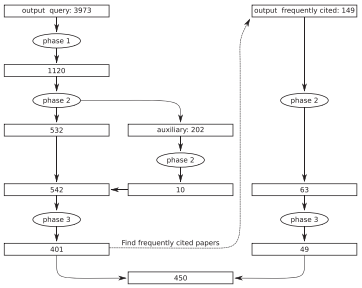


Fig. 1. Overview of three of the phases in the selection process.

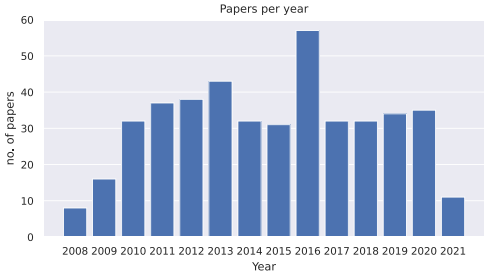


Fig. 2. No. of articles per year.

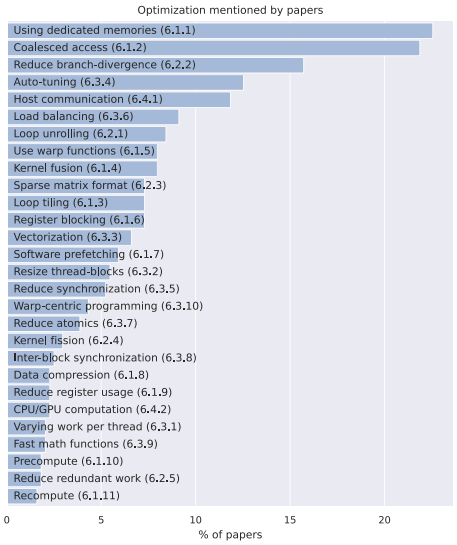


Fig. 3. No. of articles per optimization.

Appendix B). This resulted in 401 articles that we analyzed for articles that were cited often but were not in our selection. We deemed these 149 frequently cited articles important for our selection process as well and after phase 2 and 3, this resulted in 49 articles, bringing the set of articles that we analyzed for optimizations to 450.

In *phase 4* we extract information from these 450 articles, such as the applied optimizations, the used GPUs, and explicitly mentioned bottlenecks, and store this information in a database. In *phase 5* we analyze and describe the optimizations, for which Appendix A forms the reference material for readers who are interested in more details. Due to space limitations, a summary of this reference material is presented in Section 6. The performance bottlenecks are analyzed and brought into relation with the optimizations in Section 7.

4 DATA ANALYSIS

In this section, we analyze the set of articles as selected by our search query. Appendix C contains more details and graphs. Figure 2 shows the number of articles per year based on the year of publication as reported by Scopus. The plot shows first articles originating from 2008, after which the number of articles published per year steadily increases and stabilizes around 2013. There is an outlier in 2016, further analysis showed no anomalies in the data for this year other than an above average number of publications. Not all articles for 2021 were available at the time of writing.

Scopus also reports the publication source and IPDPS, SC, and PPoPP are the three most popular conferences for research into GPU optimizations, while **Concurrency & Computation** (**CPE**), TPDS, and JPDC are the three most popular journals. The distribution of the publication sources shows that research into GPU optimizations is ubiquitous and not necessarily specific to one certain community or publication venue (see Appendix C).

Figure 4 shows the number of times GPU architectures are mentioned in articles for each year. Firstly, it is clear that NVIDIA is by far the most popular vendor for GPUs, whereas other vendors

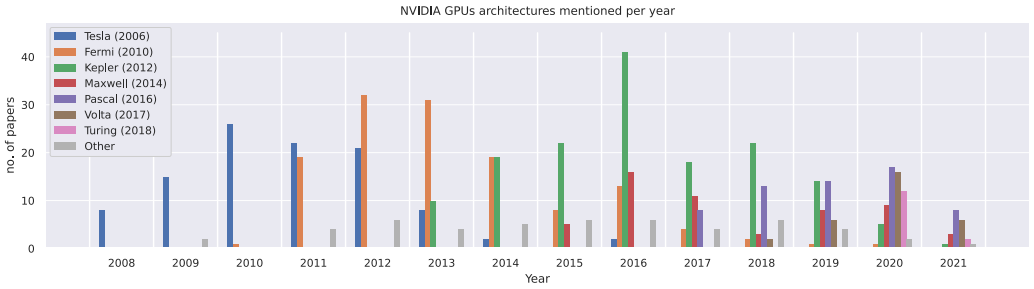


Fig. 4. No. of articles per GPU architecture per year.

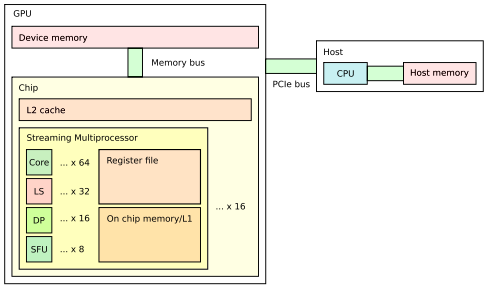


Fig. 5. Schematic of a representative GPU architecture.

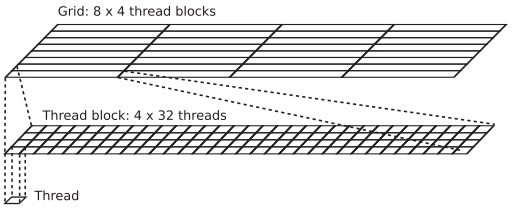


Fig. 6. Thread hierarchy on GPUs.

such as AMD and Intel are in the "Other" category being used in slightly more than 10% of the articles. Furthermore, there is a clear rise and fall of each architecture throughout the years, while GPUs in general have a long lifetime. For instance, even as late as 2019, Kepler GPUs were still the most common architecture even though the architecture had been released 7 years earlier.

In Section 6 we shall discuss 28 optimization techniques that we extracted from the set of articles. Figure 3 shows, for each optimization, the percentage of articles that mention that specific technique. It shows that *coalesced access*, *use dedicated memories*, *reduce branch divergence*, and *autotuning* are the most popular optimization techniques.

5 INTRODUCTION TO GPU PROGRAMMING

To understand the details of the optimization techniques, it is necessary to have an understanding of GPU programming. In this section, we discuss relevant architectural components of GPUs, the programming abstractions for GPUs, how those programming abstractions map to the architectural components, and how various parameters interact to achieve high performance.

5.1 GPU Architecture

In general, GPU architecture evolves very rapidly, so to prevent focusing on a specific GPU, we discuss a representative GPU architecture that does not exist in reality but explains enough of GPUs to understand the main concerns for optimization.

Figure 5 shows a high-level schematic of our GPU. Most commonly, GPUs exist as a **PCI-express** (**PCIe**) card in a *host* system with a *CPU* and *host memory*. The *GPU* itself is a PCIe card with a *device memory* and the *chip*, the processor itself. The chip consists of a large *L2 cache* and several ***Streaming Multiprocessors*** (**SMs**) as they are called in NVIDIA terms, in our case 16 SMs.

```
__global__ void vecadd(int n, float *c,
    const float *a, const float *b) {
  // Get our global thread ID from the
  // thread block ID and thread ID
  int id = blockIdx.x*blockDim.x+threadIdx.x;

  // Verify the bounds
  if (id < n)
    c[id] = a[id] + b[id];
}
```

(a) The kernel.

```
// Vector size
int n = 1000000;

// Number of threads in a thread block
blockSize = 1024;
// Number of thread blocks in the grid
gridSize = (int) ceil((float) n/blockSize);

// Execute the kernel with the specified
// grid and block sizes
vecadd<<<gridSize, blockSize>>>(n, c, a, b);
```

(b) Host code for the kernel.

Fig. 7. Vector addition in CUDA.

A SM consists in our case of 64 *cores* for arithmetic instructions such as single-precision floating point operations. An SM has 32 ***load/store units*** **(LS)** for loading data from the memory hierarchy into the *register file* and 16 ***dual precision units*** **(DP)** responsible for dual precision arithmetic. An SM has 8 ***special function units*** **(SFUs)** for transcendental functions. Finally, there is *on-chip memory*, a scratchpad for shared memory that is combined with the *L1* cache.

5.2 Programming Abstractions

A *kernel* is a function that is launched from the host to run on the GPU. In CUDA or OpenCL, a kernel is written for a single thread and when the kernel is launched, the same function is launched with at least one *thread*, but typically many more.

In GPU programming models threads are organized in a hierarchy. Threads are organized in *thread blocks* and thread blocks are grouped in a *grid* as shown in Figure 6. In this case, we have a 2-dimensional thread block of 4×32 threads, with 128 threads in total, and we have a two-dimensional grid of 8×4 thread blocks. Based on the thread block configuration, each thread acquires a unique identifier to steer the thread to a specific data element or task.

Threads *within* a thread block can communicate with each other by means of *shared memory*: a value written into shared memory by a thread, can be read by the other threads. Since threads have no means to know when another thread wrote a value they want to read, they require synchronization which happens by means of a barrier. Historically, threads *from different thread blocks* cannot communicate with each other during the lifetime of a kernel (although modern architectures have relaxed this requirement, see Section 6.3.10 for more details). So, the only way to allow synchronization among different thread blocks is through separate kernel launches.

In Figure 7(a) we see a simple CUDA kernel that performs an element-wise vector addition of arrays `a` and `b` into `c`. The thread computes its global thread ID based on the thread block ID, thread block dimension, and thread ID within the block to perform the element-wise addition. In Figure 7(b) we see the host code determines the thread block and grid size used to launch the `vecadd` kernel. A thread block here has 1,024 threads and we launch in the order of 1000 thread blocks. This degree of parallelism is common for GPU codes.

Although not explicit in the programming model, it is useful to know that threads within a thread block are grouped in *warps* of 32 threads for NVIDIA and typically 64 for AMD GPUs. Instructions are scheduled per warp, so each thread in the warp executes the same instruction.

5.3 Mapping the Programming Abstractions to the GPU

The programming abstraction *thread block* is mapped to the architectural component SM. The functional units such as *cores* execute the instructions for warps and *shared memory* resides in *on chip memory*. This shared memory is allocated per thread block.

The compiler determines the number of thread-private registers a thread requires. It is in general not possible to communicate by means of registers. NVIDIA GPUs have warp-shuffle functions

with which threads within a warp can communicate bypassing shared memory (see Section 6.1.2). If threads require more registers than available in the register file, the registers may be "spilled" to device memory, making execution more expensive, although these values are typically cached.

GPUs are throughput-oriented processors. Scheduling occurs by means of warps and as soon as a warp performs a long-latency operation, such as a memory access from device memory, execution continues immediately with (1) either the next independent instruction in the instruction stream of the warp, or (2) other warps that can do useful work. This means that GPUs are designed to overcome long-latency operations with scheduling other warps, called **Thread-Level Parallelism** (**TLP**) and **Instruction-Level Parallelism** (**ILP**). The latter form of parallelism is often overlooked, even in programming guides from vendors [373].

SMs can execute multiple thread blocks simultaneously, typically 8 or 16. The number of thread blocks that run on an SM depends on three factors: (1) the number of threads per thread block (thread block size), (2) the number of registers per thread, and (3) the amount of allocated shared memory per thread block.

The maximum *thread block size* is typically 1,024. Running a kernel with this thread block size typically limits the number of thread blocks to 1 or 2. Lowering the thread block size allows the SM to execute more thread blocks concurrently. The register usage per block depends on the number of registers needed by each thread and the thread block size. Finally, the shared memory on an SM is also partitioned among thread blocks. If a thread block allocates all of the *shared memory*, then the number of thread blocks per SM is 1.

5.4 Balancing Parameters for Performance

From the above discussion it is quite clear that many parameters interact with each other and that a proper balance between the various parameters is important for performance. Since our representative GPU has 16 SMs, each capable of processing multiple thread blocks, it is also clear that generally, the number of thread blocks in a grid needs to be high. So, in general, the SMs are overprovisioned also for load balancing reasons.

The general belief is that if you choose the number of registers, shared memory, and/or thread block size such that the occupancy is low, it is not possible to achieve high performance. However, Volkov [373] states that occupancy as a measure of utilization only counts for TLP, whereas it is also important to take *instruction-level parallelism* into account. Moreover, they state that low occupancy is even required for certain kernels to achieve high-performance.

Achieving high-performance is often a balancing act, where TLP and ILP are to some extent communicating vessels but with buckets in between so that the result is not always immediately clear unless one of the buckets overflows. For example, increasing instruction-level parallelism is often beneficial until you hit a resource limit on the SM, for example, the number of registers decreasing the number of active blocks and therefore TLP. In Appendix D we highlight Volkov's performance model [373] that defines utilization in terms of TLP and ILP.

6 OPTIMIZATION TECHNIQUES

In this section we give an overview of the optimization techniques extracted from the literature. This section contains a broad overview of the techniques organized by the four themes *Memory Access* (Section 6.1), *Irregularity* (Section 6.2), *Balancing* (Section 6.3), and *Host Interaction* (Section 6.4). Since memory access is typically much slower than computation, one of the most important themes is *Memory Access*. We subdivide this theme into the sub themes *On-Chip* and *Off-Chip*, both areas having their own challenges and solutions. Since GPUs are highly regular architectures, another important theme is *Irregularity* discussing how irregular algorithms can be efficiently mapped to GPUs. In theme *Balancing*, we discuss techniques subdivided into three

subthemes: *Balancing the Instruction Stream*, an important aspect of performance as explained in Section 5.4, *Parallelism-related Balancing* important for granularity and load balancing, and finally *Synchronization-related Balancing*, an important aspect to increase performance. Finally, in the theme *Host Interaction*, we discuss the interplay between GPU as an accelerator and the host.

We do not consider these themes as a classification per se but more as a means to present the optimizations. Membership to a theme is also not always completely clear. For example, *load balancing* is in theme *Balancing* but is often necessary for irregular applications, so we could have chosen *Irregularity* as its theme as well, where we prefer *Balancing*. Readers interested in more detailed descriptions of the optimization techniques are referred to Appendix A that follows the same structure in themes and contains a table of contents of the described techniques.

6.1 Memory Access

Optimizing for *memory access* is an important theme in GPU programming because GPUs are computationally very strong due to the high degree of parallelism whereas memory access is typically much slower than computation. This disparity is reflected in the elaborate memory hierarchy that GPUs have and in the fact that many techniques center around improving memory access. We subdivide the techniques into *On-Chip*, relating more to the memories on the GPU chip itself, and *Off-Chip*, relating more to device memory (see Figure 5).

6.1.1 On-Chip — Use Dedicated Memories. Optimizations focused on the area of memory access are the most common because most often memory access problems impede high performance. Besides two levels of caches, GPUs also provide dedicated memory spaces that are mapped into fast memory. *Constant Memory* is used for read-only data in a kernel and broadcasting this data to multiple threads only results in one memory transaction [107, 200].

Texture Memory is also read-only memory that is optimized for accesses with the spatial locality in two dimensions which many exploit for general-purpose computing [203, 264, 349, 364, 418]. However, it also finds use in handling boundary values [193, 195, 206, 318, 319], improving uncoalesced access [53, 435], automatic integer to float conversion [234], or simply for the fact that accesses are cached [46, 78, 230, 247, 339]. Liu et al. have an interesting application for texture memory where they fetch packed string data with one texture fetch for gene sequence alignment [231].

Shared Memory resides in the on-chip memory of a streaming multiprocessor and data in shared memory can be shared between threads within a thread block [84, 127, 186, 280] where they synchronize by means of barriers. Stalling warps as a result of these barriers can be mitigated by decreasing the amount of shared memory for a thread block to allow multiple thread blocks to run on a single streaming multiprocessor [170].

Besides sharing data other uses of shared memory are decreasing expensive memory operations [203, 275], allowing uncoalesced access [60, 66, 116, 127, 129, 278, 315, 319, 329, 414], or minimizing the impact of irregular access [64, 127, 203, 325]. Shared memory can be very effective when data reuse in the form of temporal [5, 14, 79, 92, 149, 162, 187, 217, 263, 304, 363, 407, 408, 414] or spatial locality [107, 162, 251, 257, 304, 310] can be exploited, something that can be improved with blocking [162, 209, 249, 257, 304, 310, 315, 319, 345, 350, 353, 407, 408, 425] (Section 6.1.7) or kernel fusion [223] (Section 6.1.8).

Since shared memory is organized in banks that can result in bank conflicts that reduce the performance, there are several techniques to mitigate this, such as padding [97, 107, 127, 137, 200, 287, 304, 345, 375, 390, 407, 408, 414], reordering data [66, 209, 315, 318, 431], or remap threads to data [52, 134, 182, 182, 318, 412, 412]. Reddy et al. exploit the commutativity of operations [311] to reorder data access. Bernstein et al. wrote a search tool that assigns threads with data such that almost all bank conflicts are eliminated [49].

6.1.2 On-Chip — Use Warp Functions. NVIDIA introduced warp-voting functions in the Fermi architecture (2010) with which threads within a warp can reach consensus without the need for explicit synchronization and going through shared memory. Starting with the Kepler architecture (2012), NVIDIA extended these *warp vote functions* with *warp shuffle functions*.

The warp vote functions are used to evaluate predicates at warp-level and then broadcast the result of such evaluation to all threads in the warp [37, 38, 115, 231, 298]. The largest use-case for warp shuffle functions in the literature is to share data intra-warp without going through shared memory [7, 12, 14, 47, 65, 70, 150, 155, 204, 210, 223, 328, 380, 411, 422].

Interesting is how Barnat et al. [44] use these functions for a warp-level caching system. Another specialized use-case for warp shuffle functions is to implement collaborative algorithms such as reduction [35, 42, 117, 266, 282, 311] and prefix-sum [185].

6.1.3 On-Chip — Register Blocking. Registers form the fastest memory available on GPUs [373] and as such, can be very effective for increasing performance by storing often-used values in them. Since registers are not shared among threads, registers are mostly effective for data reuse with temporal locality and less for spatial locality although some use warp-shuffle to distribute values among threads in a warp [7, 12]. Because of these restrictions, register blocking, sometimes called temporal blocking, is highly related to loop unrolling [83]. It is also related to *varying work per thread* (Section 6.3.4) because sometimes the same data is reused for multiple elements that a thread computes. Hong et al. [152] indeed acknowledge that varying the work per thread can achieve register blocking.

Register blocking is often used in 3D stencil computations [83, 174, 257, 262, 302, 372, 379, 421, 433], serializing over the Z dimension and using registers to reuse values in this dimension, studied extensively by Matsumura et al. [251]. Some note that registers are especially useful for accumulator values on GPUs since the values in registers cannot easily be shared with other threads [80, 137]. Tan et al. note that registers cannot be indexed and therefore techniques such as macro expansion or templates have to be used [349] to have control over arrays of registers [345].

6.1.4 On-Chip — Reduce Register Usage. Register usage is an important factor in the occupancy and many applications are limited in performance due to their high register usage which can lead to *register spilling* where thread-local variables are stored in slow off-chip device memory, although the spilled variables are typically cached.

Typical techniques are using pointer arithmetic, minimizing temporary variables, and rewriting arithmetic operations [135]; moving temporary variables to shared memory, packing small types into larger types (e.g., pack 4 bytes into 1 integer), and refraining from storing values that can be recalculated [358] (see Section 6.1.5). Unrolling loops can also save registers due to removing the need for induction variables [318, 331] (see Section 6.2.1). Other techniques are changing the algorithm [104, 308], force the compiler to restrict register usage [183], or –more radically– write a custom register allocator [49].

6.1.5 On-Chip — Recompute. The *recompute* optimization recomputes one or more values that have already been computed in the past to avoid communication [149, 205], and reduce memory operations or footprint [95, 201]. It can be seen as the opposite of precompute and is often beneficial on modern GPUs, where compute capabilities far exceed memory and communication bandwidth. Particularly interesting is the case of Lefebvre et al. [205], where they combined *kernel fusion* (see Section 6.1.8) with *recompute* to reduce expensive synchronization through global memory and therefore avoided storing intermediate results.

6.1.6 Off-Chip — Coalesced Access. Coalesced access is the second-most applied optimization and refers to the fact that if the combined memory accesses of a warp of 32 threads satisfy a set of

coalescing rules, data can be fetched in less than 32 transactions from the device memory. The set of coalescing rules varies per architecture and have become less strict over time.

There are various ways to achieve coalesced access [319] and the complexity of achieving coalesced access depends on several factors, such as the data layout, the organization of threads, and the availability of shared memory. If the data layout allows, means to achieve coalesced access are reorganizing threads [86, 123, 160], choosing a good thread block size [206], tiling [34, 277, 278, 353], by choosing a different parallelization strategy [96, 139, 398], or perform complex indexing operations such as space-filling curves [19, 250].

Another technique is loading data from global memory in coalesced fashion into shared memory from which the uncoalesced access can happen [89, 104, 104, 105, 117, 120, 129, 187, 271, 277, 282, 319, 329, 379, 414–416, 423] or stage data for coalesced stores to global memory [37, 60, 110, 127, 127, 182], possibly in combination with tiling [278, 318].

It is also possible to adapt the layout in global memory [337], transforming to a struct of arrays from an array of structs [59, 72, 86, 112, 212, 319, 331, 344, 428], switching from row-major to column-major arrays [72, 329] or other transpositions [19, 85, 316, 337, 344, 432], or applying padding [9, 117, 135, 136, 182, 205, 250, 316, 331]. An interesting case is shown by Ito and Nakano where two different layouts for optimal polygon triangulation perform better in different stages of the algorithm [166]. They conclude that reorganizing data from one layout to the other is faster (including the transformation time) than sticking to one layout.

Coalescing is difficult to achieve on applications with a high degree of random access, such as highly irregular applications such as sparse matrix-vector multiplication and graph processing. One approach is caching the expensive random-access loads [89]. Bell and Garland [46] show that various sparse matrix-vector multiplications formats support or are designed to maximize coalescing and Zhong et al. [434] introduce a new graph format with a data layout such that it supports coalesced access. Ashkiani et al. present a hash table that supports updates and is designed with coalesced access in mind [38]. The hash table uses chaining and the linked lists use a "slab", a linked list node with multiple key-value pairs and one next pointer, such that operations can be performed with warp instructions and memory access is coalesced.

6.1.7 Off-Chip — Loop Tiling or Spatial Blocking. Blocking [277] or loop tiling [85] are two frequently used terms in the literature that refer to the same concept, which is partitioning the data or the computation in such a way that it is processed one block at a time. We refer to this optimization as blocking, or more specifically as *spatial blocking*, to distinguish between spatial and *temporal blocking* (Section 6.1.8). Spatial blocking gives the programmer control over locality [39, 70, 104, 137, 138, 188, 269, 277, 304]. This allows the kernel to more effectively exploit data reuse in L1/L2 caches [188, 269, 330, 358, 382], the texture cache [413], registers [138, 162, 407], and shared memory [104, 138, 162, 277, 315, 353, 383, 396, 407, 425]. Blocking may have other beneficial effects, for example, Střelák et al. [338] use blocking to reduce warp divergence (See Section 6.2.2).

The dimensions of the block are often directly linked to the dimensions of the thread block [182, 320] but with the block dimensions decoupled from the thread block dimensions, programmers have the opportunity to vary the amount of work, which is considered another optimization, see Section 6.3.4. Auto-tuning (see Section 6.3.6) is frequently used for selecting the optimal thread block dimensions (see Section 6.3.5) in combination with tile or block dimensions [166, 209, 287, 316, 320, 389, 413].

6.1.8 Off-Chip — Kernel Fusion. Kernel fusion (sometimes also called kernel merge [205, 323, 346, 422] or kernel unification [173]) is analogous to loop fusion, defined as "collapsing consecutive and dependent loops" by Carabaño et al. [62, p. 220]. However, there is a crucial difference between kernels and loops since kernels also allow synchronization at the device memory level. If two

threads *within* a thread block need to synchronize, they can use shared memory and a barrier to do so. When two threads *in different thread blocks* need to synchronize, this is only possible with different kernel launches acting as a barrier [18, 62, 77, 111, 422], although there are techniques to allow global synchronization without kernel launches [400], see Section 6.3.10.

Sarkar et al. show that solely optimizing each kernel does not necessarily lead to the most optimized configuration but that fusing kernels can result in better execution times [323]. An important reason to apply kernel fusion is reducing global memory accesses if two kernels use the same data [4, 31, 65, 86, 111, 117, 155, 188, 205, 223, 224, 255, 256, 323, 418, 433]. Some also mention improved cache benefits [77, 188], improved locality [77, 346], and data reuse [4, 5, 102, 138, 188, 223, 323]. Finally, it is often mentioned that kernel fusion reduces kernel launch overhead [4, 5, 18, 111, 117]. As with many optimizations, kernel fusion can introduce new performance trade-offs, as the resulting fused kernel likely requires additional registers and shared memory [4, 62, 205]. Appendix A.1.8 discusses many more considerations for kernel fusion.

6.1.9 Off-Chip — Software Prefetching. Because of high memory latencies, GPUs are designed to switch to an eligible warp (a warp that can make progress) when blocked on long-latency memory operations or other stalls. However, by means of *software prefetching* it is possible to exert more control over the availability of data.

Prefetching (sometimes called "software pipelining" [407]) has been applied in various dense linear algebra kernels, such as matrix-vector multiplication [8, 9], matrix-matrix multiplication [6, 27, 39, 162, 237, 315, 320, 348], stencil operations [149, 193, 262, 379], typically in combination with tiling (see Section A.1.7) where data for the next iteration is prefetched [320]. There are a few instances of this being called *double buffering* [10, 137, 249] since two buffers are required (one for the current tile and one for the next tile), although this term is more commonly used in the context of overlapping computation and data transfers (see Section 6.4.1).

Interesting cases are Bauer et al. [45] that present an approach for programming GPUs in which separate warps fulfill separate roles (*warp specialization*) one of which is prefetching. Wu et al. [396] investigate the effectiveness of inter-block barrier synchronization over invoking the same kernel many times (see Section A.3.10). They prefetch data before an inter-block barrier, data that is required just after the barrier, something that is impossible to achieve when using individual kernel invocations.

6.1.10 Off-Chip — Compress Data. Since memory is more abundant in the host than on GPUs, *compressing data* is a way to reduce the memory footprint, both in terms of storage and data transfers [53, 321], but also in terms of reducing pre-processing time as shown by Neelima et al. [279]. This optimization is widely used for sparse matrices [241, 242, 279, 282], but also for the longest common subsequence problem in the context of sequence alignment [297] and regular expression matching [273].

6.1.11 Off-Chip — Precompute. The main idea is to perform computations offline, before the main processing starts, and then reuse these precomputed values whenever necessary [402]. Often *precompute* is seen as a tradeoff optimization between time and space [338], where time is saved by reusing some precomputed and stored values, at the cost of increased memory pressure. In this case, this optimization can be seen as the opposite of *recompute*, described in Section 6.1.5. In general, this optimization can also be seen as one of the possible ways to implement another optimization, and *reduce redundant work* (Section 6.2.5).

The literature offers multiple examples [23, 112, 303, 308] of precomputing on the host and dividing computations between the host and GPU (Section A.4.2). Interesting cases are Greathouse et al. that precompute how many rows can fit in shared memory [129] and Zhou et al. [435] that

combine precompute with the use of texture memory by precomputing the city distance matrix for the traveling salesman problem, and then storing it in texture memory.

6.2 Irregularity

GPUs are highly regular architectures but applications and algorithms are not necessarily very regular. This subsection discusses various techniques on how irregular algorithms can be efficiently mapped to GPUs.

6.2.1 Loop Unrolling. *Loop unrolling* is an optimization technique in which the body of a loop is explicitly repeated multiple times [270]. Unrolling of a loop can be done manually (by explicitly duplicating the body of the loop) or automatically (either using macros, C++ templates [7], or compiler directives).

The benefits of loop unrolling are reducing loop-related instructions (such as branches and address calculations [242, 270]), increasing the opportunity for ILP [242, 270, 282, 345], due to offering more independent instructions to hide latencies; and enabling further compiler optimizations such as storing arrays in registers [67, 216, 320], removal of loop-dependent branches [112, 318, 439], and vectorization of instructions [54, 67, 309].

However, although unrolling loops is often beneficial, unrolling by a factor too large might hurt performance [105, 169, 270, 319]. The unroll-factor is an important consideration [270, 287, 309] that often differs per kernel and device [287].

6.2.2 Reduce Branch-Divergence. *Branch divergence*, sometimes also called *path divergence*, is a problem for SIMD or warp-based architectures such as GPUs where a warp executes instructions in lock-step. On a branch that depends on the thread index, there is a chance that different threads in a warp have to take different paths on a branch. On a GPU this results in all threads taking both branches where the threads that should not take the branch do not write the result. This hurts performance because both branches are executed serially. If the instructions within the branch are simple enough, the branches can be replaced with predicated instructions in which only the threads with a true predicate write back the result of the instructions [52, 112].

There are various techniques for reducing branches such as *removing branches* altogether, replacing them with arithmetic instructions [59, 68, 231, 358, 369, 379], sometimes called *branch refactoring* [68], or *algorithm flattening* [369]. Using lookup-tables (*instruction stream normalization*) [172] or kernel fission [80] (Section 6.2.4) is also possible. Another technique is *reducing branches*, by notifying threads of performed work [64, 65], using loop unrolling [112, 308, 311, 439], or applying the loop-based *iteration-delaying* technique in various forms [140, 431] where loop iterations are delayed to increase the chance that threads take the same path. Branches can also be reduced by trading them off against another metric, such as performing redundant work [296], introducing errors [324], or serial execution [154]. Another technique is *reducing the effect of branches*, for example by moving code outside of branches (*branch distribution*) [140] or simplifying the instructions to enable predication of instructions [198]. The *thread/data remapping* technique [426] does not affect branches directly but causes decreased branching by changing the parallelization scheme [183, 206, 426], sometimes by grouping similar threads (*stream compaction*) [121] or sorting the data to realize similar branches for threads [59, 276]. Branching can also be reduced with *data layout changes* such as padding [64, 396, 435] or introducing sparse formats [34, 109, 186, 242, 342, 352, 366]. Finally, branching can be highly affected by algorithm design [122, 428]. Appendix A.2.2 contains many more details.

6.2.3 Sparse Matrix Format. GPUs can be useful for sparse linear algebra operations (most notable SpMV). The memory format used to represent a sparse matrix has an important influence

on performance and an abundance of different formats have been proposed. Bell and Garland [46] give an overview of the essentials and Muhammed et al. [266] give a recent review.

The ELL format (also named *ELLPACK*) [46] takes the non-zeros from each row and adds padding to enforce equal row length, leading to a highly regular format that is suitable for GPUs. Many ELL-based formats for GPUs have been proposed such as *ELL-R* [366], *SELL* [264], *ELLR-T* [100], *AdELL* [240], *AdELL+* [242], *CoAdEll* [241], and *SELL-C-σ* [192]. Other GPU-friendly formats have been proposed for cases where the number of non-zeros per row is highly variable: *CRSD* [342] is suitable for diagonal sparse matrices, *ASCR* [35] helps to reduce thread divergence of regular CSR, *SIC-CSR* [109] reduces imbalance between warps, *CEL* [436] allows out-of-core processing, Anh et al. [26] significantly improve memory throughput by using *superrows*, Ashari et al. [34, 36] use auto-tuning to find the optimal parameters. Hybrid formats combine several existing formats: Bell and Garland [46] propose *HYB* (combines COO and ELL), Yang et al. [413] choose different formats per row (ELL or CSR) based on a heuristic, and Su et al. [339] partition the matrix and select one of 9 formats for each submatrix based on a trained performance model. Compression of formats can help to increase the memory throughput at the cost of additional computation to decompress: Xu et al. [403] use index compression, Tang et al. [352] use delta encoding, and Yan et al. [409] use bit flags for compression. The above formats assume static data, but dynamic formats also exist, often for graph processing allowing adding and removing edges at runtime [253].

6.2.4 Kernel Fission. *Kernel fission* is either the process of dividing a single kernel into multiple ones, or splitting a single kernel iteration into multiple iterations. It can be seen as the opposite of kernel fusion (Section 6.1.8). The way this optimization improves performance is by better resource utilization, achieved through simpler and more regular kernels.

Kernel fission is used for improving regularity in sparse matrix-vector [35, 81, 89] but also in dense matrix-vector multiplication [8], for simplifying the structure of complex kernels [23, 52], for *reducing branch divergence* [63, 80] (Section 6.2.2), and many split kernels to improve *auto-tuning* (Section 6.3.6) [19, 88, 112].

6.2.5 Synchronization-related Balancing — Reduce Redundant Work. This optimization consists of avoiding to perform work that is considered redundant. Its implementation is usually algorithm-specific and it is the opposite of recomputing values (see Section A.1.5). The optimization is often used in processing irregular data structures, such as graphs [253, 255, 298], in the dynamic programming [296], or linear algebra [34].

6.3 Balancing

Although GPUs are relatively simple architectures, many architectural details are highly interconnected and require careful balancing to achieve performance. This section discusses several techniques to map parallelism to hardware and to achieve balance in resource usage. We make a distinction into three subthemes: *Balancing the Instruction Stream*, *Parallelism-related Balancing*, and *Synchronization-related Balancing*.

6.3.1 Balancing the Instruction Stream — Vectorization. This optimization consists of using vectors instead of scalar variables, and manipulating such vectors by means of instructions that are applied to all the elements of a vector. The most common use of vector data types is to access memory [12, 45, 54, 67, 78, 80, 102, 134, 188, 273, 287, 415, 416] and it can also help with improved *coalesced access* [12, 54, 67, 134, 188]. Other uses are *varying the work per thread* [119, 169] or the work in a loop iteration [282, 309], accessing sparse matrices [214, 339], improving data storage [220], optimizing numerical operations [314], and reducing branches [23].

6.3.2 Balancing the Instruction Stream — Fast Math Functions. These optimizations consists of using approximate mathematical functions that are significantly faster than the standard ones, and are usually implemented in hardware in the special function units. This optimization can be manually applied by the programmer, using intrinsics [59, 64, 68, 363], or it can be enabled for all mathematical functions by the compiler [367]. An interesting case is Chakroun et al. [68] which use fast math functions to *reduce branch divergence* (Section 6.2.2), replacing conditions with the approximated functions.

In recent architectures NVIDIA introduced tensor cores, functional units specialized for machine learning applications with tensor operations that support varying floating point precisions. Yan et al. [407] studied this operation in detail analyzing the throughput and latency in relation to architectural bottlenecks. Typically these tensor cores are targeted via libraries such as cuBLAS or cuDNN but they can also be targeted directly [137, 350, 407, 425].

6.3.3 Balancing the Instruction Stream — Warp-Centric Programming. This optimization makes warps an integral part of the programming model where code is structured around the idea of warps as units of computation. Particularly interesting is the relationship between this optimization and *reduce synchronization* (see Section A.3.8), as one of the effects of *warp-centric programming* is a reduction in synchronization overhead. One way to implement *warp-centric programming* is to reorganize the code so that work is not assigned to threads, or thread-blocks, but to warps instead [46, 65, 150, 154, 179, 328, 437].

Common reasons to use *warp-centric programming* are implementing some form of load balancing [121, 239, 281], hiding latency [10], and implementing nested parallelism [417]. Another application is "warp specialization", where warps perform different work [45].

6.3.4 Parallelism-related Balancing — Varying Work per Thread. Varying the amount of work assigned to each thread and thread block is one of the most important and generally applicable code optimizations in GPU programming. There are different terms that refer to the same concept, such as *1x2 rectangular tiling* [320], *strip mining* borrowed from decreasing loop-count with *vectorization* [375], *thread-block merge* and *thread merge* [414], or *work-item merge* [157], *thread coarsening* [243] and *block coarsening* [152] or *block merge* [119].

In general, increasing work per thread increases the exploitation of data reuse in kernels that exhibit data reuse, at the cost of increased resource usage in terms of registers and shared memory. The optimal amount of work per thread is generally difficult to find and often included in auto-tuning optimizations (Section 6.3.6).

6.3.5 Parallelism-related Balancing — Resize Thread Blocks. In general in GPU programming, programmers have quite some freedom in choosing the parameters *number of threads per thread block* and the *number of thread blocks*. For example, for a simple vector addition of 2^{20} elements where each thread adds one element can be executed with 1,024 threads per thread block and 1,024 thread blocks, 512 threads and 2,048 thread blocks, 256 threads and 4,096 thread blocks, and so on. In some cases, the precise number of threads per thread block will not matter much [124] but in more complex kernels performance may differ significantly [205].

Resizing thread blocks impacts performance because it influences register usage [119, 173, 201, 209, 363, 372], the number of (independent) thread blocks per SM, keeping the compute units busy in case of barriers [119, 125, 173], and the amount of shared memory used by a thread block [125, 186, 195, 201, 209, 372] which also influences the number of thread blocks per SM [173]. Although this optimization is often performed for the same reasons as *varying the work per thread* (see Section 6.3.4), it does not necessarily change the amount of work per thread, although this is possible [182, 209, 372]. The number of threads per thread block is one of the most used parameters

applied in auto-tuning (see Section 6.3.6) because it is easy to change but also difficult to reason about and find [201, 205].

6.3.6 Parallelism-related Balancing — Auto-tuning. We have seen that even for *thread block size*, one of the easiest to change parameters, it is often difficult to find an optimal configuration [201, 205]. *Auto-tuning* is the process of automatically exploring the configuration space to find the optimal configuration of a set of parameters. It has been applied in various domains [68, 125, 166, 348, 405, 410, 413] but is prevalent in *sparse formats* [16, 34, 36, 78, 101, 129, 214, 242, 266, 272, 409, 420, 432] (Section 6.2.3), stencils [79, 83, 119, 316, 351], dense matrix multiplication [191, 209, 277, 282, 380], and linear algebra [4, 6, 9, 10, 235], in particular the Cholesky factorization [5, 7, 138, 375].

Interesting approaches are the works of Yoshizawa et al. [420] that tune the parameters of a sparse format; Ashari et al. [34] that tune based on the sparsity of the matrices; Choi et al. [78] that combine modeling and benchmarking, and Garvey et al. [119] which is based on machine-learning. Generic tuners and tuning frameworks are CLTune [287], PADL [58], Kernel Tuner [389], OpenTuner [147], and the Kernel Tuning Toolkit [304] that provides an API to run and tune kernels. Besides tuning for performance, another often mentioned goal is performance portability [9, 191, 209, 214, 338, 351].

6.3.7 Parallelism-related Balancing — Load Balancing. GPUs provide multiple levels of parallelism and on each level load balancing is important for performance. We discuss load balancing on several of those levels. *Load balancing* on the lowest level, *within warps* seems similar to *warp divergence*, but the latter refers to the problem that threads in a warp execute both paths of a branch serially, so there is duplication of work, whereas *load balancing* refers to the problem that some of the threads do not perform useful work whereas others do, so there is no duplication of work. Several articles target these both goals simultaneously [34, 109, 154, 183, 242, 428].

Graph processing on GPUs typically leads to load-imbalance and often implemented techniques are *defer outliers* using a global work list [154], *CTA+Warp+Scan* dividing work over a thread block or warp [77, 255, 384, 411], and *Load-Balanced Partitioning* grouping edges [84, 384]. Brahmakshatriya et al. [55] provide a good overview of various techniques. The area of sparse matrices is closely related to graph processing and various formats are designed for load balancing within warps [35, 239, 266]. Khorasani et al. [185] abstract from sparse formats and generalize the main pattern to *nested parallel patterns*. They propose *Collaborative-Task-Engagement* that executes coarse-grained tasks that can be split up in more fine-grained tasks.

Load balancing *within a thread block* can be mitigated by using a global worklist [154], donating work between threads using shared memory [275], or sorting data [430]. Load balancing *among thread blocks* is typically not a problem because the GPU programming model expects oversubscription of thread blocks onto the Streaming Multiprocessors. However, for irregular applications, the *persistent threads model* [132] can be beneficial [46, 155, 275]. Tzengy et al. employ this where threads remain active for the duration of the kernel and keep stealing work. They divide the work into bins for a ray-tracing application [360] and employ work-stealing and -donation which means that as soon as a bin is overflowing with work, the thread block donates tasks to another streaming multi-processor in a round-robin fashion. Load balancing between the GPUs and CPU (see Section 6.4.2) is often implemented with a static partition, where a predetermined percentage of the total amount of work is offloaded to the GPU, and the rest is left on the CPU [72, 355, 356], but dynamic solutions exist as well [249, 377].

6.3.8 Synchronization-related Balancing — Reduce Synchronization. On a highly parallel architecture synchronization can have a considerable impact on performance, especially when there are not enough threads to hide the latency of a barrier. Avoiding synchronization often depends

on algorithmic changes to prevent affecting correctness [97, 432]; an interesting example of this is provided by Petre et al. [303] using a version of the Cooley-Tukey FFT algorithm that does not require synchronization. Other common techniques are increasing the amount of work per thread [79, 202, 280, 345, 405, 424] (see Section 6.3.4), *using shared memory* [345, 424], and *blocking* [79, 280]. In particular, Reddy et al. [311] achieve the result of avoiding synchronization by completely unrolling a tree-based reduction algorithm. An interesting conflicting application of *reduce synchronization* is the work of Bauer et al. [45] that replace barriers with more fine-grained primitives, compared to Nasre et al. [274, 275] that replace fine-grained synchronization with barriers.

6.3.9 Synchronization-related Balancing — Reduce Atomics. Atomic operations allow programmers to perform parallel memory updates without conflicts. However, although important for the correctness of many algorithms, atomic operations introduce overhead due to synchronization.

One option is to *avoid atomics* altogether, which is often application-specific, for example, applying an application-specific partitioning of data [66], relaxing the application's memory model [65], or maintaining an approximate view of the data [255]. The lock-free implementation of Xian et al.'s *inter-block synchronization* strategy [400] (Section 6.3.10) is one that does not rely on atomics.

Reducing atomics, so reducing the number of atomic operations is often application-specific as well, for example by aggregating push operations on a graph [298] or removing instances with the highest degree of possible conflict [28, 321]. More standardized techniques are also possible, for example, using shared memory for atomics [206] or using *shuffle instructions* (Section 6.1.2) to reduce expensive atomic operations [28, 115].

6.3.10 Synchronization-related Balancing — Inter-Block Synchronization. Historically the only way for synchronization at a scope larger than the thread block is via multiple kernel calls. Researchers have proposed various methods to implement *inter-block synchronization*, for example, to be able to prefetch data (Section 6.1.9) for after the global barrier [396]. There are various implementations of which two by Xiao et al. [400] are the most used [18, 77, 132, 397], one using atomic operations and one lock free.

In 2017 NVIDIA released a new version of CUDA (9.0) that supports cooperative groups. This allows programmers to define a custom group of threads for synchronization, among which groups larger than thread blocks or the whole grid of thread blocks. This advanced inter-block-synchronization requires architectural support provided by Pascal (2016) and later architectures.

6.4 Host Interaction

Many optimizations are targeted at the GPU kernel alone. However, the overall application can considerably be improved with proper interaction between host and device, either by ensuring optimized communication between the devices or by dividing computation among the CPU and GPU.

6.4.1 Host Communication. To improve application performance, it is often necessary to optimize the communication over the PCI-express bus between the host and GPU device. The most effective is eliminating communication altogether by executing all or as much as possible of the algorithm on the GPU [59, 95, 138] or keeping as much as possible data on the GPU [29, 40, 88, 173, 223]. Another approach is to compress the data that is to be transferred over the PCI-e bus [53, 279].

An interesting approach is applying dynamic parallelism (where threads are able to launch kernels as well) to move the control loop of the host to the GPU [210]. Hong et al. [153] simplify and speed up communication with a system that provides users with a unified memory view of the CPU and GPU memory in OpenCL, similar to what CUDA provides.

Approaches to improve host communication are using pinned memory [106, 128, 136, 205, 235, 326], pinned and/or mapped memory for overlapping with computations [29, 40, 197, 250, 377],

or using streams or command queues for elaborate schemes [12, 70, 92, 98, 235, 237, 419] (called *streamlizing* by Ma et al. [237]) such as pipelining [79, 128, 174, 204, 253, 316, 437]. Often it is necessary to manage the buffers, applying double buffering [195, 249, 364] or even triple buffering [367]. Zhang et al. [426] have an interesting pipelining scheme to hide the latency of *thread-data remapping*, discussed in Section 6.2.2.

6.4.2 CPU/GPU Computation. This optimization technique consists of splitting the computation between CPU and GPU, using both devices to perform useful work [72, 112].

A common use-case for this optimization is when an application can be divided in a set of independent, or semi-independent, tasks [23, 102, 355, 356]. With dependencies between CPU and GPU tasks, optimizing the data transfers also becomes important [102]. Load balancing is often also important and discussed in Section 6.3.7.

7 ANALYSIS OF THE OPTIMIZATION TECHNIQUES

In the previous section, we have presented a high-level overview of the optimization techniques in the literature. In this section, we analyze the optimizations from different perspectives to understand how they relate to each other and what role they have in the optimization process. We also discuss the evolution of GPU architectures over the years and its impact on optimization techniques. Finally, we give an overview of the potential performance impact of each optimization.

7.1 Based on Application Specifics

The first analysis is based on application specifics. We discuss several properties of GPU applications or kernels that direct the reader to relevant optimization techniques. The properties we discuss below can be regarded as axes in a multi-dimensional space, but unfortunately, the axes are not completely orthogonal. We, therefore, discuss each property separately.

The first property is *application as a whole* against *kernel* techniques. Unless a specific kernel is a clear bottleneck, it is often more useful to optimize the pipeline of kernels. The next property is whether a kernel is *compute* or *memory bound*, something that also leads to different techniques that apply. Highly related, but not completely orthogonal, is whether a kernel exhibits *data reuse* or not, a property that also influences which optimizations apply. Finally, we make a distinction between *irregular* or *regular* kernels.

7.1.1 Application as a Whole or Kernel. Typically, a GPU application consists of multiple kernels in which optimizing only a few of the kernels may hardly benefit overall application performance. Examples of such applications with a pipeline of kernels are provided by Sarkar et al. [323] with a typical pipeline of kernels for molecular dynamics, and Igarashi et al. [165], with a pipeline for HEVC (video) encoding.

If optimizing the application as a whole is the goal, one of the first techniques that manifest itself is *Host Communication* (Section 6.4.1), see Table 1. It is not uncommon that GPU kernels execute so fast that the bottleneck at the application level is the communication between the host and device. Using pinned memory in combination with proper pipelining and double buffering can help in this respect. The *Compress Data* technique (Section 6.1.10) is also relevant because compression is often applied to minimize data transfers between the host and GPU.

Another important optimization technique is *CPU/GPU Computation* (Section 6.4.2) that discusses ways to divide the work between the CPU and GPU. The *Precompute* optimization that typically precomputes values on the CPU for use within the kernels may also be relevant here.

Other relevant optimization techniques are *Kernel Fusion* (Section 6.1.8) with which kernels that operate on the same data and can be fused into one can result in significant performance benefits. Related to kernel fusion is *Inter-Block Synchronization* (Section 6.3.10), which allows kernels that

Table 1. Optimization Techniques for GPU Applications as a Whole

Optimization	Section	Explanation
Host communication	6.4.1	Often bottleneck in pipelines
Compress Data	6.1.10	Reduce size of data transfers
CPU/GPU Computation	6.4.2	Improve division work between CPU and GPU
Kernel Fusion	6.1.8	Reuse data among kernels
Inter-Block Synchronization	6.3.10	Means to apply kernel fusion
Software Prefetching	6.1.9	Prefetch data before barrier for after barrier
Kernel Fission	6.2.4	Break down large, difficult to optimize kernels

Table 2. Optimization Techniques for Compute-bound Kernels

Optimization	Section	Explanation
Reduce redundant work	6.2.5	Make better use of the memory
Loop Unrolling	6.2.1	Reduce redundant operations
Varying work per thread	6.3.4	Make better use of computational units
Resize thread blocks	6.3.5	Make better use of computational units
Vectorization	6.3.1	Make use of efficient vector instructions
Auto-tuning	6.3.6	Tune the above parameters
Reduce atomics	6.3.9	Reduce expensive operations
Fast Math Functions	6.3.2	Replace expensive math operations with cheaper ones

require global synchronization to be fused together. If this is possible, the optimization *Software Prefetching* becomes relevant as global synchronization makes it possible to prefetch data for the kernel after the inter-block synchronization. The opposite of kernel fusion, *Kernel Fission* (Section 6.2.4) may benefit applications with large monolithic GPU kernels that are difficult to optimize. The rest of the optimization techniques are applicable at the kernel level.

7.1.2 Compute or Memory Bound. Whether a kernel is compute or memory bound determines to a large degree what kind of optimizations to apply. The *Roofline model* is an often applied performance model that gives insight into whether a kernel is memory-bound [394]. It defines *operational intensity*, broadly the ratio between memory operations and computational operations.

Vector addition, is a typical example of a *memory bound* kernel with a low operational intensity because, for each addition, two reads and one write are performed. A typical example of a *compute bound* kernel is a Fast Fourier Transform with many floating-point computations per data element.

A compute-bound kernel is often preferred over a memory-bound kernel and the options for compute-bound kernels are less than for memory-bound kernels. It may be the case that improving the efficiency of computations, a kernel becomes memory bound again. For an overview of the applicable optimizations, see Table 2. The first optimization is *reducing redundant work*. It may be the case that values that are computed can be stored and used later. This can also be indexing operations that often can be simplified with *loop unrolling*. With *loop unrolling*, *varying the work per thread*, *resizing thread blocks*, *vectorization*, *loop unrolling*, and *auto-tuning*, it may be possible to find a good balance that utilizes the functional units to their maximum capacity. The *reduce atomics* technique can give rise to reduce or avoid expensive atomic operations, therefore increasing the computational throughput. Finally, applying *fast math functions* gives up some precision in favor of more computational load.

There are plenty of opportunities to exploit the GPU's memory hierarchy for memory-bound kernels. Table 3 shows the applicable optimizations. By *using dedicated memories* and applying

Table 3. Optimization Techniques for Memory-bound Kernels

Optimization	Section	Explanation
Use dedicated memories	6.1.1	Make better use of the memory hierarchy
Coalesced memory access	6.1.6	Improve bandwidth loads
Spatial Blocking	6.1.7	Improve locality of memory accesses
Register Blocking	6.1.3	Improve temporal locality of memory access
Kernel Fusion	6.1.8	Reduce loads/stores among kernels
Software Prefetching	6.1.9	Balance memory loads within the kernel
Use Warp Function	6.1.2	Reduce shared memory usage
Warp-centric Programming	6.3.3	Hide latency of memory operations
Reduce Synchronization	6.3.8	Reduce shared memory barriers
Varying work per thread	6.3.4	Increase parallel memory requests
Resize thread blocks	6.3.5	Allow for ILP
Vectorization	6.3.1	Make use of wide load instructions
Auto-tuning	6.3.6	Tune the above parameters

Table 4. Optimization Techniques for Kernels with Data Reuse

Optimization	Section	Explanation
Use dedicated memories	6.1.1	Use dedicated memories for data reuse
Spatial Blocking	6.1.7	Improve locality and therefore opportunity to reuse data
Register Blocking	6.1.3	Improve temporal locality and therefore opportunity to reuse data
Kernel Fusion	6.1.8	Reuse data shared between kernels
Inter-block Synchronization	6.3.10	Means to apply kernel fusion
Use Warp Function	6.1.2	Exchange reused data within warps
Warp-centric Programming	6.3.3	Hide latency of memory operations
Varying work per thread	6.3.4	Reuse data within a thread
Resize thread blocks	6.3.5	Allow for more shared memory
Auto-tuning	6.3.6	Tune the above parameters

coalesced access it is possible to improve the memory throughput. *Spatial blocking* and *register blocking* can help improve locality and therefore reduce memory pressure. *Kernel fusion* (and *inter-block synchronization*) can bring out data that is reused between the fused kernels. With *software prefetching* programmers can gain more control over when to load specific data, although it requires storage space to use it at a later time. By *using warp functions*, pressure on shared memory can be reduced and it may be necessary to adopt a *warp-centric programming style* to achieve this. Memory limitations can also manifest themselves as stalls because of barriers on shared memory, so the *reducing synchronization* techniques can improve on this. In some cases, it may be possible to trade memory space for computation time by *recomputing* values that had already been computed before and are required later in the kernel. *Loop unrolling*, *varying work per thread*, *resizing thread blocks*, *vectorization*, and *auto-tuning* can all contribute to improve instruction-level parallelism that allows more memory requests.

7.1.3 Data Reuse or no Data Reuse. This part is highly related to whether a kernel is compute or memory-bound. A kernel that can exploit *data reuse* is sensitive to other techniques (see Table 4) than kernels without data reuse (see Table 5). A typical kernel *without data reuse* is vector addition, each element is used once to compute part of the output. A typical kernel *with data reuse* is matrix multiplication. Kernels that do not exploit data reuse in matrix multiplication are likely to be memory bound because they need to load data multiple times for different output elements. However, if data reuse is exploited in the kernels, the matrix multiplication kernel can become compute bound.

Table 5. Optimization Techniques for Kernels without Data Reuse

Optimization	Section	Explanation
Resize thread blocks	6.3.5	Allow more thread blocks per SM
Reduce register usage	6.1.4	Allow more thread blocks per SM
Loop Unrolling	6.2.1	Create more independent instructions
Auto-tuning	6.3.6	Tune the above parameters

Table 6. Optimization Techniques for Irregular Kernels

Optimization	Section	Explanation
Compress data	6.1.10	Reduce size of indices
Reduce branch divergence	6.2.2	Remove serialization of branches
Sparse matrix formats	6.2.3	Increase regularity
Kernel fission	6.2.4	Split into simple kernels
Reduce redundant work	6.2.5	Maximize computation on loaded data
Load balancing	6.3.7	Divide the irregular work load equally among compute units

Obviously, GPUs' *dedicated memories* can be used to store data for reuse. *Spatial blocking* and *register blocking* allow organizing the code such that reuse can be exploited more. *Kernel fusion* (and *inter-block synchronization*) helps improving the opportunity of data reuse that exists between two separate kernels. With *warp functions* data that is reused between threads can be shared within warps and it may be necessary to adopt a *warp-centric* programming style. *Varying work per thread* may improve the reuse of data that is shared among threads and *Resizing thread blocks* can be used to tune the amount of shared memory available to the thread block. These two last values can be *auto-tuned* for finding the right balance.

For kernels without data reuse, there are fewer possibilities. Since there is no data reuse and a data element is only used once for a computation, this means that the kernel is likely to be memory bound, unless many computations are carried out based on the same data element. In the latter case, the optimizations for compute-bound kernels are valid, otherwise, the following optimization techniques are applicable. Note that many of the optimization techniques are the same as for kernels *with* data reuse. However, they are valid for different reasons, because all of the optimizations are used to increase parallelism to overcome bandwidth limitations. With *resizing thread blocks* and *reducing register usage* it may be possible to have more TLP (see Section 5.4) by allowing more thread blocks. *Loop Unrolling* and *varying work per thread* allows kernels to have more independent instructions, therefore increasing ILP. With *vectorization* it is possible to improve the achieved memory bandwidth. *Auto-tuning* allows to tune these parameters for a good balance.

7.1.4 Regular or Irregular. It is difficult to map irregular algorithms to the highly regular architectures of GPUs. Irregular memory access is particularly expensive on GPUs and this makes kernels almost always memory bound, so the memory-bound optimization techniques are often relevant as well. Besides these techniques, there are various other techniques that stand out, see Table 6.

A typical example of an *irregular kernel* is sparse matrix-vector multiplication. In the de-facto standard CSR format, it requires unpredictable random-memory access and the performance not only relies on the specific kernel but also on the input data, for example on the sparsity of the input data. Contrast this to the *regular kernel* dense matrix multiplication in which memory access is predictable and the performance does not rely on the input-data.

Compressing data is often used for compressing the indices that refer to non-zero elements in sparse matrices or graph algorithms. Irregular kernels are often also irregular in branching, leading to warps where many of the threads are disabled in a warp. *Reducing branch divergence* discusses many techniques to improve these inefficiencies. *Sparse Formats* generally aim at making a format such that kernels become more regular, often aiming for reduced branch divergence and coalesced access. *Kernel fission* is a technique to break up complex kernels into kernels that are simpler or easier to simplify. Since irregular memory accesses are expensive, it is often beneficial to compute as much as possible with the data that is available, potentially *reducing redundant work*. A common problem for irregular workloads is that they are not balanced among the computational units. *Load balancing* discusses various techniques on different levels of parallelism.

7.2 Based on Bottlenecks

In this analysis we take the perspective of bottlenecks. A performance bottleneck is an aspect of the kernel or application or an architectural bound that directly limits the performance. An example of a bottleneck is *global memory bandwidth*: many articles identify that a kernel is limited by global memory bandwidth and in that case, if global memory bandwidth would be higher, the performance of their kernel would increase.

From the articles in our selection that clearly identify bottlenecks, we categorized these bottlenecks and show a selection together with the number of articles that mentioned these. The long tail of bottlenecks that were mentioned only once or twice were omitted. We divide the bottlenecks over the themes *Memory Access*, *Irregularity*, *Balancing*, and *Host Interaction* and relate those bottlenecks to the optimization techniques in the subsections below.

7.2.1 Memory Access. Table 7 shows the bottlenecks for memory access extracted from articles. The two most identified bottlenecks are *global memory bandwidth* and *latency*. The optimization techniques described in *use dedicated memories* (Section 6.1.1), *coalesced access* (Section 6.1.6) and *kernel fusion* (Section 6.1.8) are relevant here. This is also the case for the bottlenecks *uncoalesced access* and *irregular access* although more techniques are relevant for the latter one, see Section 7.2.2 below. *Spatial blocking* (Section 6.1.7) and *register blocking* (Section 6.1.3) are also relevant for bandwidth or latency-limited kernels with the goal to extract as much as possible computation from what has been loaded into memory. *Vectorization* (Section 6.3.1) may improve memory bandwidth if wide load instructions are used. *Software prefetching* (Section 6.1.9) can help overcome long latency operations. If irregular access is the cause of the bandwidth limitations and latency, then the techniques for *sparse matrix formats* (Section 6.2.3) are also very relevant.

If shared memory usage forms the bottleneck, either *bank conflicts*, its *capacity*, *bandwidth*, or *latency*, the techniques described in *use dedicated memories* (Section 6.1.1), *spatial blocking* (Section 6.1.7), *use warp functions* (Section 6.1.2), and *register blocking* (Section 6.1.3) are all relevant. *Software prefetching* (Section 6.1.9) is a technique that can help with the capacity and latency problems and *compress data* (Section 6.1.10) can mitigate capacity and bandwidth problems. The *precompute* (Section 6.1.11) and *recompute* (Section 6.1.5) techniques can reduce additional shared memory pressure. *Varying work per thread* (Section 6.3.4) and *resize thread blocks* (Section 6.3.5) can help to organize the data in shared memory better and we will see below (Section 7.2.3) that the shared memory capacity plays an important role in balancing parallelism as well. In general, adopting a *warp centric approach* (Section 6.3.3) can also help with shared memory issues.

Register file capacity and *spilling* are closely related and the relevant techniques for these bottlenecks are *use dedicated memories* (Section 6.1.1), *spatial blocking* (Section 6.1.7), *register blocking* (Section 6.1.3), and *reduce register usage* (Section 6.1.4). The techniques *precompute* (Section 6.1.11) and *recompute* (Section 6.1.5) are also relevant as this can free up registers that hold data for a

Table 7. Bottlenecks Related to Memory Access

Bottleneck	Nr. of articles	Articles
global memory bandwidth	66	[11, 14, 16, 29, 44, 49, 54, 62, 71, 80, 87, 88, 96, 99, 101, 104, 105, 110, 120, 125, 129, 139, 148, 152, 159, 168, 173, 187, 188, 194, 196, 200, 210, 219, 226, 235, 242, 251, 262, 263, 272, 310, 315, 318–321, 326, 328, 345–347, 375, 381, 398, 399, 402, 407, 409, 418, 420, 421, 423, 429, 432, 433]
global memory latency	24	[14, 44, 49, 87, 92, 139, 152, 156, 168, 169, 176, 200, 201, 234, 263, 315, 318, 326, 348, 363, 375, 402, 417, 418]
uncoalesced access	22	[33, 38, 45, 80, 89, 92, 96, 105, 123, 139, 148, 156, 170, 183, 201, 242, 256, 305, 324, 334, 408, 423]
irregular access	19	[14, 16, 28, 33, 37, 40, 82, 89, 92, 108, 160, 170, 223, 256, 272, 329, 370, 420, 424]
bank conflicts	14	[49, 80, 87, 94, 123, 124, 201, 256, 305, 326, 343, 354, 407, 408]
shared memory capacity	14	[7, 37, 80, 94, 104, 110, 194, 210, 299, 310, 318, 370, 379, 402]
shared memory bandwidth	10	[7, 37, 49, 148, 194, 251, 316, 375, 381, 407]
shared memory latency	3	[348, 363, 375]
register file capacity	16	[23, 49, 80, 104, 169, 176, 194, 200, 210, 271, 287, 299, 318, 348, 379, 408]
register spilling	9	[23, 87, 120, 169, 194, 216, 251, 326, 348]
cache conflicts	2	[120, 320]
cache misses	2	[33, 272]
cache capacity	2	[31, 33]
cache bandwidth	1	[407]
cache latency	1	[318]

long time. The same is true for *loop unrolling* (Section 6.2.1)—since this is often a large contributor to register spilling—and *kernel fission* (Section 6.2.4) that can simplify kernels and therefore improve their register usage. Finally, *varying work per thread* (Section 6.3.4) and *resize thread blocks* (Section 6.3.5) can have a large influence on register usage. These techniques are from the theme balancing parallelism but register usage plays an important role for this aspect as we will see below in Section 7.2.3.

Cache usage can also be a bottleneck as some articles report, either *conflicts*, *misses*, its *latency*, *bandwidth*, or *capacity*. Relevant techniques here are *use dedicated memories* (Section 6.1.1), *spatial blocking* (Section 6.1.7), *register blocking* (Section 6.1.3), *kernel fusion* (Section 6.1.8), and for bandwidth and capacity *compress data* (Section 6.1.10) can be relevant.

7.2.2 Irregularity. Most of the articles that experience a bottleneck related to irregularity name *branch divergence* and the overhead of *branch instructions* (Table 8). *Load imbalance* that we categorize in the *Balancing* section below is also explicitly named as a bottleneck and is highly comparable to branch divergence on the thread and warp level. The relevant optimization techniques for these bottlenecks are *reduce branch divergence* (Section 6.2.2) and *sparse formats* (Section 6.2.3) that discusses many techniques to handle irregular applications. Techniques discussed in *loop unrolling* (Section 6.2.1) and *kernel fission* (Section 6.2.4) can improve branch divergence as well and can reduce branching instructions in general. *Warp-centric programming* (Section 6.3.3) can also improve branch divergence. Few articles mention *redundant computation* as bottleneck and the techniques in *reduce redundant work* (Section 6.2.5) and *precompute* (Section 6.1.11) are relevant here.

Table 8. Bottlenecks Related to Irregularity

Bottleneck	Nr. of articles	Articles
branch divergence	21	[24, 38, 49, 80, 82, 87, 112, 139, 148, 156, 183, 242, 256, 262, 305, 313, 324, 334, 379, 402, 439]
branch instructions	4	[23, 63, 102, 105]
redundant computation	2	[23, 34]

Table 9. Bottlenecks Related to Balancing Parallelism

Bottleneck	Nr. of articles	Articles
atomic contention	16	[37, 42, 82, 92, 123, 124, 194, 203, 219, 274–276, 298, 333, 343, 370]
load imbalance	16	[28, 34, 35, 82, 183, 202, 212, 242, 256, 275, 281, 298, 377, 409, 429, 432]
synchronization	16	[14, 23, 40, 48, 94, 102, 132, 194, 203, 204, 210, 256, 305, 345, 399, 429]
utilization	13	[84, 87, 176, 200, 201, 226, 251, 299, 305, 318, 321, 402, 429]
instruction issue throughput	5	[49, 168, 318, 326, 375]
instruction issue latency	5	[49, 210, 320, 348, 402]
finite field arithmetic	3	[47, 49, 118]
arithmetic throughput	2	[235, 318]
SFU throughput	2	[23, 169]
pipeline latency	1	[375]
SFU latency	1	[152]
integer throughput	1	[120]

7.2.3 Balancing. In this theme, *atomic contention* and, more general, *synchronization* are an often named bottleneck as Table 9 shows. Relevant techniques for *atomic contention* are *reduce atomics* (Section 6.3.9), *use dedicated memories* (Section 6.1.1), and *use warp functions* (Section 6.1.2), the latter two using shared memory or warp functions for reductions, respectively. A *warp-centric approach* (Section 6.3.3) may also be relevant for these bottlenecks.

For *synchronization* in general, *reduce synchronization* (Section 6.3.8), *inter-block synchronization* (Section 6.3.10), and the interplay with *kernel fusion* (Section 6.1.8) are relevant. The techniques described for *using dedicated memories* (Section 6.1.1) and especially the application of barriers in shared memory are useful as well.

If *load imbalance* is a bottleneck, the techniques in *load balancing* (Section 6.3.7) are relevant. As said before, on the warp level load imbalance issues are highly related to *reduce branch divergence* (Section 6.2.2) and the techniques in *warp-centric programming* (Section 6.3.3) can help as well with these issues. On a higher level, the techniques for *CPU/GPU computation* (Section 6.4.2) may be relevant also.

An important category of bottlenecks is *utilization* which covers how well the parallel resources of GPUs are utilized, such as threads, thread blocks, special function units, and so on. Recall that the number of thread blocks executing concurrently on an SM depends on the thread block size, the number of registers per thread, and the amount of shared memory allocated to each thread block. Therefore, the techniques for the bottlenecks *shared memory capacity* and *register file capacity* are also highly relevant and the techniques of *varying work per thread* (Section 6.3.4) and *resize thread blocks* (Section 6.3.5) are especially important to find a balance that improves the performance.

However, these techniques only focus on TLP, whereas ILP can be as relevant, and sometimes even necessary to achieve high performance [373]. The amount of instruction-level parallelism depends on the number of independent instructions, but also on the instruction latency, and read-after-write delays on registers. Table 9 lists several articles that specifically mention the latency

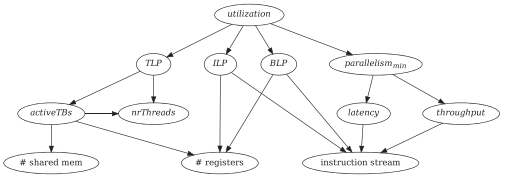


Fig. 8. Dependencies between the terms in *utilization*.

Table 10. Bottlenecks Related to Host Interaction

Bottleneck	Nr. of articles	Articles
PCIe bandwidth	34	[29, 34, 46, 62, 66, 76, 102, 104, 106, 121, 128, 135, 168, 179, 205, 210, 253, 254, 265, 271, 299, 302, 308, 316, 326, 347, 363, 375, 377, 379, 397, 413, 419, 434]
global memory capacity	10	[14, 34, 131, 160, 200, 225, 235, 318, 408, 437]
kernel launch throughput	4	[92, 210, 298, 375]

or throughput of instructions in the instruction stream that form the bottleneck, such as integer instructions throughput [120], and SFU throughput [23, 169], and latency [152].

Figure 8 shows the dependencies between all the factors that influence the utilization based on Volkov's performance model that is detailed in Appendix D. For example, if the conclusion is that more ILP or TLP is required to improve utilization, the optimizations in turn may affect the instruction stream itself, the number of registers, and possibly the amount of shared memory. Especially the latter two can have a dramatic effect on the number of active thread blocks. In other words, utilizing the resources of a GPU well is truly a balancing act, and reasoning about utilization is challenging because of the strong relationship between the various factors. A good and widely applied technique is *auto-tuning* (Section 6.3.6) to find the right balance for high utilization.

7.2.4 Host Interaction. The most important bottleneck revolving around host interaction is the *PCI-express bandwidth*, see Table 10. The techniques in *host communication* (Section 6.4.1) are naturally relevant, but dividing the *computation over the CPU or GPU* may also help (Section 6.4.2). *Kernel fusion* (Section 6.1.8) can help reusing data that has been transferred over the slow bus, but equally *kernel fission* (Section 6.2.4) can help separating kernels such that transfers can be overlapped. A final relevant optimization technique is *compress data* (Section 6.1.10) in an attempt to send less data over the PCIe bus and use undersubscribed computational resources to decompress the data.

Another mentioned bottleneck is *global memory capacity*. The same optimization techniques for the PCI-express bottleneck are valid here, such as compressing data and creating streaming versions of the kernels. In addition, the techniques described in *recompute data* (Section 6.1.5) can be relevant.

Finally, the bottleneck *kernel launch throughput* can be alleviated by means of *kernel fusion* (Section 6.1.8), reducing the number of kernel launches with larger kernels or enlarging the work of a kernel with *varying work per thread* (Section 6.3.4).

7.3 Impact of Architectural Evolution on GPU Optimizations

This section gives a brief overview of the major GPU architectural evolutions and their influence on optimizations for GPU programming. Figure 9 shows a timeline of major GPU architecture and programming model releases.

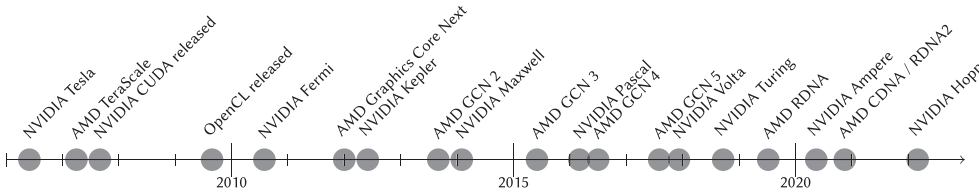


Fig. 9. Timeline of major NVIDIA and AMD architectures and GPU programming models.

In November 2006, NVIDIA launched the GeForce GTX 8800 GPU of the Tesla architecture [221], which was the first GPU with unified shaders. This meant the unified shader processors could be allocated to perform vertex, fragment, and geometry shader workloads. This paved the road for programming these cores using CUDA as the first GPU Programming model for workloads outside of graphics. Some authors have used the graphics APIs to use GPUs for scientific applications on these GPUs predating CUDA [307], but these approaches are out of scope for this survey. In 2007, NVIDIA released the CUDA programming model. In the same year, AMD (ATI at the time) released the Radeon HD 2000 GPU series of the TeraScale 1 architecture that followed the unified shader approach taken by NVIDIA. Both vendors started supporting the OpenCL programming model with the release of the Radeon R700 and Geforce 9 series.

In the early days of GPU programming, memory optimizations such as coalescing and using dedicated memories were extremely important and could improve performance by orders of magnitude [325, 414]. With the release of the Fermi [288] architecture, NVIDIA introduced an L1/L2 cache hierarchy in GPUs, which significantly improved the performance of uncoalesced memory accesses [95, 391], as all memory transactions are executed at the granularity of cache lines. Using dedicated memories was still important, but the difference in performance has become less dramatic over the years, and its effect is sometimes hard to notice in modern architectures.

NVIDIA's Kepler [289] and AMD's GCN architectures appeared in 2012. GCN moved away from the VLIW architecture of TeraScale in favor of a RISC SIMD architecture and introduced support for unified virtual memory. In contrast to Fermi, Kepler GPUs did not cache global memory accesses in L1 by default, which made Kepler GPUs are more sensitive to irregular memory accesses [274] and increased the importance of shared memory. Kepler GPUs used a default shared memory bank width of 64 bits, which meant it required extra work to optimize shared memory bandwidth for 32-bit data [390]. Kepler GPUs also introduced warp shuffle instructions and improved the programmability of the texture cache by allowing it to be used as a read-only cache. The NVIDIA K20 introduced a second copy engine, which improved the effectiveness of using streams to overlap CPU-GPU data transfers [392]. As of CUDA 6, Kepler GPUs and newer support Unified Memory, also called *managed memory*, as an alternative way to ease programmability of CPU-GPU data movements [142] (see Section A.4.1).

The Maxwell architecture [290] features a dedicated shared memory unit and the L1 cache is shared with the texture cache. This corresponds to the observation that for some applications texture memory reads are no longer beneficial on Maxwell and newer architectures, while these used to be effective on Kepler GPUs [317]. In addition, the Maxwell SM can schedule multiple instructions every cycle allowing special function units and CUDA cores to perform operations in parallel [368].

The Volta architecture [292] introduced Tensor Cores for performing mixed-precision matrix arithmetic. Three years later, AMD followed this approach by including MCEs in their MI100 GPU [22]. Another important change in the Volta architecture is the introduction of independent thread scheduling, in contrast to the SIMT model used by Pascal [291] GPUs and earlier, improving

the performance of applications that suffer from divergence on older GPUs. Anzt et al. [28] report however that for their sparse matrix applications using of any of the sub-warp synchronization features introduced in Volta did not improve performance. The introduction of HBM2 memory over the off-chip GDDR memory significantly improved device memory bandwidth. Tang et al. [353] state that using shared memory became less important to their application on GPUs with HBM.

AMD's GCN 5 architecture [20] added support for HBM2 and introduced native support for half-precision arithmetic. Reis et al. [312] compare the impact of several optimizations on AMD GPUs of GCN2 versus GCN 5 for half-precision applications concluding that applying vectorization is only important on GPUs with native support for half-precision.

The Turing architecture [293] unifies shared memory, texture cache, and L1 cache into a single unit, effectively doubling the size and bandwidth of L1 in workloads that do not use dedicated memories. This again reduces the performance benefits of *using dedicated memories*. The configurable size of shared memory is also limited to 64 KB in Turing, and over 96 KB in Volta [408]. The reduced shared memory size on Turing also limits the effectiveness of certain optimizations, such as avoiding shared memory bank conflicts [354]. Yan et al. [408] note that it is easier to avoid register bank conflicts on Volta and Turing, due to the increase in width from 32 to 64 bits over earlier architectures.

RDNA is the first architecture by AMD that uses a warp (wavefront) length of 32, motivated by having less divergence and finer granularity in control logic, registers, and cache [21]. The later CDNA architecture moved back to wavefronts of length 64, however [22].

NVIDIA's Ampere architecture [295] adds many new innovations such as multi-instance GPU (MIG), asynchronous operations on shared memory, and asynchronous barriers that allow to reduce synchronization for warp or group centric programming. In early 2022, NVIDIA has announced the Hopper architecture [25] as the successor to Ampere.

7.4 Quantitative Overview Performance Potential

In this section, we aim at giving a quantitative overview of the performance potential for each of the reviewed techniques. While a full quantitative analysis would be very valuable, it would be a near impossible task to give a definite answer to the performance potential of each technique because performance relies on a context in which many factors cannot be tested in isolation, such as GPU architecture, kind of application, datasets for irregular applications, and other contributing optimizations.

However, to give an impression of the performance potential, we have collected for each technique a representative article that lists the performance improvement in isolation to other optimizations. Table 11 lists the optimization techniques together with the article that discusses the performance in isolation, the speedup that is reported, the architecture that was used to measure, and notes about the context that are relevant to interpret the speedup. Note that the speedup numbers are not comparable between optimizations because the context is highly relevant. However, this table may provide an indication of the potential and range of individual optimizations if the performance numbers are interpreted in context.

8 DISCUSSION AND CONCLUSION

Our extensive study of 450 articles resulted in a wide variety of optimization techniques for GPUs that we divided into the four themes *memory access*, *irregularity*, *balancing parallelism*, and *host interaction*. From our detailed descriptions, especially the breadth of the different techniques stands out, in addition to the many ways the optimization goals can be achieved.

Our analysis considers the optimizations from different perspectives, first based on application-specifics such as compute- or memory-bound kernels or for kernels with or without data reuse,

Table 11. Indication of Potential Performance Benefits for the Various Optimizations

Optimization	Section	Reference	Speedup	Architecture	Context
Use shared memory	6.1.1	[325]	2 orders of mag.	NVIDIA Tesla (2008)	sorting
Use Warp Functions	6.1.2	[381]	1.2–2x	NVIDIA Kepler (2012)	sequence alignment
Register Blocking	6.1.3	[353]	2–7x	NVIDIA Volta (2017)	graph comparison
Reduce Register Usage	6.1.4	[358]	7–9%	NVIDIA Kepler (2012)	Lattice Boltzmann sim.
Recompute	6.1.5	[149]	up to 2.5x	NVIDIA Tesla (2008)[1]	stencil
Coalesced Access	6.1.6	[414]	1.8–4x	NVIDIA Tesla (2008)[1]	avg. over benchmarks
Spatial Blocking	6.1.7	[337]	3–6x	NVIDIA Tesla (2008)	regular applications
Kernel Fusion	6.1.8	[224]	10–17%	NVIDIA Kepler (2012)	graph algorithms
Prefetching	6.1.9	[27]	20–40%	NVIDIA Kepler (2012)	sparse matrix multiplication
Compress Data	6.1.10	[226]	1.27–5x	NVIDIA Pascal (2016)[1]	non-det. finite automata
Precompute	6.1.11	[23]	49%	AMD Tahiti (2012)	password recovery
Loop Unrolling	6.2.1	[318]	up to 25%	NVIDIA Tesla (2008)	matrix multiplication
Reduce Branch Divergence	6.2.2	[379]	3.45x	AMD RV770 XT (2008)	stencil, branch intensive
Sparse Matrix Formats	6.2.3	[266]	1.1–40x	NVIDIA Kepler (2012)	avg. over benchmarks
Kernel Fission	6.2.4	[80]	70%	AMD Cypress (2009)	molecular modeling
Reduce Redundant Work	6.2.5	[226]	1.7–6x	NVIDIA Pascal (2016)[1]	non-det. finite automata
Vectorization	6.3.1	[416]	2.3–6.1x	AMD Cypress (2009)[1]	avg. across various benchmarks
Fast Math Functions	6.3.2	[64]	3.3–4.8x	NVIDIA Fermi (2010)	Traveling Salesman Problem
Warp-centric Programming	6.3.3	[154]	1.04–8x	NVIDIA Tesla (2008)	Breadth-first Search
Varying Work per Thread	6.3.4	[244]	2–12x	AMD Tahiti (2011)[1]	var. benchmarks
Resize Thread Blocks	6.3.5	[205]	30x	NVIDIA Tesla (2008)	struc. grid CFD solver
Auto-tuning	6.3.6	[244]	12x	AMD Tahiti (2009)	matrix multiplication
Load Balancing	6.3.7	[202]	40%–60%	NVIDIA Turing (2018)[1]	avg. over graph algorithms
Reduce Synchronization	6.3.8	[274]	3%–50%	NVIDIA Kepler (2012)[1]	graph algorithms
Reduce Atomics	6.3.9	[115]	35%	NVIDIA Pascal (2016)	avg. over BFS datasets
Inter-Block Synchronization	6.3.10	[400]	11%–60%	NVIDIA Tesla (2008)	various benchmarks
Host communication	6.4.1	[95]	18.9%	NVIDIA Fermi (2010)[1]	meshfree comp. fluid dynamics
CPU/GPU Computation	6.4.2	[72]	15%–20%	NVIDIA Fermi (2010)	k-means

[1]Other architectures are also featured in this article.

then based on performance bottlenecks. The analysis shows that many different techniques are relevant for a specific application characteristic or bottleneck. In other words, the techniques and their effects are highly interrelated. This is once more confirmed with Volkov's simple but highly insightful performance model for utilization that makes explicit to what extent the various factors are dependent on each other to achieve high utilization, explaining and confirming the wide use of auto-tuning for GPUs.

In this work, we looked back at 14 years of performance optimization and aims at enabling GPU programmers to learn about various applied optimization techniques, give researchers in the domain of compilers and programming language insight into the optimization challenges that programmers face, and inform architecture researchers and hardware manufacturers how programmers attempt to extract performance from GPUs. Our data analysis shows that GPU optimization remains relevant and therefore it is important to have a good understanding of the applied optimization techniques while GPU hardware, applications, and programming systems are rapidly diversifying with new fields such as IoT, autonomous vehicles, and exascale computing.

APPENDICES

A OPTIMIZATION TECHNIQUES

This appendix contains the optimizations techniques extracted from the literature. The optimization techniques are grouped and organized in the same themes as in Section 6 and form reference material for those who are interested in more details. Below is an overview of the optimizations:

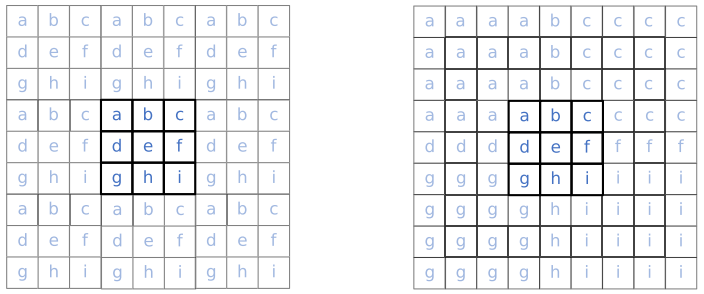
A.1 Memory Access

A.1.1 On-Chip — Use Dedicated Memories. For most applications GPUs have more computational resources than the memory system can provide. Therefore, many optimizations center around improving memory access. Similar to CPUs, GPUs provide caches, albeit with less levels than CPUs (two levels compared to typically three levels on a CPU). However, GPUs also provide dedicated memory spaces that are mapped into fast memory: constant memory, texture or surface memory, and shared memory.

Constant memory. A typical maximum size for constant memory is 64 KiB. Constant memory is read-only data during kernel execution and it has to be written before kernel launch and declared to be constant memory. Constant memory is especially useful for data that is broadcasted to multiple threads since it only results in one memory transaction [107, 200]. It is also useful for data that needs to be available for a sequence of kernel calls [59]. Considerations for using constant cache are cache conflicts [319] and Lee et al. measure the constant cache hit ratio [201].

Jang et al.[170] measure a speedup of 3 to 5 times by using constant memory on a Tesla GPU. Xiao et al. [401] report a speedup of 12% by using constant memory for storing the query sequence and scoring matrix in constant memory in the context of a gene-sequencing application on the Fermi architecture.

Texture memory. Texture memory is also a read-only memory with typical sizes of 12 KiB to 128 KiB. Although texture memory is often used solely for the fact that accesses are cached [46, 78, 230, 247, 339], texture memory is optimized for accesses with the spatial locality in two dimensions and several implementations exploit this spatial locality [203, 264, 349, 364, 401, 418] or use it for uncoalesced access [53, 435]. In addition, it is also possible to exploit the fact that texture fetches



(a) The "wrap" addressing mode. (b) The "clamp" addressing mode.

Fig. 10. A 2-dimensional 9-element texture with different addressing modes.

can automatically handle boundary values as shown in [193, 195, 206, 318, 319], possibly reducing branch divergence. Figure 10(a) shows a 2-dimensional 9 element texture with the "wrap" addressing mode meaning that an out-of-bound read will return a value as if the texture was wrapped, the transparent elements being the out-of-bound accesses. Figure 10(b) shows the same with the "clamp" addressing mode where the boundary element is repeated for out-of-bounds reads.

The fact that one texture fetch can retrieve an R, G, B, and A value inspired Liu et al. to use the same technique for packing string data in the same way and retrieving this data with one texture fetch [231] for gene sequence alignment. Texture fetches are performed with a special API and addressing is typically performed by dedicated hardware. Texture units can also automatically perform integer to float conversion (with normalization) which is very useful for graphics but can also be exploited in general-purpose applications [133, 234]. A typical performance improvement is reported by Ryoo et al. [318] where a kernel in the context of video compression showed a 2.8 times speedup on a Tesla GPU where boundary calculation is done by the texture units.

Shared memory. To use shared memory, one can declare a variable, most often an array, to be "shared". Shared arrays reside in the on-chip memory of a streaming multiprocessor and typical sizes of this on chip memory are 16, 48, 64, or 96 KiB. This data can be shared between threads within a thread block. Since this allows parallel access to the memory, synchronization is needed and this happens with barriers that each warp within a thread block must reach before execution continues.

Stalling warps as a result of barriers is often bad for performance. An often used strategy is to decrease the amount of shared memory within a thread block to allow multiple thread blocks to run on a single streaming multiprocessor [170]. Since thread blocks are independent, they do not need synchronization between thread blocks, so if one thread block stalls on a barrier, another thread block can continue execution.

On-chip memory is typically organized in banks equal to the size of the number of threads within a warp. Threads within a warp that access the same element are serialized decreasing performance, which is called "shared memory bank conflicts". An often used strategy is to reduce the amount of bank conflicts.

Shared memory can be used for a number of reasons. Foremost, shared memory is used to allow sharing of data between threads [84, 127, 158, 186, 280]. Another purpose is to decrease expensive memory operations of global memory [203, 275], for example, allowing global memory access to be coalesced and stored in shared memory to allow uncoalesced access there [60, 66, 116, 127, 129, 278, 315, 319, 329, 414] or to minimize the impact of irregular access altogether [64, 127, 203, 215, 325]. Satish et al. [325] report a speedup of 2 orders of magnitude faster by sorting data in shared memory

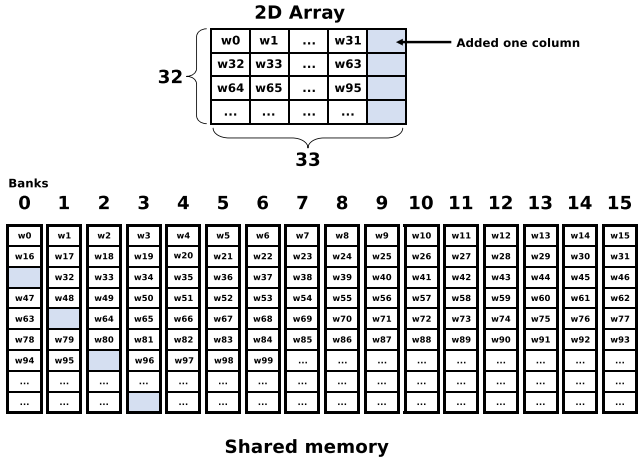


Fig. 11. Padding applied to resolve bank conflicts (from Man et al. [245], © IEEE).

replacing scattered writes to global memory with scattered writes to shared memory on the Tesla architecture (2008). Shared memory can also be used for reducing synchronization, for example by using a hash table per thread block that is later merged in a global hash table [27].

The effect of shared memory increases if data reuse or temporal locality can be exploited [5, 14, 79, 92, 149, 162, 187, 217, 263, 304, 363, 407, 408, 414] but for spatial locality shared memory is also useful [107, 162, 251, 257, 304, 310]. A specific pattern that is often performed by means of shared memory are reductions [9, 14, 46, 59, 97, 177, 264, 266, 311, 335].

Once the kernel has been transformed to use shared memory it may be important to optimize the usage of shared memory. Typical transformations are blocking that often is performed with the help of shared memory [162, 209, 249, 257, 304, 310, 315, 319, 345, 350, 353, 407, 408, 425] or fusion of kernels to increase the opportunity to reuse data [223] (see Section A.1.8).

Often shared, texture, and constant memory are regarded as the fastest memories for data reuse and this is true for the spatial locality, but for a temporal locality, registers can form an even faster memory. Volkov and Demmel investigate the relation between bandwidth, instruction level parallelism, and memory bandwidth and conclude that given enough instruction level parallelism, for example, achieved with aggressive loop unrolling, it may be beneficial to use a low number of threads to allow registers to be the primary on-chip memory instead of shared memory [375]. Swirydowicz et al. continue with this line of thought and create a Roofline model based on shared memory [345]. Typically, these implementations have low occupancy while achieving very high performance. To use registers in this way shared memory is often used to stage the data for the registers [278, 316] or to communicate data between registers via shared memory [127].

In contrast, for other implementations it is sometimes necessary to use the shared memory to reduce register spilling [52, 172]. In another case, registers are used as a spill-location for shared memory[49]. The amount of shared memory used is a limitation on the number of thread blocks per SM and often it is necessary to reduce shared memory to allow more independent threads to decrease barrier synchronization penalties [127, 170, 201, 303]. For example, this can be done with loop-fission [50], recomputing values [200], or using registers [49, 200].

Another important technique to improve shared memory performance is reducing the amount of bank conflicts. There are various techniques to do this. Often padding is applied to offset threads to access a subsequent bank [97, 107, 110, 123, 124, 127, 137, 200, 245, 246, 287, 304, 343, 345, 345, 371, 375, 376, 390, 398, 407, 408, 414]. Man et al. [245] show in Figure 11 where they pad a 32×32

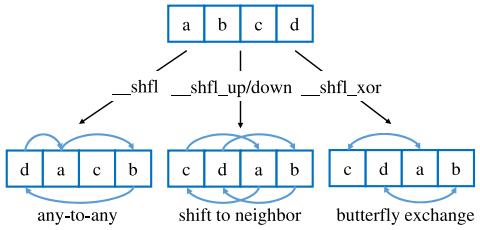


Fig. 12. Shuffle instruction variants (from Wang et al. [381], © IEEE).

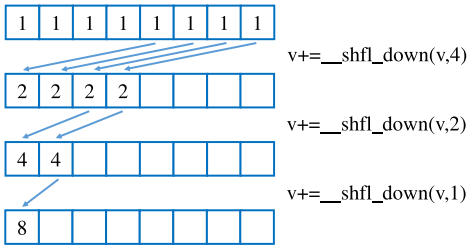


Fig. 13. A reduction implemented with shuffle instructions (from Wang et al. [381], © IEEE).

arrays with one extra element on each row to ensure that when all 32 threads of a threadblock access a column in the 32×32 arrays (element 0, 32, 64, and so on.) the threads do not access the same bank resulting in serialized access, but subsequent banks resulting in parallel access.

Reordering the data is a different technique [66, 209, 315, 318, 359, 361, 431] and Govindaraju et al. accomplish this by transforming from an array of structs to a struct of arrays [127]. It is also possible to keep the same layout but to remap threads to data [52, 134, 146, 182, 318, 412, 412] or, related, perform loop interchange [318]. Reddy et al. exploit the commutativity of operations [311] to reorder data access. Bernstein et al. wrote a search tool that assigns threads with data such that almost all bank conflicts are eliminated [49].

A.1.2 On-Chip — Use Warp Functions. In general and historically, GPUs do not support synchronization between thread blocks (see Section A.3.10 for a more detailed discussion) and support communication and synchronization between threads within a thread block via shared memory and barriers. However, NVIDIA introduced warp-voting functions in the Fermi architecture (2010) with which threads within a warp can reach consensus without the need for explicit synchronization. Starting with the Kepler architecture (2012), NVIDIA extended these *warp vote functions* with *warp shuffle functions* making warp-level functions more powerful. Exploiting these *warp functions* is considered an optimization because of the performance gains that can be achieved with them.

The shuffle functions allow threads in the same warp to exchange data that is stored in registers, without having to store it in shared memory first, and without explicit synchronization at thread-block level. In Figure 12, Wang et al. [381] show variants of the `shuffle` functions that were introduced in the Kepler architecture (2012). The advantages of using this optimization are multiple: first, the latency to access registers is lower than for shared memory, and bandwidth is higher; second, the amount of shared memory allocated is less; third, the communication overhead is lower because of the lack of explicit synchronization. Because of these advantages, the largest use-case for warp shuffle functions in the literature is to share data intra-warp without going through shared memory [7, 14, 47, 65, 70, 150, 155, 204, 210, 223, 233, 268, 328, 380, 411, 422].

Particularly interesting is the use that Barnat et al. [44] have for these functions, using them to reduce the amount of memory accesses to global memory and save memory bandwidth via a warp-level caching system. There have also been attempts at comparing warp shuffle functions to a more traditional intra-warp communication mechanism based on shared memory. Wang et al. [381] develop micro-benchmarks to compare the two approaches, and conclude that the use of this optimization provides tangible gains in performance, between 20% and a factor 2 for two different types of sequence alignment algorithms (Kepler 2012), while Mawson et al. [252] show that using warp shuffle functions for their Boltzmann solver is faster than using shared memory, albeit slightly. Although this is often the case, there are also cases where this optimization does not provide benefits, such as documented by Abdelrahman et al. [12] using their cryptographic micro-benchmarks.

Another specialized use-case for warp shuffle functions is to implement collaborative algorithms such as reduction [35, 42, 117, 266, 282, 311, 381] and prefix-sum [184, 185]. Wang et al. [381] show in Figure 13 how reduction can be implemented in terms of `shuffle` instructions. The main advantage provided by this optimization for these algorithms is the lower synchronization overhead.

The warp vote functions are used to evaluate predicates at warp-level, and then broadcast the result of such evaluation to all threads in the warp. In this way, developers can test if all, any, or which threads in a warp satisfy a given predicate without explicit communication between them [28, 37, 38, 115, 231, 298, 438]. To provide some examples of the use of warp vote functions, Pai et al. [298] use this optimization in graph processing, to determine which thread should execute a push operation in their algorithm, and Ashkiani et al. [37] use the same optimization to compute a warp-level histogram.

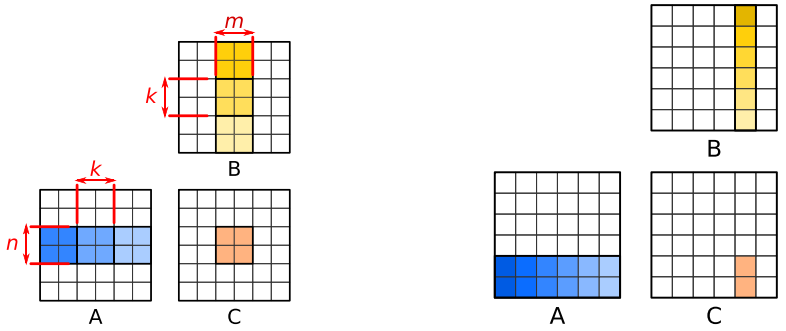
A.1.3 On-Chip — Register Blocking. Registers form the fastest memory available on GPUs [373] and as such, can be very effective for increasing performance by storing often-used values in them. Since registers are not shared among threads, registers are mostly effective for data reuse with temporal locality and less for spatial locality although some use warp-shuffle to distribute values among threads in a warp [7, 12]. Because of these restrictions, register blocking, sometimes also called temporal blocking, is highly related to loop unrolling, explained as "[r]egister blocking is, in essence, unroll and jam in [the] X, Y, or Z [dimension]" [83, p. 7]. It is related to "varying work per thread" (Section A.3.4) because sometimes the same data is reused for multiple elements that a thread computes. Hong et al. [152] indeed acknowledge that varying the work per thread can achieve register blocking. Often, register blocking is combined with spatial blocking (Section A.1.7) and is sometimes called "two-level blocking" [162, 407, 408].

From the literature, it appears that register blocking is prevalent in 3D stencil computations [83, 174, 257, 262, 302, 372, 379, 421, 433]. An often used approach is using shared memory for the XY plane and serialize the computation over the Z dimension, keeping data reused in the Z direction in registers [83, 174, 257, 302, 372, 421]. Mo and Li apply a similar technique but they reuse a partial sum of the neighboring points instead of elements in the Z direction [262]. Matsumura et al. provide a comprehensive overview of temporal blocking and its relation to spatial blocking and provide a framework to achieve high-degree temporal blocking for stencil computations [251].

Daga et al. note that registers are especially useful for accumulator values on GPUs since the values in registers cannot easily be shared with other threads [80], also observed by Hagedorn et al. [137]. This use is apparent in many articles that perform some form of matrix multiplication where the elements of the output matrix are accumulated in registers [6, 138, 162, 180, 278, 287, 314, 315, 353, 407] based on blocks of the input matrices that are potentially also blocked in registers [6, 138, 180, 278, 314]. Remmelg et al. [314] compare in Figure 14 register blocking (Figure 14(b)) with spatial-blocking (Figure 14(a), see Section A.1.7). At the left, the output elements are computed based on the tiles in the rows and columns of A and B respectively. At the right, the column in B is stored in registers and reused for each of the highlighted column in the row of A to accumulate the output elements that are also in registers.

Mukonoki et al. perform matrix/vector multiplication and make use of vector instructions for register blocking [267]. For the same application, Xu et al. carefully combine register blocking with coalesced memory access and also take into account reuse of the values that are loaded into the registers for other warps via the cache [404].

Since registers form the fastest memory available on GPUs, they can provide impressive speedup gains. Tang et al. [353] show speedups between a factor 2 and 7 depending on the block size in their graph comparison algorithm on the Volta architecture (2017).



(a) Matrix multiplication using spatial blocking. (b) Matrix multiplication using register blocking.

Fig. 14. Matrix multiplication blocking methods (from Remmelg et al. [314], © Remmelg et al.).

Tan et al. note that registers cannot be indexed and therefore techniques such as macro expansion or templates have to be used [349] to have control over arrays of registers [345]. Bernstein et al. use an interesting technique where registers replace the function of constant memory [49]. In addition, they pack four values in a register trading-off computations by means of bit shifts for faster memory access.

A.1.4 On-Chip — Reduce Register Usage. When compiling a GPU kernel, the compiler assigns each thread block a certain number of registers to hold intermediate values. Since the number of registers per SM is fixed, using too many registers can thus limit the number of concurrent thread blocks per SM (i.e., *occupancy*). In addition, increased register usage can result in register spilling, the phenomenon where thread local variables are stored in slow off-chip device memory, although the spilled variables are typically cached. Therefore, in certain scenarios reducing the register usage can reduce register spilling, which can lead to more concurrency, and thus performance benefits.

One obvious way to reduce register usage is to rewrite the kernel's source code. For example, Habich et al. [136] experimented with a 3D stencil application and found that each index operation into a multi-dimensional array consumed one register in CUDA 1.1. In later work [135], they were able to reduce 100 to 32 registers by manually writing these indexing operations using pointer arithmetic, minimizing temporary variables, and rewriting arithmetic operations. Tran et al. [358] investigate Lattice Boltzmann simulations and they are able to reduce register usage from 70 to 40 by using several techniques including moving certain temporary variables to shared memory, packing small types into larger types (i.e., pack 4 bytes into 1 integer), and refraining from storing values that can be recalculated. With these techniques they gain a performance gain between 7 and 9% on a Kepler GPU (2012). Unrolling of small loops can also save registers due to removing the need for induction variables [318, 331] (see Section A.2.1).

Another way to save registers is to change the algorithm. Qiu et al. [308] perform SHA1 calculations on the GPU; they use an algorithm that requires 80 precomputed values to be stored in registers and switching to a different compute scheme saves 64 registers. Fang et al. [104] switch from a multi-to-one scheme to a one-to-multi scheme for stencil operations and are able to reduce register usage by 40%.

Finally, it is also possible to force the compiler to use less registers. Khorasani et al. [183] use the `maxrregcount` option of the CUDA compiler to forcefully spill registers and find it to be beneficial for certain benchmarks. Bernstein et al. [49] take a more radical approach: being unsatisfied with NVIDIA's register allocator, they use a reverse-engineered CUDA assembler to perform their own

register allocation. However, it should be noted that this work was performed in 2010 and it is likely that the register allocation has since been improved.

A.1.5 On-Chip — Recompute. The *recompute* optimization consists of recomputing one or more values that have already been computed in the past, potentially by some other computational element. In general, when applying this optimization, work that would not be necessary otherwise is carried out to avoid either communication, in case the work has already been performed by a different thread or thread block, or to reduce memory operations or footprint, in case the work has already been performed by the same thread or thread block in the past. This optimization is based on a resource tradeoff that is in most cases beneficial for modern GPUs, where compute capabilities far exceed memory and communication bandwidth.

Various examples of the use of *recompute* to reduce communication can be found in the literature [149, 205, 208]. In particular, Holewinski et al. [149] chose to recompute the halo of every tile in their stencil operation, thus performing unnecessary work, to avoid the need of complex synchronization schemes between thread blocks. This approach resulted in speedups up to a factor 2.5 depending on architecture (a Tesla GPU (2008), Fermi (2010) GPUs, and an AMD Beaver-Creek (2011) GPU), and the ratio between computational instructions and overhead instructions. In highly compute intensive stencil computations, redundant computations could not be overlapped, resulting in equal performance to the original kernel. Different is the case of Li et al. [208], where the goal achieved by using *recompute* is to reduce the amount of communication between host and GPU. In this article, instead of communicating between the GPU and the host to correctly compute all values, the authors accepted the fact that some of the values computed on the GPU would be incorrect due to conflicts, and instead decided to recompute these values afterward on the host. Particularly interesting is the case of Lefebvre et al. [205], where they combined *kernel fusion* (see Section A.1.8) with *recompute* to reduce expensive synchronization through global memory and therefore avoided storing intermediate results.

A classic example of using *recompute* is to reduce the number of accesses to global memory and the volume of data read. Domínguez et al. [95] opted to store only values that cannot be easily recomputed instead of storing all values. It is also possible to use this optimization to reduce shared memory requirements, as Lee et al. [201] show.

In addition, there are articles that test the effects of this optimization. Lee et al. [200] (different authors than Lee et al. [201] above) implement a 3D image registration kernel, a compute-bound algorithm, and test the effectiveness of the *recompute* optimization by comparing this approach against a version of the same algorithm that stores intermediate results for reuse; in their experiments they found that recomputing is not always the optimal solution for performance. Similarly, Střelák et al. [338] compare different implementations of their GPU kernels, using this optimization to recompute some weights on the fly, instead of storing them; they conclude that *recompute* is probably more beneficial for architectures that have more floating-point performance than memory bandwidth.

A.1.6 Off-Chip — Coalesced Access. Coalesced access is perhaps the most applied optimization. When the combined memory accesses of a warp of 32 threads satisfy a set of coalescing rules, data can be fetched in less than 32 transactions from the device memory. The set of coalescing rules varies per architecture and have become less strict over time. A typical example of coalesced access is a warp accessing consecutive 32-bit memory locations that are aligned to 128 bytes, a typical size of a cache line. If these 128 bytes are in the cache, there is only one memory transaction, otherwise, there are four 32-byte memory transactions to the device memory. Figure 15 shows an example adapted from the CUDA Programming Guide [294] where the top part shows threads 0–31 accessing addresses 128 to 255 in a coalesced manner leading to 4 device memory transactions

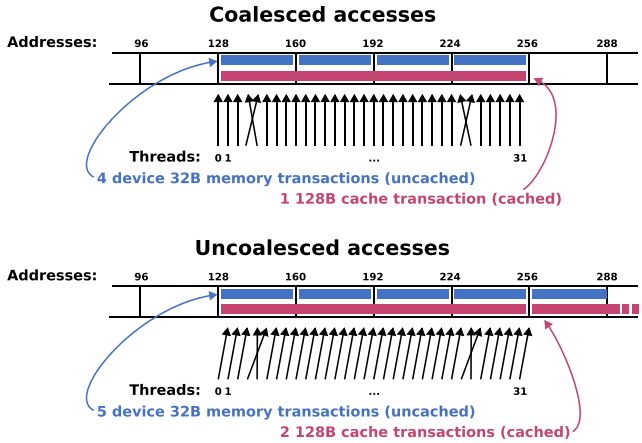


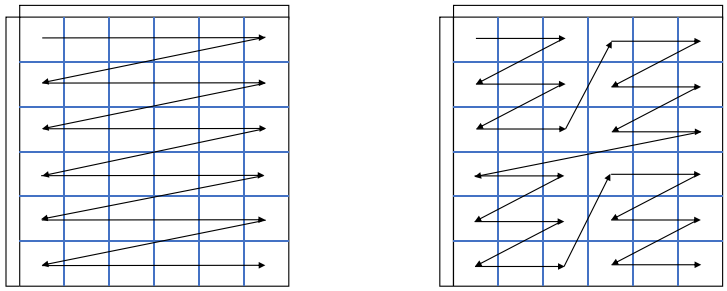
Fig. 15. Coalesced and uncoalesced access (adapted from the CUDA Programming Guide [294], © NVIDIA).

if the data was uncached and 1 cache transaction if the data was cached. In the bottom, threads 0–31 access addresses 129 to 256 leading to 5 device memory transactions if the data is uncached and 2 cache transactions if the data is in a cache.

There are various ways to achieve coalesced access [319] and the complexity of achieving it depends on several factors, such as the data layout, the organization of threads, and the availability of shared memory. If the data layout allows, a simple method to coalesce access is to organize the threads such that each thread aligns with the element on which it operates [13, 86, 123, 160, 167, 238, 359]. Other forms of aligning threads with data are simply choosing a good thread block size [206, 207], by tiling [34, 277, 278, 353] or by choosing a different parallelization strategy [96, 139, 398]. For complex data layouts, it may be an option to use the GPU's computational power to perform complex indexing operations such as space-filling curves to align the threads with the data [19, 250, 285].

If a kernel exhibits uncoalesced memory access, an often used approach is to load data from global memory in a coalesced fashion into shared memory from which the uncoalesced access can happen [50, 89, 103, 104, 104, 105, 117, 120, 129, 187, 271, 277, 282, 319, 329, 379, 414–416, 423, 438]. This can also be combined with applying tiling such that a tile is loaded into shared memory [278, 318]. Instead of loading into shared memory, it is also possible to stage the data in shared memory to store it in device memory [37, 60, 110, 127, 127, 182, 376] and several articles apply both techniques [75, 182, 245, 246, 259]. Yang et al. [414] isolate the performance of coalescing with respect to a naive version and find a performance improvement between a factor 1.8 and almost 4 averaged over various benchmarks depending on the specific GPU they used (Tesla 2008).

The above techniques maintain the layout in global memory. However, if the application allows, another often used approach is to reorganize the data such that threads access the data in a coalesced manner [337]. Besides reorganizing data in a custom way [15, 69, 163, 164, 175, 220, 227, 230, 283, 284, 314, 332, 361, 363, 387, 408, 428, 439], there are also often applied approaches. An example is creating a struct of arrays from an array of structs [51, 59, 72, 86, 112, 212, 252, 319, 331, 344, 428] possibly tiled [337, 358]. Another simple transformation is switching from row-major to column-major arrays [72, 329] or other transpositions [19, 85, 316, 337, 344, 395, 432]. A final common technique is to add padding to align the data [9, 117, 135, 136, 182, 205, 250, 316, 331, 332]. Gómez-Luna et al. improve on their own work [343] with a minimal padding scheme in an in-place matrix transposition kernel. The new approach has a higher throughput including a padding and unpadding kernel [125].



(a) Data accessed regularly. (b) Data accessed with spatial blocking.

Fig. 16. Comparison of a 2-dimensional array accessed regularly and with spatial blocking.

Ito and Nakano show two different layouts for optimal polygon triangulation [166]. They show that the two layouts are optimal in different stages of the algorithm and that reorganizing data from one layout to the other is faster (including the transformation time) than sticking to one layout.

For some architectures, there is a difference in coalesced reads and writes throughput. Therefore, in some cases, it is necessary to tradeoff data layouts to either benefit coalesced reads or writes [86, 358]. Lin et al. [220] present a new data layout that—compared to related work—manages to achieve both coalesced reads and writes.

Coalescing is difficult to achieve in applications with a high degree of irregular access. Examples of such applications are sparse matrix-vector multiplication and graph processing. One approach is caching the expensive random-access loads [89], but as seen in Section A.2.3 many sparse formats are introduced to increase the regularity. Bell and Garland [46] show that various sparse matrix-vector multiplications formats support or are designed to maximize coalescing. They show that with the regular CSR format without coalescing, Zhong et al. [434] introduce a new graph format with a data layout such that it supports coalesced access.

For sparse data structures, padding data is an often-used technique [78, 192, 277, 339, 386]. Kreutzer et al. propose a sparse matrix-vector format to achieve coalescing with reduced padding requirements compared to other sparse formats [192].

A final interesting approach is to redesign data structures to benefit from coalescing. Ashkiani et al. present a hash table that supports updates and is designed with coalesced access in mind [38]. The hash table uses chaining and the linked lists use a "slab", a linked list node with multiple key-value pairs and one next pointer, such that operations can be performed with warp instructions and memory access is coalesced.

A.1.7 Off-Chip — Loop Tiling or Spatial Blocking. Blocking or loop tiling are two frequently used terms in the literature that are used to describe a common code optimization. For example, Deftu and Murarasu state that "Loop tiling is a loop reordering transformation primarily used to improve cache reuse by dividing the iteration space into tiles" [85, p. 354]. And Nath et al. state that "Blocking is a D[ense] L[inear] A[lgebra] optimization technique where a computation is organized to operate on blocks or submatrices of the original matrix." [277, p. 2]. Despite the difference in terminology, the two refer to the same concept, which is partitioning the data or the computation in such a way that it is processed one block at a time. We refer to this optimization as blocking, or more specifically as *spatial blocking*, to distinguish between spatial and *temporal blocking* (Section A.1.8). Figure 16 shows data that is accessed in a regular way on the left. At the right spatial blocking is applied dividing the data into four tiles.

Spatial blocking is of particular interest in the context of GPU programming, because it gives the programmer control over where and how data is used and reused in the computation [39, 70, 104, 137, 138, 188, 269, 277, 304]. This allows the kernel to more effectively exploit data reuse in L1/L2 caches [188, 269, 330, 358, 382, 408], the texture cache [413], registers [138, 162, 180, 407], and shared memory [104, 138, 162, 277, 315, 353, 383, 396, 407, 425]. Blocking may have other beneficial effects, for example, Střelák et al. [338] use blocking to reduce warp divergence (See Section A.2.2). Stratton et al. [337] report a performance benefit of tiling between 12% (for an irregular application) and 3-6 times speedup (for regular applications) depending on the application. Although they report performance on a GPU of the Tesla architecture (2008) without a cache, they concede that tiling is important for GPUs with caches as well.

To apply spatial blocking it may be required to first change the data layout used for certain variables [358]. Nath et al. [277] introduce *recursive blocking* in a dense matrix-vector multiplication kernel where part of the computation is performed with a rearranged set of threads. Rojek et al. [316] introduce the term *2.5D blocking* to describe a special case of blocking applied in an advection transport algorithm.

The dimensions of the block are often directly linked to the dimensions of the thread block [182, 320]. In addition to the thread block dimensions, there may be other restrictions to the block dimensions, for example, the vector width when using vector data types [339]. When the thread block dimensions and the tile or block dimensions are decoupled, the GPU programmer has the opportunity to vary the amount of work performed by each thread and each thread block, which is considered another optimization, see Section A.3.4. Auto-tuning (see Section A.3.6) is frequently used for selecting the optimal thread block dimensions (see Section A.3.5) in combination with tile or block dimensions [166, 209, 287, 316, 320, 389, 413].

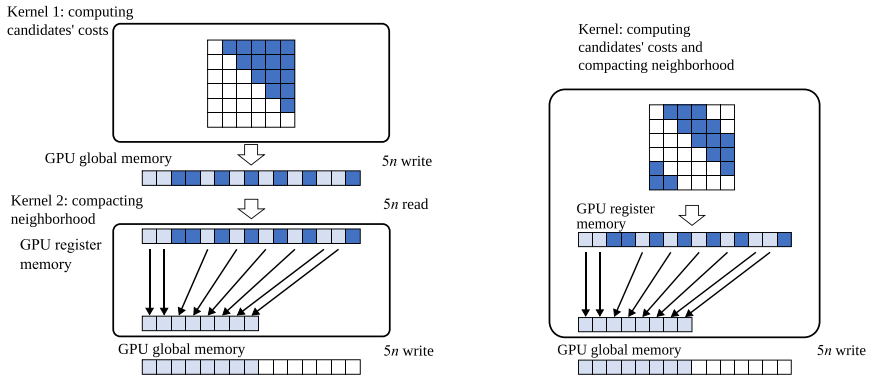
A.1.8 Off-Chip — Kernel Fusion. A kernel is a function launched on the GPU with a specific thread and thread block configuration. Kernel fusion (sometimes also called kernel merge [205, 323, 346, 422] or kernel unification [173]) is analogous to loop fusion, defined as "collapsing consecutive and dependent loops" by Carabaño et al. [62, p. 220]. However, there is a crucial difference between kernels and loops since kernels also allow synchronization at the device memory level. If two threads *within* a thread block need to synchronize, they can use shared memory and a barrier to do so. When two threads *in different thread blocks* need to synchronize, this is only possible with different kernel launches acting as a barrier [18, 62, 77, 111, 422], although there are techniques to allow global synchronization without kernel launches [400], see Section A.3.10.

Many optimizations are targeted at the kernel-level, but Sarkar et al. show that solely optimizing each kernel does not necessarily lead to the most optimized configuration but that fusing kernels can result in better execution times [323]. There are several reasons for why kernel fusion can benefit the execution times, although some articles do not list an explicit reason [9, 46, 336].

It is widely recognized that global memory accesses can be reduced with kernel fusion if two kernels use the same data [4, 30, 31, 65, 86, 111, 117, 155, 188, 205, 223, 224, 255, 256, 323, 418, 433]. Instead of fetching the data in the two kernels, data is only loaded once in the fused version. Some note that there are two ways to pass on these intermediate results: via shared memory or via registers [62, 65, 86, 102, 111, 188, 223]. In Figure 17 Hou et al. [155] show how their fused kernel (on the right) saves $10n$ accesses. They pass on the data in registers.

Some also mention other benefits to kernel fusion, such as improved cache benefits [77, 188], improved locality [77, 346], and data reuse [4, 5, 30, 102, 138, 188, 223, 323]. Finally, it is often mentioned that kernel fusion reduces kernel launch overhead [4, 5, 18, 111, 117].

Besides these reasons, Filipovič et al. also mention that kernel fusion allows room for other optimizations [111]. This article also lists potential drawbacks of fusion, such as the risk of



(a) The two original kernels. (b) The two kernels fused.

Fig. 17. Kernel fusion applied on two kernels (adapted from Hou et al. [155], © Higher Education Press and Springer-Verlag GmbH Germany, part of Springer Nature 2019).

reduced occupancy in relation to the amount of shared memory or registers that are required for the fused kernel. In addition, there can be a mismatch in parallelism that hurts performance. They define "kernel fusibility" and recognize that on-chip memory is essentially a distributed memory, so if intermediate results are passed on through registers, a kernel needs to have the same thread mapping. If the intermediate results are passed through shared memory, it requires the same block mapping and synchronization at the block level is needed. Finally, if the two kernels require global synchronization, fusion is generally not possible.

Tabik et al. also recognize that kernel fusion provides room for other optimizations [346] but they also list other advantages. Besides global memory access, they mention that kernel fusion can also reduce data movement between the host and the CPU. In addition, they note that kernel fusion can increase concurrency, and specifically instruction level parallelism because each thread has more, potentially independent work to perform. Jiménez et al. recognize that this increased concurrency provides an additional possibility to hide memory latency access [173]. Finally, Tabik et al. also list the drawback that if a kernel touches a larger volume of data, there is also a risk for bad data locality because the data can reside in different memory ranges.

Wang et al. discuss three methods for fusion [378]. The "inner thread" method fuses the work of each kernel per thread. This is what most would consider kernel fusion and this requires that the thread block and grid size matches for both kernels. The "inner thread block" method fuses two kernels and in the first instruction of the fused kernel, one of the original kernels is executed based on the thread index. For this type of fusion, the kernels need to be independent and crucially, none of the kernels can do thread block-level synchronization. Finally, the "inter thread block" method is similar to the previous one but instead of jumping to one of the kernels based on the thread index, it is based on the block index. Sarkar et al. also propose different strategies to fuse kernels [323] and are similar to the methods discussed above: fuse the work of each kernel per thread and fuse independent kernels and select per warp. They note that for kernel fusion register usage is very important but that fusing two independent kernels does not necessarily lead to the sum of the number of used registers that Wang et al. and others seem to assume. Filipovič et al. have a similar categorization [111] as Sarkar et al. [323] and Anzt et al. note that independent kernels often benefit more from concurrent kernel execution that is supported in newer GPUs than from fusion [30].

Carabaño et al. make a distinction based on the data shared between kernels [62]. It is possible to fuse kernels that share the same input (that they call "flat fusion") and fuse kernels with a

```
__shared__ float As[16][16], Bs[16][16];
float Ctemp = 0;


for (...) {
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();



  for (i = 0; i < 16; i++)
    Ctemp += As[ty][i] * Bs[i][tx];

  __syncthreads();
}
```

(a) Without prefetching.

```
__shared__ float As[16][16], Bs[16][16];
float  Ctemp = 0;
float  a = A[indexA];
float  b = B[indexB];
for (...) {
  As[ty][tx] = a;
  Bs[ty][tx] = b;
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  a = A[indexA]; // Prefetch next tile
  b = B[indexB];
  for (i = 0; i < 16; i++)
    Ctemp += As[ty][i] * Bs[i][tx];

  __syncthreads();
}
```

(b) With prefetching. Changes in red.

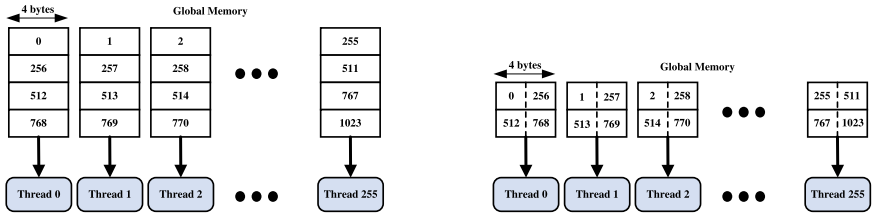
Fig. 18. Example of prefetching for matrix multiplication kernel, (adapted from Ryoo et al. [320], © ACM).

producer/consumer relationship between them (that they call "pipe fusion"). Yi et al. make the same distinction but do not use these specific names [418].

As with many optimizations, kernel fusion can introduce trade-offs. Lefebvre et al. tradeoff fusion with recomputing values, resulting in redundant work that can become expensive [205]. Chen et al. fuse kernels by adopting the work from Xiao et al. for global synchronization [77] which forces them to reduce the number of threads per block. An important resource that requires balancing is the number of registers and shared memory [4, 62]. Biferale et al. apply fusion on stencil computations but execute the non-fused version for the boundaries and the fused version for the non-boundary points [51]. Finally, Yan et al. carefully analyze dependencies for prefix sum and conclude that the traditional implementations with three kernels can be replaced with a version with only one kernel [410]. However, this relies on serializing part of the algorithm that has to be balanced by ample parallelism in another dimension. They propose auto-tuning to balance the concurrency. In the context of graph algorithms, Liu et al. [224] report a general performance improvement between 10% and 74% (Kepler 2012) depending on the application and kind of fusion, but also note that in some cases kernel fusion leads to slow down (13% in one case). Finally, they note that on all memory-intensive applications kernel fusion shows improvement.

A.1.9 Off-Chip — Software Prefetching. *Data prefetching* is a general optimization technique in which data is fetched before it is actually required. In principle, GPUs are designed to do this by switching to an active warp as soon as the current warp is stalled on a memory operation. In this section, we discuss techniques that allow more control to overlapping the high latency of global memory reads with computation. This can be done by rewriting the kernel such that those reads are initiated into temporary registers long before their values are actually used, thus enabling the scheduler to hide the memory latency of these operations at the cost of increased register usage.

Prefetching has been applied in various dense linear algebra kernels, such as matrix-vector multiplication [8, 9], matrix-matrix multiplication [6, 27, 39, 162, 237, 315, 320, 348], and stencil operations [149, 193, 262, 379]. For these kernels, the input data is tiled and each thread block loops over tiles of the input data (see Section A.1.7). Prefetching can be applied by fetching the tile for the next iteration just before processing the current tile. For example, Figure 18 shows an example of



(a) Input data uncompressed. (b) Input data compressed.

Fig. 19. Compress data to reduce the number of memory transfers (adapted from Samadi et al. [321], © Samadi et al.).

prefetching in the matrix multiplication kernel by Ryoo et al. [320]. There are a few instances of this being called *double buffering* [10, 137, 249] since two buffers are required (one for the current tile and one for the next tile), although this term is more commonly used in the context of overlapping computation and data transfers (see Section A.4.1). Some also call it *software pipelining* [407].

Prefetching has also been applied in other applications. For example, Yang et al. [414] propose a framework that can automatically optimize GPU kernels by applying code transformations and one of these is a prefetching transformation. Bauer et al. [45] present an approach for programming GPUs in which separate warps fulfill separate roles (*warp specialization*): one of these roles is prefetching data for the other warps. Wu et al. [396] investigate the effectiveness of inter-block barrier synchronization over invoking the same kernel many times (see Section A.3.10). They prefetch data before an inter-block barrier, data that is required just after the barrier, something that is impossible to achieve when using individual kernel invocations. With this technique they succeed to improve the performance of the application.

The performance benefit of prefetching varies: some researchers claim to see some improvement, while others actually see a loss in performance. This happens since additional registers are required to hold the prefetch values, which in turn can reduce occupancy [320]. For instance, Anh et al. [27] see an improvement of 20%–40% for sparse matrix multiplication on a Tesla K20Xm. Ma et al. [237] see an 8.5% improvement for dense matrix multiplication on a GTX280 when prefetching two values, but there is a 62% decrease in performance when fetching more values. Yang et al. [414] evaluate their optimizing framework using 10 kernels and the prefetching transformation obtains no additional performance across their benchmark suite. They even disable this transformation in later iterations of their framework [415].

A.1.10 Off-Chip — Compress Data. Memory is more abundant in the host than it is in GPUs, and there are numerous optimizations that aim is to reduce the memory footprint of an application; among these optimizations is *compress data*. Compression is used to reduce the size of data, be that the input, output, or even intermediate data products, and the reduction in size helps with both storage and data transfers [53, 321]. Samadi et al. [321] show in Figure 19 how they compress input data to reduce the number of memory transfers.

This optimization is widely used for sparse matrices, with different compression schemes proposed to reduce the size of the indices of non-zero elements [241, 242, 279, 282]. This compression can be beneficial not only to reduce communication overhead and memory footprint, but also to reduce pre-processing time, as shown by Neelima et al. [279]. However, reducing the number of bits used to store variables is not just used for sparse matrices but also for the longest common subsequence problem in the context of sequence alignment [297] and regular expression matching [273].

```
for (int i = 0; i < 8; i++) {
  sum += item[i];
}
```

(a) Before loop unrolling.

```
for (int i = 0; i < 8; i+=2) {
  sum += item[i];
  sum += item[i+1];
}
```

(b) Partial loop unrolling by a factor of two.

```
sum += item[0];
sum += item[1];
sum += item[2];
sum += item[3];
sum += item[4];
sum += item[5];
sum += item[6];
sum += item[7];
```

(c) Full loop unrolling.

Fig. 20. Example of loop unrolling.

Reducing the number of bits used for representing indices is not the only way compression is applied to sparse matrices; examples can be found in the work from Liu and Vinter [229], where they remove duplicate indices to save memory, or in the work by Nisa et al. [281], where they avoid using pointers in case a slice has a single non-zero value. Liu et al. [226] (unrelated to Liu and Vinter [229]) use compression for states in non-deterministic finite state machines. Since this is an irregular application, the performance of the compression relies on the presence of compressible states and whether the benefit of compression outweighs its overhead. In the automata they tested with where the overhead does not outweight the benefit, the maximum slowdown is 20%, whereas the speedup they measure is between 27% and a factor of almost 5 on a Pascal GPU (2016).

A.1.11 Off-Chip — Precompute. The idea behind this optimization is to perform some parts of the computation offline, before the main processing starts, and then reuse these precomputed values whenever necessary [402]. Often *precompute* is seen as a tradeoff optimization between time and space [338], where time is saved by reusing some precomputed and stored values, at the cost of increased memory pressure. In such cases, this optimization can be seen as the opposite of *recompute*, described in Section A.1.5. However, there are also cases in which precompute is used to move computing to different, and better suited, computing devices even if the precomputed values are just used once, such as in the work of Fortin et al. [112] where certain computations are moved to the CPU. Another special case for this optimization is precomputing some values to change the structure of the computation, as exemplified by Greathouse et al. in precomputing how many rows can fit in shared memory [129].

Zhou et al. [435] combine precompute with *using texture memory* by precomputing the city distance matrix for the traveling salesman problem, and then storing it in texture memory. The literature offers multiple examples [23, 112, 303, 308] of the combination between this optimization and dividing computation between CPU and GPU (Section A.4.2), with the precomputing phase happening on the host and the main computation on the GPU. To give an indication of potential performance benefits, An et al. [23] show a speedup of 49% by precomputing data that is equal for all threads in a password recovery program on an AMD Tahiti GPU from 2012. In general, this optimization can also be seen as one of the possible ways to implement another optimization, *reduce redundant work*, described in Section A.2.5.

A.2 Irregularity

A.2.1 Loop Unrolling. Loop unrolling is an optimization technique in which the body of a loop is explicitly repeated multiple times [270]. Unrolling of a loop can be done manually, by explicitly duplicating the body of the loop, or automatically, either using macros, C++ templates [7], or compiler directives [294]. Figure 20 shows an example of manual loop unrolling for a simple snippet of code.

Loop unrolling can have various effects that benefit performance. One clear benefit is a reduction in the overhead of loop-related instructions, such as branches and address calculations [242, 270]. Another benefit is the increase in opportunity for ILP [242, 270, 282, 345], due to offering more independent instructions to hide latencies. Finally, a common effect of loop unrolling is that it enables further compiler optimizations such as replacing array indices with compile-time constants to enable the array elements to be stored in registers [67, 216, 320], removal of branches which depend on the loop variable [112, 318, 439], and vectorization of instructions [54, 67, 309].

Overall, loop unrolling has been found to be beneficial in many different scenarios. For example, Ryoo et al. [318] optimize tiled matrix multiplication for the 8800GTX and find that best performance can be achieved for 16×16 tiles and completely unrolling the innermost loop. This gave a performance increase up to 25% but the choice of the right tile size had much more effect. Perhaps even more interesting is that loop unrolling had no effect on a tile size of 8×8, most likely caused by register usage that limits the number of threads. Van Werkhoven et al. [390] show how to loop unrolling can increase the performance of convolution operations on GPUs. Qiu et al. [308] unroll the innermost loop of a hash function since it removes a large switch statement. Fortin et al. [112] accelerate Lefèvre's algorithm and unroll a loop by a factor of two since it removes a conditional that only holds every other iteration. Reddy et al. [311] completely unroll a reduction tree since it reduces synchronization between threads. Lin et al. [216] unroll a loop which causes a small array to be no longer dynamically indexed allowing the compiler to store the elements in registers, resulting in a 71% improvement in performance.

However, although unrolling loops is often beneficial, unrolling by a factor that is too large might hurt performance. One reason is that loop unrolling increases the code size and performance can deteriorate when the loop size exceeds the instruction cache size [270]. Another reason is that unrolling loops can increase pressure on register usage which in turn reduces occupancy [105, 169, 270, 319], although some authors claim to see a decrease in register usage after unrolling [320]. The factor by which to unroll a loop is thus an important consideration and various researchers have focused on finding the ideal factor [270, 287, 309], often concluding that it differs per kernel and device [287].

A.2.2 Reduce Branch-divergence. Branch divergence, sometimes also called path divergence, is a problem for SIMD or warp-based architectures such as GPUs where a warp executes instructions in lock-step. On a branch that depends on the thread index, there is a chance that different threads in a warp have to take different paths on a branch. On a GPU this results in all threads taking both branches where the threads that should not take the branch do not write the result. This hurts performance because both branches are executed serially. If the instructions within the branch are simple enough, the branches can be replaced with predicated instructions in which only the threads with a true predicate write back the result of the instructions [52, 112].

Removing branches. A first technique is to remove branches altogether, since branching contributes to overhead [23, 97]. In some cases, maximum and minimum functions can remove branches [59, 231] or other arithmetic operations [177, 358, 379].

Figure 21 shows an example from Wang et al. [379]. The original code with branches is at the top and at the bottom, the branches are replaced by arithmetic operations. A shaded box is an instruction for an inactive thread within the warp (a thread for which the condition is false) meaning that the instruction is executed but the result is not written back.

To give an indication of the potential and limitations of branch-divergence reduction, Wang et al. [379] observe a speedup of 3.45 on an AMD GPU from 2008 using the Brook+ programming language on a branch-intensive kernel and no speedup for this optimization on other kernels. Tran

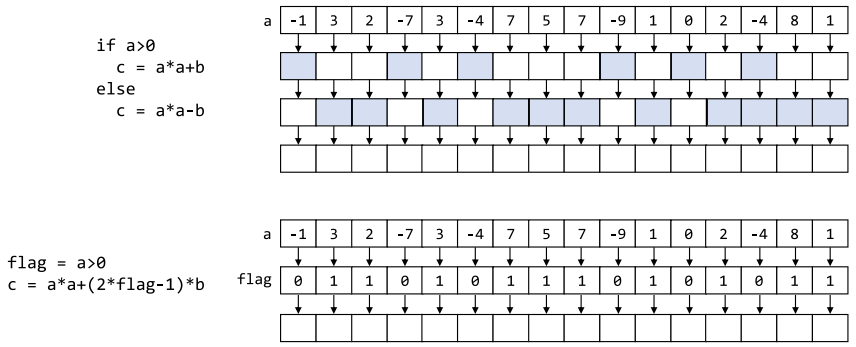


Fig. 21. Reducing branch divergence by replacing a branch with arithmetic as exemplified by Wang et al. [379]. The shaded boxes are instructions of inactive threads in the warp.

et al. [358] report speedups around 10% on a Kepler GPU (2012) with a Lattice Boltzmann solver where branches are avoided on the boundaries of the 3D stencil computation.

Chakroun et al. [68] replace branches with sine, cosine, and exponentiation functions (using fast math functions, see Section A.3.2), something they call branch refactoring. Vespa et al. take this technique to an extreme in what they call algorithm flattening where each branch is expressed as a computation that is reduced and optimized further [369].

Jiang et al. perform "instruction stream normalization" [172] that eliminates branches as well. They store conditions in a 4-bit wide feature vector that indexes in a lookup table in constant memory that provides the thread with the right value. Daga et al. [80] remove branches with kernel-fission (Section A.2.4), splitting kernels based on conditionals that can be computed before kernel launch on the CPU. They state that this optimization affected AMD GPUs more in their case since they had less branching units per processing core than NVIDIA GPUs.

Reducing branches. If branches cannot be removed, the next best strategy is to reduce branching. Cecilia et al. [64] apply Ant Colony Optimization to the Traveling Salesman Problem and notify other threads in the block via shared memory whether a city has been visited. They later improve on this by using warp instructions such as shuffle [65]. Several articles use loop unrolling to reduce control flow in a kernel [112, 308, 311, 439]. In loops, it is also possible to adopt a different technique. Han and Abdelrahman introduce iteration-delaying [140] which delays iterations of a loop with a branch inside such that an iteration only takes the path that a majority of the threads will take. In a later iteration, the other path is taken when it has the majority of the threads. They use warp-voting functions to determine the majority of threads. The authors note that it may be problematic that the minority threads for a certain path may be stalled for a long time. Novak solves this by allowing only stalled threads to take part in the vote, trading off some efficiency for the progress of threads [286]. Zhang et al. use a somewhat similar method that checks the condition at the end of the loop, allowing threads without work to skip an iteration [431]. Another loop-based method is introduced by Han et al. that merges nested loops to allow threads in a warp to start the next iteration while other threads execute code from the inner loop [141].

Trading off branching. Another strategy to reduce branching is to tradeoff divergence against another metric. Okada et al. reduce branching in gene sequence alignment at the cost of more redundant work because threads in a warp take a similar path [296]. In their case, more redundant work is less costly than branching. Sartori and Kumar introduce branch-herding [324]. They apply this technique on kernels that can tolerate errors and they use warp voting instructions to find the

majority of the threads taking a certain path. They then force all threads in the warp to take this path. Users can control the error rate by using performance counters allowing to set a threshold of forced branches. They apply similar techniques for memory coalescing.

Hong et al. propose a new programming method for dynamic applications such as graph processing that discriminates between SIMD execution and traditional, sequential SISD execution [154]. For the latter mode, each warp executes the same instruction on the same data, which reduces the parallelism within a warp but eliminates branch divergence. By decomposing warps into smaller virtual warps, they can tradeoff branch divergence with serial execution within a warp.

Reducing the effect of branches. In contrast to reduce branching, it is also possible to reduce the effect of branching. Han and Abdelrahman introduce branch-distribution [140] that aims at reducing the code in branches as much as possible, removing code that does not necessarily need to be inside branches, often at the cost of introducing new registers to store temporary results [112, 140, 178]. If the instructions in the branching code are simple enough, the compiler may be able to predicate the instructions, masking threads that should not write a result. Lalami and El-Baz use a similar technique by only employing registers in the branches [198].

Thread/data remapping. The above techniques apply to branches directly but it is also possible to affect branching indirectly. We will discuss the parallelization scheme and data layout in the following paragraphs focusing on more static applications first.

In stencil computations, often data is loaded into shared memory including a border. Instead of using branches to load the border data, some assign threads to the border as well that in a later stage do not participate in the computation [206, 251, 262, 402]. Itu et al. apply a different strategy and do not load the left and right border into shared memory but instead load these values directly from global memory [168]. Yet another approach for stencils is to assign all the divergent load operations across all the warps to a single warp to speed up the others [316].

Phuong et al. study several optimizations of reductions on GPUs and the first optimization to apply is avoiding branch divergence [305]. They achieve this with a strided access and evaluate this on GPUs with different generations. They notice that the newer GTX 970 benefits more from this optimization than older ones such as the GTX 560-Ti.

There are more articles that change the parallelization scheme for reducing branch divergence. Chang et al. present a dynamic tiling solution for a high-performance tridiagonal solver [69]. Honda et al. employ warp-synchronous programming to perform high-precision integer multiplication (1024 bits). They assign the multiplications to threads such that branch divergence is mostly avoided [150]. Yamashita et al. propose new thread assignments to the data of optimal polygon triangulation [405]. With these new thread assignments, they eliminate branch divergence. Hu et al. reduce branch divergence in a graph application by processing an edge per warp instead of an edge per thread [160].

Zhang et al. investigate how to reduce branch divergence in highly dynamic, irregular applications where static analysis does not apply [426]. They propose the thread-data-remapping technique to reduce branch divergence with two mechanisms: transform the data layout and reference redirection. The former technique creates a copy of the data such that it minimizes thread divergence and the latter technique creates an extra indirection to a memory access such that the accesses minimize thread divergence. Since these techniques result in overhead, especially the data layout transformation, they apply pipelining to overlap kernel execution with changing the layout that happens on the CPU. In addition, they propose an algorithm to minimize the data layout transformation by replacing it with reference redirection. In later work [427] the authors extend their work to the G-Streamline framework that also focuses on irregular memory accesses and introduces an additional technique *job swapping* that essentially results in reordering the thread

indices. Lin et al. present a source-to-source compiler solution that applies thread-data-remapping on the GPU instead of on the CPU [219]. Later work also supports nested branches [218].

An often used term in combination with warp divergence is thread block compaction introduced by Fung and Aamodt [114], a micro-architectural improvement. This term or the more general term stream compaction is used in combination with warp-divergence but is not to be confused with stream compaction as a parallel algorithm to filter items, also a target of optimizations on GPUs [42, 327]. In the context of warp-divergence, it means that threads for a specific path are filtered to execute together.

Besides the work from Zhang et al., the work by Khorasani et al. can also be regarded to belong to this class of solutions. They propose Collaborative-Context-Collection that targets loops with branches [183]. Threads in a warp use shared memory to store a context that describes which branch would be taken. If enough contexts for a specific branch are recorded, the threads in a warp choose a context to execute, ensuring that there is no branch divergence.

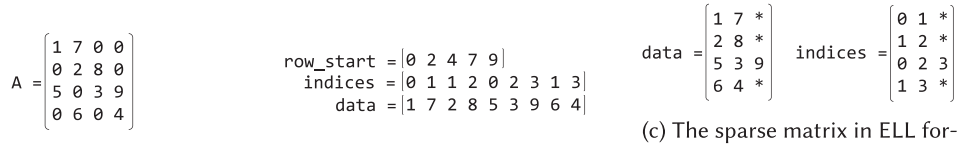
Gmys et al. use several techniques to reduce the amount of branching code [121]. They apply stream compaction by first computing which similar work should be assigned to each thread, then launch a reduced number of threads to perform the operations. In addition, for code in which they cannot prevent a high degree of branching, they launch kernels with only one thread per warp, similar to the virtual warps that Hong et al. [154] introduced.

Related to compaction is sorting the data such that a similar workload is assigned to the same threads in a warp or block. Burtscher et al. present a Barnes-Hut n-body application. They sort the bodies to group spatially close bodies together. This grouping reduces branch divergence because force computation is initially the same for close-by bodies [59]. Nasre et al. apply a similar technique to sort triangles in Delaunay Mesh Refinement [276]. Others use similar techniques [68, 175, 301]. Kovac et al. predicts the number of iterations that a thread has to make and sorts the tasks based on this prediction [189]. Arumugam use a heuristic based on the previous iteration of their algorithm to map threads to similar tasks [33]. Djenouri et al. also sort the input based on size and distribute this equally over the thread blocks [90]. However since this distribution may still result in thread divergence, they apply statistical techniques to minimize this occurrence.

Change the data layout. In a sense, sorting the data is changing the data layout but it is tied to parallelism in a high degree. Another solution that changes the data layout more purely is padding [64, 396, 435]. Sitardi et al. search strings in parallel for a certain first match [332]. They first group the strings by length and then split the strings in multiple phases to give each thread in a warp an equal-sized string.

For sparse data structures such as sparse matrix-vector multiplication or graph algorithms there are many techniques used for branch divergence as well. Several researchers introduce sparse formats that reduce branch divergence [109, 242, 342, 352, 366, 386]. Some formats are specifically designed with the goal to reduce branch divergence: Ashari et al. introduce the Blocked-Row Column format [34], Khorasani et al. propose a graph representation format called G-shards [186] and other graph formats are designed with reducing branch divergence in mind as well [109, 253, 266]. Sha et al. decode the adjacency list of a graph in two steps reducing branch divergence [328].

Algorithmic changes. Finally, there are also techniques that are highly tied to the algorithm they execute: Zhang et al. improve with their framework RegTT [428] on AutoRopes [122], frameworks that perform multiple queries in parallel while traversing trees. RegTT improves on AutoRopes by employing breadth-first-search until a certain depth before a depth-first search that allows it to reorder the queries for common paths. This ensures that threads in a warp have a high chance of reduced branch divergence.



(a) The sparse matrix in dense format.

(b) The sparse matrix in CSR format.

(c) The sparse matrix in ELL format. The entries marked * are padding.

Fig. 22. A sparse matrix in three different formats (adapted from Bell et al. [46], © ACM).

A.2.3 Sparse Matrix Format. GPUs are commonly used for sparse linear algebra operations, and most notably SpMV. The memory format used to represent a sparse matrix has an important influence on performance and an abundance of different formats have been proposed. This section discusses some of the most prominent contributions in sparse formats designed for GPUs. We refer to Bell and Garland [46] for an overview of the essentials and to Muhammed et al. [266] for a recent review. Figure 22 shows three common formats from Bell et al. [46]. The ELL format has padding which is marked with *.

It is difficult to give an indication of performance benefits because sparse matrix-vector multiplication is a highly irregular application where performance relies heavily on the input. Muhammed et al. [266] give a good overview with mean and aggregate throughput over various inputs. Concretely, for their SURAA sparse matrix format, they report an average speedup over a range of inputs between 1.1 and 40 depending on the competing format on a Kepler GPU (2012). For specific inputs, they report a speedup of between 1.94 and 562 but also slowdown compared to a competing format.

The *ELL* format [46] (also called *ELLPACK*) is commonly used for matrices where the *number of **non-zeros*** (**nnz**) is roughly consistent across all rows which is common for matrices obtained from structured grids or semi-structured meshes. In this format, a sparse matrix is condensed into a dense matrix by removing the zeros, shifting the non-zero entries to the left, and padding with zeros until reaching exactly K entries in each row. This results in a format which is highly regular and suitable for GPUs, at the cost of adding padding. Several improvements have been proposed to overcome this padding overhead. For instance, *ELL-R* [365, 366] adds an additional array storing the actual length of each row without padding. This allows threads to skip the padding zeros and avoids performing useless work, albeit at the cost of storing an additional array. *Sliced ELL* [264] divides the matrix into strips of C adjacent rows and stores each strip in ELL format. The value of K differs in each strip, thus reducing the overhead of padding for the overall matrix. *ELLR-T* [100] combines these two ideas: it stores each strip in ELL format, like sliced-ELL, and also adds an additional array storing the row lengths, like ELL-R. SELL-C-σ [192] improves upon sliced-ELL in two ways: it ensures that C is a multiple of the SIMD width, to allow vectorization, and it sorts each consecutive σ rows by row lengths, putting rows of similar lengths into the same slice. Kreutzer et al. [192] perform an extensive analysis of the parameters C and σ to tune the best performance for different architectures.

Instead of modifying an existing format, another direction of research is into combining different sparse formats. For example, Bell and Garland [46] propose a hybrid format called *HYB* which stores the first K non-zeros of each row in ELL format and any excess non-zeros in COO format. This benefits from the regularity of the ELL format without suffering from excessive padding. Yang et al. [413] combine ELL and CSR: rows of the matrix are grouped and each group is stored as either ELL or CSR format based on a simple heuristic. Su et al. [339] take a more rigorous approach by presenting the *cocktail* format. In this format, the matrix is partitioned and each submatrix is stored in one of 9 different existing formats. They present a framework which is able to recommend

the best partitioning of any given sparse matrix for certain architectures based on offline training data.

Sparse matrix operations usually have a low arithmetic intensity and are memory-bound [403]. Compression of the format can help to increase throughput at the cost of additional computations. For instance, Xu et al. [403] use index compression to store the column indices using 16 bits instead of 32 bits, but it works only for certain matrices. Tang et al. [352] propose a **bit-representation optimized** (**BRO**) compression scheme which consists of delta encoding followed by bit packing. They apply their scheme to three existing formats and see an average speedup of 1.5×. Yan et al. [409] propose a format called ***blocked compressed common coordinate*** (**BCCOO**) which relies on bit flags to compress the row index array. This essentially achieves a compression ratio of 32 (from 32-bit integers to 1-bit per index), but at the cost of decoding the indices at runtime.

The above formats assume matrices are static, but—often in the context of graph processing where sparse matrices represent graphs—there are also dynamic formats that allow addition and removal of edges at runtime. Méndez-Lojo [253] present a dynamic format based on wide sparse bit vectors which they use to concurrently apply graph rewrite rules. Busato et al. [61] present *Hornet* which represents the matrix through a hierarchical data structure that relies on vectorized bit trees.

A.2.4 Kernel Fission. *Kernel fission* is either the process of dividing a single kernel into multiple ones, or of splitting a single kernel iteration into multiple iterations. It can be seen as the opposite of kernel fusion (Section A.1.8). The way this optimization improves performance is by better resource utilization, achieved through simpler and more regular kernels.

A classic example of the use of kernel fission can be found in the work by Reddy et al. [311], where a parallel reduction is split in two consecutive kernel executions where the first one computes multiple partial reductions in parallel, and the second one combines the partial results to compute the final value. In a similar way, this optimization is used in the work by Satish et al. [325] to overcome the lack of inter-block synchronization (Section A.3.10) on the GPU.

In the context of cryptography, kernel fission is often used to simplify the structure of complex kernels [23, 52]. Particularly interesting is the fact that An et al. [23] decided to execute one of those kernels on the CPU, therefore combining this optimization with the CPU/GPU computation technique (Section A.4.2). Kernel fission has also been used as a means to *reduce branch divergence* [63, 80] (see Section A.2.2), by having different kernels for different branches, and then letting the host execute the correct kernel depending on run-time conditions. However, this solution can be complex in practice, and in the use-case provided by Daga et al. [80] a single kernel has been replaced by 16 different ones, one for each possible execution path. However, with this technique they obtain a speedup of 70% on an AMD Cypress GPU (2009), an architecture that is low on branching units using a molecular modeling application.

Improving regularity is something for which kernel fission has been extensively used. Multiple examples can be provided for sparse matrix-vector multiplication [35, 81, 89], where splitting a single kernel into multiple ones can help in exploiting local regularities. Even in the case of *dense* matrix-vector multiplication, this optimization can be used to enable processing of matrices of arbitrary dimensions [8]; in this article, Abdelfattah et al. use a different kernel to process the bottom rows of the matrix, instead of implementing corner cases in the main kernel.

Also worth mentioning is a connection between kernel fission and *auto-tuning* (see Section A.3.6). In fact, among the reasons Alimirzazadeh et al. [19, 88, 112] have to split their original kernels into smaller and simpler ones is that these smaller kernels can be more easily optimized, and their run-time configurations better tuned.

A.2.5 Reduce Redundant Work. As the name *reduce redundant work* implies, this optimization consists of avoiding to perform work that is considered redundant. It is the opposite of recomputing values (see Section A.1.5). While it may seem that the main result of applying this optimization is only a reduction in the amount of operations performed, such as in the work of Qiu et al. [308] and Střelák et al. [338], in general reducing the amount of redundant work can also result in less memory operations and communication overhead. The optimization is often used in processing irregular data structures, such as graphs [226, 253, 255, 298], in dynamic programming [296, 322], or linear algebra [34]. In the area of non-deterministic finite state machines, Liu et al. [226] achieve a performance improvement of between 70% and a factor 6 on a Pascal GPU (2016) with their implementation that does not store redundant information compared to a naive implementation. Although the *reduce redundant work* optimization is generic, its implementation is usually algorithm specific.

A.3 Balancing

A.3.1 Balancing the Instruction Stream—Vectorization. Vectorization consists of using vectors instead of scalar variables, and manipulating such vectors by means of instructions that are applied to all the elements of a vector. This optimization is often implemented by using vector data types in CUDA and OpenCL, although both languages also support special vectorized load and store operations [102, 332, 348]. However, while support for vectorization is provided by both programming languages, not all GPUs support vector instructions in hardware, and therefore these instructions can be internally implemented as a series of scalar instructions applied to the vector's elements. There are even cases where vectorization is achieved through a mixture of *data reordering* and compiler optimizations [170].

The most common use-case for this optimization on GPUs is the use of vector data types to access memory [12, 45, 54, 67, 78, 80, 102, 134, 188, 273, 287, 415, 416]. By using vectors it is in fact possible to reduce the number of instructions issued to load or store a certain amount of memory, and therefore achieve higher memory bandwidth. Vectorizing memory operations does in particular help with improved *coalesced access* [12, 54, 67, 134, 188].

In the context of code portability, Yang et al. [415] show how both AMD and NVIDIA GPUs can benefit from vectorization, albeit by using different vector sizes, and how NVIDIA GPUs mainly benefit from vectorization when using *texture memory* [416]. Specifically, an a Fermi (2011) GPU, they achieve a speedup of 2.3 and on a Cypress AMD (2009) they measure a speedup of 6.1 on average across all their benchmarks. Another common use-case for vectorization is to control the number of threads in a certain computation and the amount of work per thread. As an example, Jang et al. [169] use this optimization to compute more values per thread, and therefore reduce the number of threads necessary to solve a problem. Reis et al. [312] also use vectorization to increase the amount of work per thread, while also using vector operations to parallelize sequential loops. Similarly Ragan-Kelley et al. [309] use single vector instruction to replace small loops, and Nisa et al. [282] use vectors to reduce the number of inner loop iterations. Another example worth of mentioning combines *merge threads* and vectorization [119], with vectors used to reintroduce the amount of parallelism removed by merging threads.

Vectorization has been used on GPUs to achieve other goals, from accessing sparse matrices [214, 339], to improve data storage [220], and from optimizing numerical operations [314, 371], to reduce the amount of conditional statements and assignment on small data types [23]. As with any other optimization, the use of vectorization is not without cost, as Volkov and Demmel demonstrate with their micro-benchmarks [375], and in some cases using vectors may even result in worse performance, as shown by Doerksen et al. [93].

A.3.2 Balancing the Instruction Stream—Fast Math Functions. The *fast math functions* optimization consists of using approximate mathematical functions that are significantly faster than the standard ones, and are usually implemented in hardware in the special function units. This optimization can be manually applied by the programmer, using intrinsics instead of the standard functions, or it can be enabled for all mathematical functions by the compiler.

In our selection of literature, fast math functions are often applied selectively by the authors where the authors use intrinsics to manually select the fast version of the mathematical functions they need [59, 64, 68, 363], with only one case where the optimization is enabled globally via a compiler flag [367]. In all cases, the authors state that the precision offered by using fast math functions is good enough for their purpose. Worth of note is that in [68] this optimization is used in conjunction with another optimization, *rewrite branch to arithmetic* (see Section A.2.2), with the approximated functions used in the equations that are introduced in the code to replace the conditions. To give an overview of the performance impact of using fast math functions, Cecilia et al. [64] report a speedup of between 3.3 and 4.8 on a Fermi GPU (2010). This speedup depends on the problem size of the Traveling Salesman Problem and is achieved by replacing the expensive `powf()` with an intrinsic.

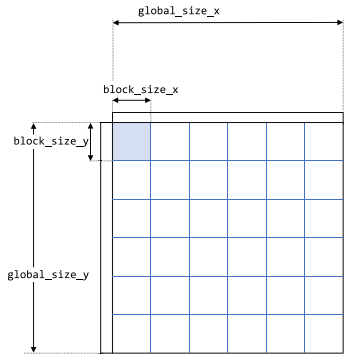
In recent architectures (Volta, Turing, Ampere), NVIDIA introduced tensor cores, functional units specialized for tensor operations that are ubiquitous in machine learning applications. These tensor cores provide the matrix-multiply accumulate (MMA) instruction [350] with varying precisions, for example, half precision. Yan et al. [407] studied this operation in detail analyzing the throughput and latency in relation to architectural bottlenecks. Typically these tensor cores are targeted via libraries such as cuBLAS or cuDNN but they can also be targeted directly. Hagedorn et al. [137] support these instructions in a scheduling language, Tang et al. [350] use them for skinny matrix multiplication, Yan et al. [407] for optimized half-precision matrix multiply, and Zhai et al. [425] use them for batched small matrix multiplications.

A.3.3 Balancing the Instruction Stream—Warp-Centric Programming. The idea behind this optimization is to make warps an integral part not only of the program execution model, but also of the programming model, and therefore write code that is structured around the idea of warps as units of computation. Particularly interesting is the relationship between this optimization and *reduce synchronization* (see Section A.3.8), as one of the effects of *warp-centric programming* is a reduction in synchronization overhead. One way to implement warp-centric programming is to reorganize the code so that work is not assigned to threads, or thread-blocks, but to warps instead [46, 65, 150, 151, 154, 179, 248, 328, 437]. (See Figure 24 in Section A.3.7 where Zhu et al. [437] compare thread-centric and warp-centric programming for load balancing.)

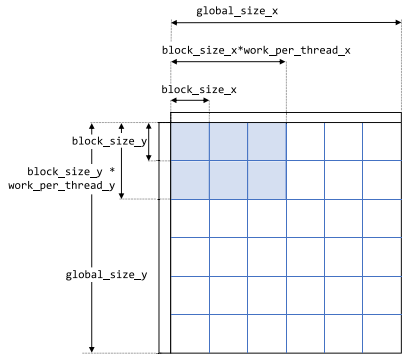
Hong et al. [154] give an example of potential speedup that can be gained with this technique using a Tesla GPU (2008): They apply breadth-first search on four different graphs and depending on the graph, they obtain a speedup of 4% and a factor 2.5 and factor 8. However in one instance, they experience a slowdown of 18% in one of the graphs.

Cecilia et al. [65] extend this concept to what they call “super warps”, virtual warps which size is larger than hardware supported warps, and assign work to them. However, they have to manually provide synchronization primitives to such warps by writing PTX code. Nonetheless, the decomposition of warps into smaller units [151, 184, 185] is more common in literature than super-warps. An interesting example of the use of virtual warps of arbitrary size is provided by Busato et al. [60], whose virtual warps are dynamic and their size is based on some properties of the input graph.

Common reasons to use warp-centric programming are implementing some form of load balancing [121, 239, 281, 332], hiding latency [10], and implementing nested parallelism [417]. Another



(a) The 2-dimensional domain split up such that a thread has one work unit.



(b) The 2-dimensional domain split up such that a thread has 6 work units.

Fig. 23. A 2-dimensional domain split up with a different amount of work per thread.

application of this optimization is the "warp specialization" described by Bauer et al. [45]. In this work, a thread-block is divided into warps that do different work, and such warps are specialized into two classes performing either memory or compute operations.

A.3.4 Parallelism-related Balancing—Varying Work Per Thread. Varying the amount of work assigned to each thread and thread block is one of the most important and generally applicable code optimizations in GPU programming. The main idea of the optimization is to change the mapping of threads to the problem domain in a way that either increases or decreases the work performed by each thread. This can be achieved in various ways, for example, for a given kernel one could increase the amount of work per thread by either decreasing the number of threads per block or decreasing a total number of thread blocks. Figure 23 shows a 2-dimensional domain split up such that a thread performs one work unit on the left. At the right side, the work per thread has been increased with a factor three by means of `work_per_thread_x` and with a factor two by means of `work_per_thread_y`.

One of the first articles to name this as an optimization in the context of GPU programming is by Ryoo et al. [320], who used the term *1x2 rectangular tiling* to refer to the concept of computing the work of two tiles using one thread block. Volkov and Demmel [375] also apply this optimization and refer to it as *strip mining*, which is synonymous to partial loop unrolling or loop sectioning. Increasing the work per thread can be seen as strip mining the outer loop that has been parallelized among the threads in the GPU kernel. Yang et al. [414] distinguish this optimization into *thread-block merge* and *thread merge*, where "thread-block merge determines the workload for each thread block while thread merge decides the workload for each thread" [414, p. 90]. Hsu et al. [157] state that thread merge is also known as *work-item merge*. Magni et al. [243, 244] performed an extensive evaluation of this optimization referring to it as *thread coarsening*. They show that on half of the benchmarks they used, this optimization has effect and that the effectiveness varies among architectures. For example, the optimization has more effect on the AMD GPUs they use (Cypress 2009, Tahiti 2011) then on NVIDIA GPUs (Fermi 2011, Kepler 2012). On Fermi, they achieve a maximum of 3.95 speedup, on Kepler a maximum of 2, on Cypress a maximum of 3.78, and on Tahiti a maximum speedup of 12.01. Hong et al. [152] also use the term *block coarsening*, analogous to thread coarsening but on the level of thread blocks. They also note that "thread coarsening is an approach to achieving register tiling" [152, p. 737], see Section A.1.3. Garvey et al. [119] also refer to this as block merge and state that it is a form of tiling.

In general, increasing work per thread increases the exploitation of data reuse in kernels that exhibit data reuse, at the cost of increased resource usage in terms of registers and shared memory. Increasing work per thread also condenses the instruction stream by eliminating redundant instructions previously distributed among multiple threads, i.e., many index calculations, loop counters, and branching instructions may be used to support the computation of multiple elements instead of just one. Increasing the work per thread is sometimes used to create an opportunity for using vector data types [169], which can in turn improve memory throughput (see Section A.3.1).

It is generally not trivial to find the optimal amount of work per thread, which may depend on input dimensions and the particular hardware architecture. Therefore the parameter is often included in auto-tuning optimizations (Section A.3.6).

A.3.5 Parallelism-related Balancing—Resize Thread Blocks. In general in GPU programming, programmers have quite some freedom in choosing the parameters *number of threads per thread block* and *number of thread blocks*. For example, for a simple vector addition of 2^{20} elements where each thread adds one element can be executed with 1,024 threads per thread block and 1,024 thread blocks, 512 threads and 2,048 thread blocks, 256 threads and 4,096 thread blocks, and so on. In some cases, the precise number of threads per thread block will not matter much [124] but in more complex kernels performance may differ significantly [205]. Specifically, they report a speedup improvement of a factor 30 on a 2D structured grid computational fluid dynamics solver between the optimal thread block configuration and the non-optimal one on a Tesla (2008) GPU.

Usually, the compiler determines the number of registers per thread [193], so a larger number of threads per thread block influences register usage [119, 173, 201, 209, 363, 372] and may lead to register spilling [152, 205]. In addition, the number of threads determines (among other factors) how many thread blocks can run on a SM, with more (independent) thread blocks per SM allowing more room to keep the compute units busy in case of barriers [119, 125, 173]. Quite some authors explicitly consider the latency of memory operations compared to computation operations of warps [152, 157, 182, 341, 359, 363]. The number of threads typically also influences the amount of shared memory that is used by a thread block [125, 186, 195, 201, 209, 372], something that also influences the number of thread blocks per SM [173].

The number of threads per thread block is one of the most used parameters applied in auto-tuning (see Section A.3.6) because firstly, the number of threads has a large influence on all of the above aspects, performance often relies on a proper interplay between all of these aspects, and a correct balance is often difficult to reason about and find [201, 205]. Secondly, the number of threads is relatively easy to change in combination with the total number of thread blocks and therefore some of the authors use this to automatically tune their kernel [152, 161, 209, 223].

Depending on the kernel, resizing the thread block may also influence other aspects. Some change the number of threads depending on whether they use regular or wide (for example 128-bit) load operations [214, 348]. Lee et al. [200] want to increase the number of threads but are limited by the number of used registers. Therefore, they replace registers with shared memory, also to increase reuse. Krotkiewski and Dabrowski adapt the thread block size to balance the number of interior elements with bordering elements in a stencil application [193]. Sugimoto et al. [341] consider a high number of resident thread blocks on an SM important if a kernel exhibits high locality among thread blocks to make good use of the cache. In contrast, they have an application with poor locality and aim for increasing the number of threads per thread block for overcoming memory latency. Wu et al. [395] adaptively adjust the number of threads for different phases in their dynamic programming application.

Although resizing thread blocks is often performed for the same reasons as varying the work per thread (see Section A.3.4), changing the number of threads does not necessarily change the amount

of work per thread. However, it is possible to organize the kernel such to make the amount of work per thread dependent on the number of threads [143, 161, 182, 209, 372].

A.3.6 Parallelism-related Balancing—Auto-tuning. There are often algorithms that require parameters to be set, so that the algorithm works as designed, and there are also performance optimizations that require some parameters to be set for the optimization to be effective. The optimal value of such parameters may be difficult to find because of the combinatorial nature of configurations, and the search can be further complicated by dependencies on the input size, or the execution platform. The process of automatically, or with minimal user interaction, exploring the configuration space to find the optimal configuration is called *auto-tuning*.

Auto-tuning has long been used for CPUs, and as early as 2008 Govindaraju et al. [127] were using this optimization to determine which FFT implementation to use on a GPU. However, although *auto-tuning* has been used to tune parameters for various GPU applications in different domains [68, 125, 166, 348, 405, 410, 413], there are some areas that have seen a wider adoption of such optimization.

The largest of these areas of application is the one related to sparse matrices [16, 34, 36, 78, 101, 129, 130, 214, 242, 266, 272, 409, 420, 432] (See Section A.2.3). Among these articles, of particular interest is the work of Yoshizawa et al. [420] in which they do not simply tune some parameters of their algorithm, but also use *auto-tuning* for parameters of the sparse format itself. Also interesting are the tuning approaches based on sparsity characteristics of the matrices proposed by Ashari et al. [34], and the technique proposed by Choi et al. [78] that is based on a combination of modeling and benchmarking.

Two other areas that have seen wide adoption of *auto-tuning* are stencil operations [79, 83, 119, 316, 340, 351] and dense matrix multiplications [191, 209, 277, 282, 380, 404]. Especially worth of mentioning are the machine-learning based tuner used by Garvey et al. [119] to accelerate their stencils, and the auto-tuning framework for stencil computation presented by Christen et al. [79]. Another related area where *auto-tuning* is used is linear algebra [4, 6, 9, 10, 235], and in particular the Cholesky factorization [5, 7, 138, 375].

A classic application of *auto-tuning* is the tuning of compiler flags, and this remains true also for GPUs [57, 316]. Other applications, such as tuning the number of threads [380] or the thread coarsening factor [244], are more specific to GPUs, as is tuning the number of execution streams [106]. Scheduling is also something that can benefit from tuning [309, 382].

The literature on *auto-tuning* on GPUs also includes the use of more generic tuners and tuning frameworks, including CLTune [287], PADL [58], Kernel Tuner [389], OpenTuner [147], and the Kernel Tuning Toolkit [304] that provides an API to run and tune kernels; we also found tuning incorporated in the generic approach to GPU optimization by Hong et al. [152]. Although the main goal of *auto-tuning* is the improvement of performance, performance portability is often mentioned as another goal for this optimization [9, 191, 209, 214, 338, 351]. Autotuning can significantly improve the performance if the right configuration is found. Magni et al. [244] provide a good example where they found an optimal amount of work per thread (Section A.3.4) resulting in a speedup of 2–12x with different benchmarks on a Fermi (2010), Kepler (2012), AMD Cypress (2009), and an AMD Tahiti GPU (2011).

A.3.7 Parallelism-related Balancing—Load Balancing. GPUs provide multiple levels of parallelism and on each level load balancing is important for performance. We discuss load balancing on several of those levels.

Within warps. Warp divergence and warp load balancing are similar but distinct problems. Warp divergence refers to the problem that threads in a warp execute both paths of branch serially,

so there is duplication of work. Load balancing, on the other hand, refers to the problem that some of the threads do not perform useful work whereas others do, so there is no duplication of work. In many cases, reducing branch divergence also improves load balancing and several articles target these both goals simultaneously [34, 91, 109, 126, 154, 183, 242, 428]. Below we discuss work that employ techniques specifically for load-balancing, see Section A.2.2 for reducing branch divergence.

Graph processing on GPUs typically leads to load-imbalance. Hong et al. solve intra-warp warp-divergence with virtual warps [154]. However, a thread in a warp can still have a large workload causing load-imbalance within the warp. They use a technique "defer outliers" that puts a large task in a global worklist from which a different kernel will execute the large task, parallelized over the threads of a warp. Merrill et al. compare various strategies for performing breadth-first search [255]. They conclude that a hybrid form they call "CTA+Warp+Scan" works best for load balancing. This method divides work over either a thread block, warps, or uses a scan-based strategy for the threads within a warp based on the size of the adjacency lists in the graph. Others also use this technique [77, 411] or a similar design [224]. Davidson et al. compare several load balancing mechanisms including Merrill's "CTA+Warp+Scan" and propose Load-Balanced Partitioning that groups work according to the number of edges in a graph using the CSR format [84]. They achieve load balancing across thread blocks and within a thread block for single-source shortest path problems. Gunrock presents a generalized graph traversal API and combines the work from Merrill and Davidson for a load balancing strategy for their framework [384]. Brahmakshatriya et al. [55] provide a good overview of load balancing schemes for graph processing frameworks and introduce an edge-based variant of "CTA+Warp+Scan". Their framework GraphIt supports seven different load balancing schemes.

The area of sparse matrices is closely related to graph processing. The AdELL sparse matrix format is designed for highly skewed sparse matrices and provides functionality to apply a heuristic that balances the workload over the threads of a warp [239]. Ashari et al. present the adaptive CSR format ACSR for sparse vector-matrix multiplication [35]. They place matrix rows in different bins organized by the number of nonzeros and launch different kernels based on the number of nonzeros. Muhammed et al. present the SURAA format based on CSR where they add a permutation array in which they sort and group rows with the same number of nonzeros [266]. Khorasani et al. abstract from sparse formats and the typical algorithms and generalize it to "nested parallel patterns", a set of coarse-grained tasks where each has a set of fine-grained tasks [185]. They propose Collaborative-Task-Engagement that aims at improving the performance of threads that perform coarse-grained tasks that can be split up in more fine-grained tasks. They use shared memory to register the fine-grained tasks and schedule the tasks avoiding warp inefficiencies. This technique is also used by others [126].

Similar to reducing branch divergence, there are also solutions that sort the data such that threads are likely to have equal work [4, 401]. Other solutions involve the work from Plauth et al. that solves the N-queens problem where each thread searches for a solution in a tree of queen-configurations [306]. Because at deeper levels, the number of correct solutions diminishes, they adapt the workload for a thread by adjusting the search depth to compensate. Zhang et al. also focus on the N-queens problem but employ a work-queue per thread block from which threads fetch a task [431]. They compare it with a single work-queue and conclude that there is too much contention and that a work-queue per thread block performs better. Nasre et al. instead use a work-list per thread and store it in shared memory [276].

Within a thread block. Within a thread block, warps can differ in the workload leading to underutilized resources. Hong et al. mentioned above study graph algorithms on GPUs [154] and for

Cycles	Warp 1		Warp 2		Warp 3		Warp 4	
	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7	Thread 8
1	1->2							
2	1->3							
3	1->4							
4	2->1	3->1	4->1					
5	2->5		4->5					
6			4->6		IDLE	IDLE	IDLE	IDLE
7			4->7					
8			4->8					
9	5->2	6->4	7->4	8->4				
10	5->4		7->9					
11	9->7							

(a) Vertices are assigned to threads.

Cycles	Warp 1		Warp 2		Warp 3		Warp 4	
	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7	Thread 8
1	1->2	1->3						
2	1->4							
3	2->1	2->5	3->1		4->1	4->5		
4				IDLE	4->6	4->7		IDLE
5					4->8			
6	5->2	5->4	6->4		7->4	7->9	8->4	
7	9->7							

(b) Vertices are assigned to warps.

Fig. 24. Difference in load balance by assigning work to threads or warps (from Zhu et al. [437], © IEEE).

load imbalance within a thread block they store the tasks in a global worklist from which warps fetch them. Zhu et al. [437] use a similar warp-centric technique (see Section A.3.3). In Figure 24 they show on the left an approach where vertices of a graph are assigned to threads in a breadth-first search program. For example, the edges of vertex 1, 2, 5, and 9 are assigned to thread 1. At the right, they assign the vertices to a warp with which the threads can work on various edges from the same vertex in parallel, resulting in increased load-balance and performance.

Lee et al. [202] analyze graphs a-priori and reorganize the block allocation by splitting and merging blocks. Nisa et al. perform sparse tensor computations and split the fibers assigned to threads to balance the load [281]. Nasre et al. perform work-donation, donating work from a thread to another thread using shared memory [275]. Zhang et al. sort sequences to align to balance the workload among threads in a thread block [430]. Lee et al. [202] show that load balancing within a thread block can lead to impressive speedups. They measure a performance improvement between 40 and 66% depending on the architecture (Pascal 2016, Volta 2017, and Turing 2018) averaged over various graph algorithms.

Among thread blocks. Typically load balancing among thread blocks is not a problem because the GPU programming model expects oversubscription of thread blocks onto the Streaming Multiprocessors. However, for irregular workloads this way of programming may not be the best. The *persistent threads* model deviates from the default of oversubscribed thread blocks with *maximal launch* where programmers allocate the maximum number of threads and blocks that can physically run concurrently on a GPU. The threads will run for the full duration of the kernel and exchange work with queues. Gupta et al. [132] discuss this model extensively and compare it to traditional GPU programming. They note that for some applications it can bring performance but is not beneficial in any situation, as also pointed out by Nasre et al. [275]. The main advantage of this model is load balancing and it makes inter-block synchronization more efficient, although it requires the techniques discussed in Section A.3.10.

Several authors employ the persistent threads model [177, 399] often for irregular workloads [46, 155, 275]. Tzengy et al. divide the work into bins for a ray-tracing application [360] and employ work-stealing and -donation which means that as soon as a bin is overflowing with work, the thread block donates tasks to another streaming multi-processor in a round-robin fashion.

Another load balancing technique among thread blocks is found in the domain of sparse tensor computations. Nisa et al. may encounter a thread block that dominates the others [281]. They extend Ashari's binning technique [35] (discussed above in *Within warps*) for balancing thread blocks.

Between CPU/GPU. In applications that both use GPUs and the CPU (see Section A.4.2), it may be necessary to balance the load between these two types of computing devices. Often authors

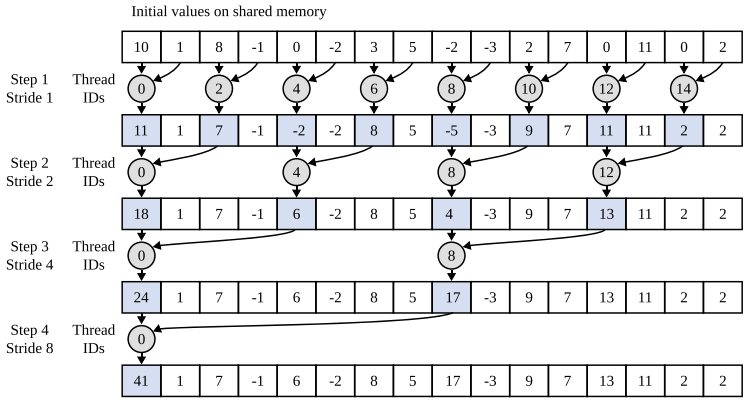


Fig. 25. A simple reduction kernel using shared memory. Between each step it is required to synchronize the threads for correctness (adapted from Phuong et al. [305], © Central South University Press and Springer-Verlag GmbH Germany, part of Springer Nature 2017).

simply decide for a static partition, where a predetermined percentage of the total amount of work is offloaded to the GPU, and the rest is left on the CPU [72, 355, 356].

However, this solution is not easily scalable and there are various examples of more general solutions to this scheduling problem. Wan et al. dynamically determine the best ratio of tasks based on feedback about execution times [377]. This works if the execution times are predictable, but for irregular applications, such as sparse matrix-matrix multiplication this may not work. Matam et al. show several heuristics to predict a good balance between the ratio of CPU/GPU computation [249]. Wang et al. learn a good ratio between workloads with the Support Vector Machine algorithm [385].

A.3.8 Synchronization-related Balancing—Reduce Synchronization. Parallel algorithms rely on synchronization primitives for a variety of reasons. Common among these reasons is ensuring that all threads have reached a certain section of code, or that memory operations have completed and updates to memory are thus available to all threads. In Figure 25 Phuong et al. [305] visualize a simple reduction using shared memory. Threads in a thread block read two values from shared memory, add those values and write back the result to shared memory. Since the threads act in parallel, it is required to add a barrier between each of the steps.

However, synchronization can have a considerable impact on performance, especially when there are not enough threads, and not enough work, to hide the latency caused by waiting idle. Therefore, reducing synchronization can be seen as a performance optimization, however one that often depends on algorithmic changes to avoid synchronization without affecting correctness [97, 432]. An interesting example of this is provided by Petre et al. [303] using a version of the Cooley-Tukey FFT algorithm that does not require synchronization. Other interesting work is by Lin et al. [218] which replaces shared memory barriers with custom synchronization to improve on branch divergence in nested branches.

It was somewhat common in CUDA programs to reduce explicit block-level synchronization by using implicit warp-level synchronization [7, 13, 232, 258, 305, 393]. However this has been deprecated since CUDA 9.0, and programmers are advised to use explicit warp-level synchronization primitives instead. Agarwal et al. [13] also note that without this implicit synchronization their performance would be significantly worse. Implicit synchronization is also available in OpenCL using large vectors [43]. A rather similar approach is to use the implicit synchronization provided by multiple kernel executions to get rid of in-kernel synchronization, as shown by Mawson et al. [252].

Another technique used to reduce synchronization is to increase the amount of work per thread, therefore reducing the need for synchronization among threads [79, 202, 280, 345, 405, 424] (see Section A.3.4). Other optimizations can be used to achieve this goal, such as *use shared memory* [345, 424] and *blocking* [32, 79, 280]. In particular, Reddy et al. [311] achieve the result of avoiding synchronization by completely unrolling a tree-based reduction algorithm.

It is also interesting to point out how there can be conflicting applications of reduce synchronization. As an example, on one side we have Bauer et al. [45] reducing synchronization overhead by replacing barriers with more fine-grained primitives, and on the other side, we have Nasre et al. [274, 275] replacing fine-grained synchronization with barriers. Nasre et al. [274] provide an indication of the potential speedup by reducing atomic instructions: In various graph algorithms, depending on the architecture (Fermi 2011 and Kepler 2012), the graph algorithm, and the input they measure a speedup of between 3% and 50%.

A.3.9 Synchronization-related Balancing—Reduce Atomics. Atomic operations allow programmers to perform memory updates with the guarantee that there will be no conflicting results. This guarantee is particularly important for GPUs where there can be thousands of threads running in parallel potentially operating on the same memory locations. However, although important for the correctness of many algorithms, atomics introduce computational overhead because of serialization: if two or more atomic operations access the same location, only one at a time can be executed, while the others have to wait. It is then clear that the higher the number of conflicts is, the higher the performance penalty is.

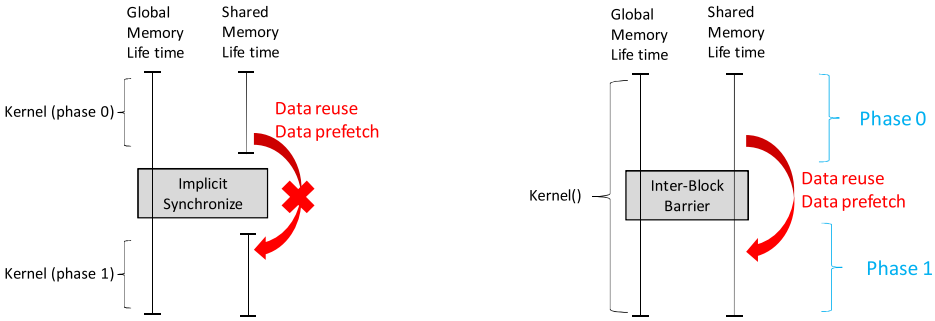
Therefore GPU programming literature consists of two common optimizations: *avoid atomics* and *reduce atomics*. Although the two optimizations are similar, with *avoid atomics* we refer to the goal of replacing all atomics in the code with something else, while with *reduce atomics* we refer to the goal of replacing only part of them, while keeping the essential ones.

Avoid Atomics. There is no single way to avoid using atomics, and most of the techniques to implement avoid atomics are application-specific. For example, Leist et all. [206] achieve this by simply replacing a counter with a boolean flag, while Vetter et al. [370] have to perform some preprocessing to achieve the same result.

A similar approach is proposed by Cecka et al. [66], and it consists of applying an application-specific partitioning of the data such that it avoids conflicts allowing the authors to avoid atomics. There are even cases where the way to avoid atomics is to relax some constraints and maintain an approximate view of the status of a graph [255], or cases where it is necessary to relax the application's memory model [65]. By enforcing ordering on when threads see the results of memory operations with the use of barriers, Burtscher et al. [59] present another implementation of this optimization.

Cecilia et al. [64] manage to avoid atomics by applying another optimization such as the *scatter-to-gather* transformation, while Soman et al. [334] rely on intrinsic properties of their application to achieve the same result. Střelák et al. [338] avoid atomics by using a fine-grained parallelization strategy. Finally, a widely known example of avoid atomics is the lock-free implementation of Xian et al.'s *inter-block synchronization* strategy [400] that does not rely on atomics.

Reduce Atomics. Similarly to the previous optimizations, there are multiple ways to implement *reduce atomics*, and most of them are application-specific. Pai et al. [298], for example, aggregate push operations on a graph to reduce the total number of updates necessary, and therefore reduce the atomic operations, while Leist et al. [206] reduce the number of atomics operating on global memory atomics by introducing atomic operations in shared memory.



(a) Prefetching is not possible because of separate kernel launches that implicitly synchronize global memory.

(b) By means of an inter-block barrier it is possible to prefetch data.

Fig. 26. Prefetching data into shared memory with inter-block synchronization (from Wu et al. [396], © IEEE).

Having threads collaborate at the block or warp level to reduce the number of updates to global memory is another technique that is used to reduce atomics. Gaihre et al. [115] reduce the number of atomics to just one per warp, at the cost of more intra-warp collaboration and the use of *shuffle instructions* (see Section A.1.2). A similar technique is employed by Anzt et al. [28]. Gaihre et al. gain an average performance improvement of 35% over various datasets in their breadth-first-search implementation making use of a Pascal GPU (2016). Others show that atomics can be reduced by using third-party libraries [77, 208].

Samadi et al. [321] selectively reduce atomics, by removing only the instances with the highest degree of possible conflict, while leaving the others in the code. It is also possible that another independent parameter influences the amount of atomic operations in the code, as it is the case for Ashari et al. [34], by controlling the amount of work per thread block. Anzt et al. reduce atomic collisions between threads by sorting the data and reduce the chance of threads performing an atomic update at the same time [28].

A.3.10 Synchronization-related Balancing—Inter-Block Synchronization. Both CUDA and OpenCL provide a way to easily synchronize threads in the same thread block by means of a barrier. However, historically the only way for synchronization at a scope larger than the thread block is via multiple kernel calls. To overcome the overhead associated with multiple calls, researchers have proposed various methods to implement *inter-block synchronization*, and because this technique can be used to improve performance, we consider it as an optimization. As an example, Wu et al. use inter-block synchronization to prefetch data in shared memory (see Section A.1.9) for after the global barrier, something not possible with only kernel calls [396]. In Figure 26 they show on the left, that it is not possible to prefetch data into shared memory because global memory is synchronized implictly with a second kernel call. On the right, they show that with an inter-block barrier they can prefetch data in shared memory to be used in the second phase.

The oldest reference to this optimization can be found in [375], where, in the context of micro-benchmarks to investigate the performance of linear algebra kernels, Volkov and Demmel introduce a global barrier, implemented using atomic operations in global memory. Nevertheless, the two implementations of a global barrier introduced by Xiao et al. [400], one using atomic operations and one lock-free, are the most common use of this optimization in the literature [18, 77, 132, 236, 397]. Xiao et al. show a performance improvement of the lock-free technique

between 11 and 66% depending on the benchmark on a Tesla GPU (2008). Overall, all of these implementations of *inter-block synchronization*, including similar ones found in [48, 74, 113], are based on having a synchronization data structure in global memory that all thread blocks can access and modify.

NVIDIA recognized the usefulness of inter-block synchronization and more fine-grained synchronization and released in 2017 a new version of CUDA (9.0) that supports cooperative groups. Cooperative groups allow programmers to define a group of threads for synchronization, shuffle, and voting functions (see Section A.1.2). It allows to define groups smaller than warps for barrier synchronization, it allows to define a group for thread blocks, equivalent to the traditional thread block barrier, and it allows to define groups larger than thread blocks, potentially the grid of thread blocks or even multiple grids for multiple GPUs. This advanced inter-block-synchronization requires architectural support provided by the Pascal (2016), Volta (2017) and later architectures.

A.4 Host Interaction

A.4.1 Host Communication. The predominant model for GPU computing is the CPU as the host and the GPU as a PCI-express card as an accelerator. This makes it necessary to copy data back and forth between the host memory and the GPU memory, the device memory over the PCI-express bus. The PCI-express bus has limited bandwidth compared to the bandwidth between device memory and the compute units and in many cases, it is necessary to optimize the host communication to increase the overall application performance.

Communicating between host and device can essentially happen in two ways: with *implicit* transfers and *explicit* transfers. Implicit transfers occur when page-locked or pinned memory has been allocated on the host but is memory mapped into the address space of the device. Transfers over the PCI-e bus still occur but overlap with the executing kernels. This technique is sometimes called zero-copy [155, 297, 300].

A more sophisticated version of implicit transfers is Unified Memory (sometimes called managed memory) where devices (GPUs or the host) acquire and exchange pages on demand. This both simplifies code [142, 271] and has potential for speed up [271], although Lu et al. [235] do not measure performance improvement most likely caused by the memory access pattern. Unified Memory can give performance benefits in pointer-heavy code, providing access to data elements directly via the pointers because of exchanged pages, whereas zero-copy access will result in PCIe transfers for each data element via a pointer [142].

It is also possible to explicitly transfer data from the host to the device in the host program. Using pinned memory makes it possible to overlap these transfers with kernel execution. Often this happens in the CUDA abstraction "streams" or OpenCL "command queues". Kernel executions and data transfers are *commands* that are associated with a stream or enqueued in an OpenCL queue where the streams or queues are independent of each other. Each command returns an event and synchronization between the streams or queues is possible with those events. In OpenCL queues can be in-order, equivalent to CUDA streams, but also out of order, which requires synchronization using events [343]. Van Werkhoven et al. [392] show in Figure 27 the performance benefit that extra streams can provide. Transfers can be overlapped with GPU computation resulting in increased performance.

The most effective optimization is eliminating communication altogether by executing all of the algorithm on the GPU [59, 95, 138] or perform as much as possible on the GPU or keeping as much as possible data on the GPU [29, 40, 88, 165, 173, 223]. Another approach is to compress the data that is to be transferred over the PCI-e bus [53, 279]. Domínguez et al. [95] provide an indication of what kinds of speedups can be expected of this optimization. On a Tesla (2009) and

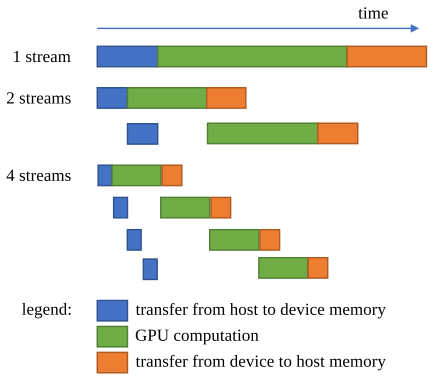


Fig. 27. Using streams to overlap transfers and computations (adapted from Van Werkhoven et al. [392], © IEEE).

Fermi (2010) GPU, they report a performance improvement of 28.9 and 18.9% respectively, applying the technique on a meshfree computational fluid dynamics application.

Li et al. have an interesting approach. They apply dynamic parallelism (where threads are able to launch kernels as well) to transfer CPU computations to the GPU [210]. From the host, they launch a kernel with one thread that will call the other kernels, essentially moving the control loop calling kernels to the GPU. They require to tweak the synchronization overhead as a result of this scheme.

Hong et al. also have an interesting approach. They use an implementation of **Asymmetric Distributed Shared Memory** (**ADSM**) called **Global Memory for Accelerators** (**GMAC**) on top of OpenCL with which data transfers are simplified and sped up [153]. It provides users a unified memory view on the CPU and GPU memory in OpenCL, something that CUDA also provides.

Many improvements for host communication are targeted at how transfers are organized. Many simply use pinned memory to speed up the transfers [106, 128, 136, 205, 211, 235, 326, 335, 362]. Others use pinned and/or mapped memory to also overlap transfers with computations [29, 40, 197, 213, 250, 377]. More advanced overlapping schemes are possible with using streams or command queues [12, 70, 92, 98, 235, 237, 362, 419], called "streamlizing" by Ma et al. [237].

Streams are used in various schemes, for example with multiple threads, each thread a stream [338], one stream for the transfers, one for kernel execution [171], or streams for disk to host transfer, host-to-device, and kernel execution [73].

In these more elaborate schemes it is also necessary to manage the buffers properly. Some apply double buffering [195, 249, 364] or even triple buffering [367] with three threads and three streams for host-to-device communication, device-to-host communication, and kernel execution. Hu et al. use a buffer that is tuned such that loading and computation is perfectly overlapped [160].

With these techniques pipelining is possible, which is especially beneficial for bandwidth-limited kernels [79]. For example, Zhu et al. have a double buffer, and a pipeline with three buffers, two command queues, one for transferring data, one for kernel execution and synchronization is achieved with events [437]. Lee et al. use streams and break up tasks into smaller ones to overlap the transfers [204]. Many more use pipelining [73, 128, 174, 253, 316] but Zhang et al. have an interesting application for it [426, 427]. They have a pipelining scheme to hide the latency of "thread-data remapping", discussed in Section A.2.2.

A.4.2 CPU/GPU Computation. GPUs are, almost always, used as part of a system that includes one or more CPUs, host memory, interconnection to other systems, and I/O. It is therefore left

to developers to decide on which computational device each part of an application should run, and although often the choice is to run either on the CPU or the GPU, this is not always true. This optimization technique consists of splitting the computation between CPU and GPU, using both devices to perform useful work [72, 112, 406]. An indication of performance is given by Che et al. [72] who measure a performance improvement of 15 and 20% depending on the data layout of a k-means application measured on a Fermi (2010) GPU.

A common use-case for this optimization is when an application can be divided in a set of independent or semi-independent tasks [23, 102, 355, 356]. If there are dependencies between these tasks, optimizing the data transfers between CPU and GPU becomes important for performance, as exemplified by Fan et al. [102]. Another example of this technique is to leave the execution of non performance-critical sections of code to the CPU [388].

When the choice about how to split the computation between CPU and GPU is not obvious, a critical task is to find some balance between the two devices as was discussed in the section about load-balancing (Section A.3.7). To conclude, it is also worth mentioning that this optimization technique does not always lead to improved performance [40, 95].

B METHODOLOGY

This appendix extends Section 3 with more details on the selection process and how articles were processed.

B.1 Phase 1, based on Title, Venue, and Keywords

For *phase 1* we took a set of articles on the topic of GPU optimizations that we as authors knew well. We gathered the author keywords from these articles and extracted a set of keywords with which we bootstrapped our query for the Scopus database to find articles on optimizations.

The keywords that we used in the query for the Scopus database are listed below. If necessary, we provided context for a keyword by appending it with `AND KEY(gpu)` and other GPU-related terms. Each line is a separate query that is used in combination with the fixed query:

```
{} AND SUBJAREA(COMP OR ENGI) AND DOCTYPE(ar OR cp) AND (PUBYEAR > 2005)
```

where `{}` is replaced with one of those lines. This helps us to track the effectiveness of the separate keywords.

```
KEY(gpu) AND KEY(optimization*)
KEY(graphics processing unit) AND KEY(optimization*)
KEY(gpgpu) AND KEY(optimization*)
KEY(cuda) AND KEY(optimization*)
KEY(opencl) AND KEY(optimization*)
KEY(many*core) AND KEY(optimization*)
KEY(control flow divergence)
KEY(warp-synchronous programming)
KEY(branch divergence)
KEY(warp execution)
KEY(thread-data remapping)
KEY(memory coalescing)
KEY(coalesced memory accesses)
KEY(divergence) AND KEY(gpu)
KEY(divergence) AND KEY(graphics processing unit)
KEY(divergence) AND KEY(gpgpu)
KEY(divergence) AND KEY(cuda)
KEY(divergence) AND KEY(opencl)
KEY(divergence) AND KEY(many*core)
KEY(kernel fusion) AND KEY(gpu)
KEY(kernel fusion) AND KEY(graphics processing unit)
KEY(kernel fusion) AND KEY(gpgpu)
KEY(kernel fusion) AND KEY(cuda)
KEY(kernel fusion) AND KEY(opencl)
KEY(kernel fusion) AND KEY(many*core)
```

```
KEY(fusion) AND KEY(gpu)
KEY(fusion) AND KEY(graphics processing unit)
KEY(fusion) AND KEY(gpgpu)
KEY(fusion) AND KEY(cuda)
KEY(fusion) AND KEY(opencl)
KEY(fusion) AND KEY(many*core)
KEY(irregular algorithms) AND KEY(gpu)
KEY(irregular algorithms) AND KEY(graphics processing unit)
KEY(irregular algorithms) AND KEY(gpgpu)
KEY(irregular algorithms) AND KEY(cuda)
KEY(irregular algorithms) AND KEY(opencl)
KEY(irregular algorithms) AND KEY(many*core)
KEY(irregular computations) AND KEY(gpu)
KEY(irregular computations) AND KEY(graphics processing unit)
KEY(irregular computations) AND KEY(gpgpu)
KEY(irregular computations) AND KEY(cuda)
KEY(irregular computations) AND KEY(opencl)
KEY(irregular computations) AND KEY(many*core)
KEY(data transformation) AND KEY(gpu)
KEY(data transformation) AND KEY(graphics processing unit)
KEY(data transformation) AND KEY(gpgpu)
KEY(data transformation) AND KEY(cuda)
KEY(data transformation) AND KEY(opencl)
KEY(data transformation) AND KEY(many*core)
KEY(graph representation) AND KEY(gpu)
KEY(graph representation) AND KEY(graphics processing unit)
KEY(graph representation) AND KEY(gpgpu)
KEY(graph representation) AND KEY(cuda)
KEY(graph representation) AND KEY(opencl)
KEY(graph representation) AND KEY(many*core)
```

The query was performed on 29 November 2019 and repeated on 27 May 2021 to update the database with the newly published articles. The query resulted in 3,973 articles that we sorted descending by year and within a year descending by a number of citations. We did this to have a first impression on the importance of the articles where we assume that the number of citations within a year is comparable and that the number of citations is a reasonable metric for the impact of a article. These articles were presented with title, a number of citations, year, venue, and keywords. Articles were selected when it was apparent that some form of GPU optimization was performed. So, for example, articles that merely ported an application to the GPU were discarded. Based on this information we discarded 2,853 articles and 1,120 were selected.

B.2 Phase 2, based on Abstract and Authors

In *phase 2* we used the same sorting and presentation as in phase 1 but added the author list and the abstract. Based on this information we then either *selected* articles, *discarded* articles, or marked them as *auxiliary*. An example of an auxiliary article is a relevant survey or articles that analyze GPUs for security properties. The selection of a article requires some evidence of performing a GPU optimization. In this phase, we discarded 386 articles, selected 532 articles and marked 202 articles as auxiliary. The auxiliary articles were reviewed again and split in relevant categories and 10 of these articles were selected, resulting in 542 selected articles.

B.3 Phase 3, Scanning the Articles

In this phase we "scan" the articles, which means that we read the title, authors, venue, keywords, and the abstract. We then inspect the tables, figures, and the headings of sections. Based on this information we decide on the inclusion and exclusion criteria which are the following:

Inclusion criteria.

(1) Introducing optimizations for a GPU kernel
(2) Introducing optimizations for multiple GPU kernels (fusion)

(3) Introducing optimizations for heterogeneous CPU-GPU
(4) Introducing optimizations for CPU-GPU data communication
(5) Apply auto-tuning on GPU kernels
(6) Applying optimizations directly on GPUs
(7) Measuring the effects of the optimizations

Exclusion criteria.

(1) Introducing/applying optimizations for multi-GPU cluster
(2) Optimizations in hardware (warp schedulers, and so on.)
(3) Compiler optimizations not available for programmers
(4) Optimizations on the algorithmic level

The inclusion criteria select optimizations for kernels or optimizations that involve multiple kernels such as fusion. Inclusion criteria (3) and (4) also involve optimizations on the host level and the last three select articles that do not necessarily introduce or discuss optimizations, but only apply existing optimizations possibly with auto-tuning. The exclusion criteria filter out optimizations for GPU clusters, optimizations in hardware or optimizations performed by compilers in such a way that these optimizations cannot be performed manually, for example polyhedral analysis. The last exclusion criterion filters out optimizations that only consist of changing the algorithm. Based on these criteria, we either select or discard articles and from this selection process we selected 401 articles and discarded 141.

B.4 Frequently Cited Articles

We used the Scopus library to find all articles that this selection of 401 articles cited. From these cited articles we gathered the ones that were not already in our set and that were cited more than 5 times by articles within our set. Since our selection of articles cited these articles frequently, we deemed the articles important enough to be incorporated in the overall selection process. This resulted in an additional 149 articles on which we applied the selection process from *phase 2*. This resulted in 63 articles. We then applied the selection process from *phase 3* which resulted in 49 articles.

B.5 Phase 4, Extracting the Optimizations

In this phase, we analyze the 450 articles (401 articles from *phase 3* and the 49 frequently cited articles) and extract the optimizations, explicitly mentioned performance bottlenecks, and the GPUs used. If we find an optimization in the article, we store the optimization under a specific key, such as “blocking” or “coalesced-access” together with a key for the article and we store a quote from the article that shows that this optimization is indeed in the article. The GPUs in the article are also stored under a key, such as “titanx” together with the key of the article and we do the same for the performance bottlenecks.

B.6 Phase 5, Describing the Optimizations

In this phase we cleaned the database with optimizations, GPUs, and performance bottlenecks. We sorted the optimizations and bottlenecks based on the number of articles. Our selection of optimizations contained a large tail with few articles and because of time and space restrictions we cut off the tail at the optimization with 20 articles or less.

We described the various optimization techniques that we captured in sections that cover a topic, such as “Use dedicated memories” that are based on various optimizations, such as “use shared mem”, “use constant mem”, “use texture mem”. These sections are listed in Appendix A

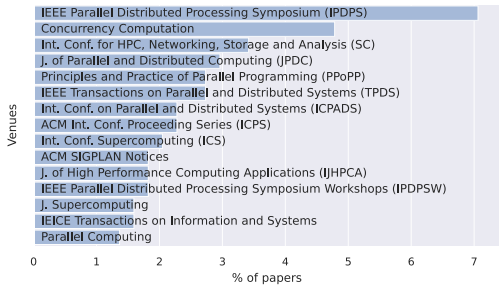


Fig. 28. No. of articles per publication source.

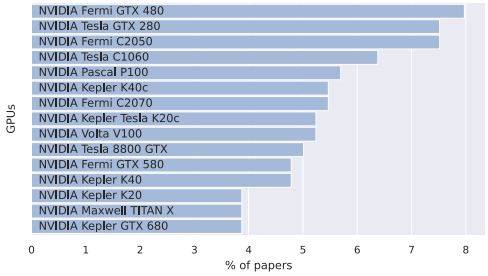


Fig. 29. No. of articles per GPU.

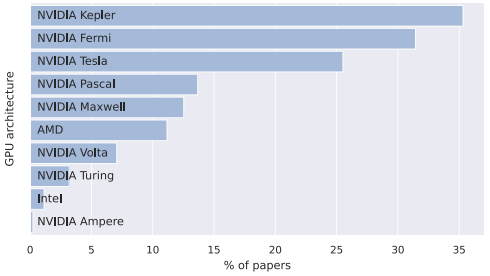


Fig. 30. No. of articles per GPU architecture.

and essentially form a reference resource for readers that want to know more details on a specific optimization technique.

C DATA ANALYSIS

This appendix extends Section 4 with more graphs and details. Scopus reports the publication source and Figure 28 shows a number of publications per source. IPDPS, SC, and PPoPP are the three most popular conferences for research into GPU optimizations, while CPE, TPDS, and JPDC are the three most popular journals. The distribution has a long tail: there is a total of 188 unique venues, many having only one or two articles from our dataset. This indicates that research into GPU optimizations is ubiquitous and not necessarily specific to one certain community or publication venue.

For each article, we manually extracted the GPUs used during evaluation since this is a good indicator for the type of architecture targeted by the authors. The number of times each GPU is mentioned across all articles is shown in Figure 29, note that one article can mention multiple GPUs. It is evident from this figure that NVIDIA is by far the most popular vendor for GPUs. Figure 30 shows the number of articles per architecture. NVIDIA GPUs make up the fast majority of GPUs used for benchmarking, while AMD GPUs are used in slightly more than 10% of the articles.

Figure 4 on page 5 shows the number of times GPU architectures are mentioned by articles for each year. We see that there is a clear rise and fall of each architecture throughout the years. For instance, the Fermi architecture was released in 2010. There is one article targeting Fermi in 2010, this number increases per year and reaches a peak in 2013 when 73% of articles use Fermi, after which it decreases per year. GPUs have a long lifetime. For instance, even as late as 2019, Kepler GPUs were still the most common architecture even though the architecture had been released

7 years earlier. In 2021, the most common architectures are Pascal, Volta, and Maxwell, despite their respective release years of 2016, 2017, and 2014.

Figure 3 on page 4 shows, for each optimization, the percentage of articles that mention that specific technique. *Coalesced access* and *use dedicated memories* are the most popular, which is expected since they have a crucial impact on the performance of GPUs. *Reduce branch divergence* and *loop unrolling* are also highly popular because control flow also plays an important role in performance analysis. *Auto-tuning* is also very common, with over 1/8th of the articles applying automatic parameter tuning in one way or the other.

D BALANCING PARAMETERS FOR PERFORMANCE

This appendix extends Section 5.4 with a simple model presented by Volkov [373] that captures the meaning of a properly utilized GPU kernel. The model takes the viewpoint of a single SM and it is assumed that the number of thread blocks is high enough for overall GPU performance.

Often, the term *occupancy* is used to describe the utilization of a GPU. Occupancy refers to the number of active warps divided by the maximum number of warps that an SM supports. The general believe is that if you choose the number of registers, shared memory, and/or thread block size such that the occupancy is low, it is not possible to achieve high performance. However, Volkov [373] states that occupancy as a measure of utilization only counts for TLP, whereas it is also important to take *instruction-level parallelism* and BLP, vector instructions, or data types into account. Moreover, they state that low occupancy is even required for certain kernels to achieve high-performance.

In later work, Volkov discusses the importance of the instructions in the instruction stream, shows that many analytical performance models lack the concept of ILP, and presents a detailed and complex analytical performance model that does take ILP into account [374]. In this section, we present Volkov's simplified model of utilization [373] which is sufficient to explain the importance of balancing parameters.

In the model we use the following parameters:

latency the latency of a memory or arithmetic instruction, measured in *cycles*
throughput the throughput of instructions measured in *#cores / SM*
parallelism measured in *#ops / SM*
nrThreads *#threads* per block, defined by the programmer
activeTBs *#active thread blocks* per SM
ILP *#independent instructions* in the instruction stream
BLP the vector width

Given these parameters, we will look at the most important instructions in the instruction stream, for example, single-precision floating point operations, and we look up or determine with a benchmark the *latency* of these operations. We then determine the throughput of these instructions by looking at the number of computational units for these instructions (for example, our GPU in Figure 5 on page 5 has a throughput of 64 cores per SM for single-precision floating point operations). Finally, we determine the minimum amount of parallelism we require by applying Little's law [222]:

$$parallelism_{min} = latency \times throughput. \tag{1}$$

In this equation latency is measured in *cycles*, throughput is measured in *#cores / SM*, and parallelism in *#ops / SM* where $\#ops = \#cores \times \#cycles$.

The TLP is determined by the number of active thread blocks that can run on an SM and as said above, the specific number for *activeTBs* is dependent on the thread block size, the number of registers per thread, and the amount of shared memory allocated per thread block. The equation

for TLP is

$$TLP = activeTBs \times nrThreads. \tag{2}$$

In this equation, each active thread is assumed to be able to carry out 1 operation, so *TLP* is measured in *#ops / SM*.

Given the above, utilization is then defined as follows:

$$utilization = min\left(1, \frac{TLP \times ILP \times BLP}{parallelism_{min}}\right). \tag{3}$$

Utilization is a ratio between 0 and 1. The fraction inside the minimum function can become larger than one, but then it means that the kernel is overprovisioned in terms of parallelism to overcome the latency of the instructions.

An example, from Volkov [373]: On a GTX480, latency is about 18 cycles for a single-precision operation, the throughput is 48 cores per SM, so the minimum parallelism is 864 operations per SM. Without independent instructions and bit level parallelism 864 active threads per SM are required to reach 100% utilization, whereas with two independent instructions in the instruction stream, only 432 active threads would be required, and with four independent instructions, only 216 active threads, and so on.

The same applies for bandwidth for which bit-level parallelism can be used. So, if you are able to reach peak bandwidth with a certain number of threads n while loading 32-bit `float` elements, then you require $n/2$ threads if you load `float2` elements.

Given the above, it is clear that achieving high-performance is often a balancing act, where TLP and ILP are to some extent communicating vessels but with buckets in between so that the result is not always immediately clear unless one of the buckets overflows. For example, increasing instruction-level parallelism is often beneficial until you hit a resource limit on the SM, for example the number of registers decreasing the number of active blocks and therefore TLP. Figure 8 on page 24 shows how all these terms influence each other.

ACKNOWLEDGMENT

We would like to thank Daan Siepelinga who performed a literature study in his Master's program that formed the inspiration of this survey.

REFERENCES

[1] 2018. Frontier: OLCF's Exascale Future. Retrieved July 2021 from https://www.olcf.ornl.gov/2018/02/13/frontier-olcfs-exascale-future/.

[2] 2019. U.S. Department of Energy and Intel to Deliver First Exascale Supercomputer, Argonne National Laboratory. Retrieved July 2021 from https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer.

[3] 2020. May We Introduce: LUMI. Retrieved July 2021 from https://www.lumi-supercomputer.eu/may-we-introduce-lumi/.

[4] A. Abdelfattah, A. Haidar, S. Tomov, et al. 2016. On the development of variable size batched computation for heterogeneous parallel architectures. *IEEE International Parallel and Distributed Processing Symposium Workshops* (2016).

[5] A. Abdelfattah, A. Haidar, S. Tomov, et al. 2016. Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on GPUs. *Procedia Computer Science* 80 (2016), 119–130.

[6] A. Abdelfattah, A. Haidar, S. Tomov, et al. 2017. Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs. In*Proceedings of the International Conference on Supercomputing*.

[7] A. Abdelfattah, A. Haidar, S. Tomov, et al. 2018. Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures. *Journal of Computational Science* 26 (2018), 226–236.

[8] A. Abdelfattah, D. Keyes, and H. Ltaief. 2013. Systematic approach in optimizing numerical memory-bound kernels on GPU. *European Conference on Parallel Processing* (2013), 207–216.

[9] A. Abdelfattah, D. Keyes, and H. Ltaief. 2016. KBLAS: An optimized library for dense matrix-vector multiplication on GPU accelerators. *ACM Transactions on Mathematical Software* 42, 3 (2016), 1–31.

[10] A. Abdelfattah, H. Ltaief, D. Keyes, et al. 2016. Performance optimization of sparse matrix-vector multiplication for multi-component PDE-based applications using GPUs. *Concurrency and Computation: Practice and Experience* 28, 12 (2016), 3447–3465.
[11] A. Abdelfattah, S. Tomov, and J. Dongarra. 2019. Progressive optimization of batched LU factorization on GPUs. *IEEE High Performance Extreme Computing Conference* (2019).
[12] A. A. Abdelrahman, M. M. Fouad, H. Dahshan, et al. 2017. High performance CUDA AES implementation: A quantitative performance analysis approach. In *Proceedings of the Computing Conference* (2017).
[13] D. Agarwal, S. Wilf, A. Dhungel, et al. 2012. Acceleration of bilateral filtering algorithm for manycore and multicore architectures. *Proceedings of the International Conference on Parallel Processing.*
[14] K. Aggarwal and U. Bondhugula. 2020. Optimizing the linear fascicle evaluation algorithm for multi-core and many-core systems. *ACM Transactions on Parallel Computing* 7, 4 (2020), 1–45.
[15] H. Ahn and S. Choi. 2017. A novel procedure for implementing a turbo decoder on a GPU with coalesced memory access. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E100.A, 5 (2017), 1188–1196.
[16] M. A. Al-Mouhamed and A. H. Khan. 2017. SpMV and BiCG-Stab optimization for a class of hepta-diagonal-sparse matrices on GPU. *The Journal of Supercomputing* 73, 9 (2017), 3761–3795.
[17] M. A. Al-Mouhamed, A. H. Khan, and N. Mohammad. 2019. A review of CUDA optimization techniques and tools for structured grid computing. *Computing* (2019), 1–27.
[18] H. Ali, G. M. Fathy, Z. Fayez, et al. 2019. Exploring the parallel capabilities of GPU: Berlekamp-Massey algorithm case study. *Cluster Computing* (2019).
[19] S. Alimirzazadeh, E. Jahanbakhsh, A. Maertens, et al. 2018. GPU-accelerated 3-D finite volume particle method. *Computers & Fluids* 171 (2018), 79–93.
[20] AMD. 2017. Radeon's Next-generation Vega Architecture (Whitepaper).
[21] AMD. 2019. Introducing RDNA Architecture (Whitepaper).
[22] AMD. 2020. Introducing CDNA Architecture (Whitepaper).
[23] X. An, H. Jia, and Y. Zhang. 2015. Optimized password recovery for encrypted RAR on GPUs. *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems.*
[24] J. Anantpur and R. Govindarajan. 2014. Taming control divergence in GPUs through control flow linearization. *International Conference on Compiler Construction* (2014), 133–153.
[25] M. Andersch, G. Palmer, R. Krashinsky, et al. 2022. NVIDIA Hopper Architecture In-Depth. Retrieved July 2021 from https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.
[26] P. N. Q. Anh, R. Fan, and Y. Wen. 2015. Reducing vector I/O for faster GPU sparse matrix-vector multiplication. *IEEE International Parallel and Distributed Processing Symposium* (2015).
[27] P. N. Q. Anh, R. Fan, and Y. Wen. 2016. Balanced hashing and efficient GPU sparse general matrix-matrix multiplication. In *Proceedings of the International Conference on Supercomputing.*
[28] H. Anzt, T. Cojean, C. Yen-Chen, et al. 2020. Load-balancing sparse matrix vector product kernels on GPUs. *ACM Transactions on Parallel Computing* 7, 1 (2020), 1–26.
[29] H. Anzt, M. Kreutzer, E. Ponce, et al. 2016. Optimization and performance evaluation of the IDR iterative Krylov solver on GPUs. *The International Journal of High Performance Computing Applications* 32, 2 (2016), 220–230.
[30] H. Anzt, E. Ponce, G. D. Peterson, et al. 2015. GPU-accelerated co-design of induced dimension reduction. In *Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing.*
[31] H. Anzt, W. Sawyer, S. Tomov, et al. 2014. Optimizing krylov subspace solvers on graphics processing units. *IEEE International Parallel & Distributed Processing Symposium Workshops* (2014).
[32] O. Artiles and F. Saeed. 2019. GPU-SFFT: A GPU based parallel algorithm for computing the sparse fast fourier transform (SFFT) of k-sparse signals. *IEEE International Conference on Big Data.*
[33] K. Arumugam, D. Ranjan, M. Zubair, et al. 2016. Memory-efficient parallel simulation of electron beam dynamics using GPUs. In *Proceedings of the IEEE 23rd International Conference on High Performance Computing.*
[34] A. Ashari, N. Sedaghati, J. Eisenlohr, et al. 2014. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs. In *Proceedings of the 28th ACM International Conference on Supercomputing.*
[35] A. Ashari, N. Sedaghati, J. Eisenlohr, et al. 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.*
[36] A. Ashari, N. Sedaghati, J. Eisenlohr, et al. 2015. A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs. *Journal of Parallel and Distributed Computing* 76 (2015), 3–15.
[37] S. Ashkiani, A. Davidson, U. Meyer, et al. 2017. GPU multisplit. *ACM Trans. Parallel Comput.* 4, 1 (2017), 1–44.

[38] S. Ashkiani, M. Farach-Colton, and J. D. Owens. 2018. A dynamic hash table for the GPU. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium.*
[39] T. Athil, R. Christian, and Y. B. Reddy. 2014. CUDA memory techniques for matrix multiplication on quadro 4000. In *Proceedings of the 11th International Conference on Information Technology: New Generations.*
[40] M. Baboulin, J. Dongarra, A. Rémy, et al. 2017. Solving dense symmetric indefinite systems using GPUs. *Concurrency and Computation: Practice and Experience* 29, 9 (2017), e4055.
[41] D. F. Bacon, S. L. Graham, and O. J. Sharp. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys* 26, 4 (1994), 345–420.
[42] D. Bakunas-Milanowski, V. Rego, J. Sang, et al. 2015. A fast parallel selection algorithm on GPUs. In *Proceedings of the International Conference on Computational Science and Computational Intelligence.*
[43] D. Barina, M. Kula, M. Matysek, et al. 2017. Accelerating discrete wavelet transforms on GPUS. *I*n Proceedings of the IEEE International Conference on Image Processing.
[44] J. Barnat, P. Bauch, L. Brim, et al. 2011. Computing strongly connected components in parallel on CUDA. *IEEE International Parallel & Distributed Processing Symposium* (2011).
[45] M. Bauer, H. Cook, and B. Khailany. 2011. CudaDMA optimizing GPU memory bandwidth via warp specialization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.*
[46] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.*
[47] E. Ben-Sasson, M. Hamilis, M. Silberstein, et al. 2016. Fast multiplication in binary fields on GPUs via register cache. In *Proceedings of the International Conference on Supercomputing.*
[48] K. Berger and F. Galea. 2013. An efficient parallelization strategy for dynamic programming on GPU. *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013).
[49] D. J. Bernstein, H. Chen, C. Cheng, et al. 2010. ECC2K-130 on NVIDIA GPUs. *Progress in Cryptology - INDOCRYPT* (2010), 328–346.
[50] C. Bertolli, A. Betts, N. Loriant, et al. 2013. Compiler optimizations for industrial unstructured mesh CFD applications on GPUs. *Languages and Compilers for Parallel Computing* (2013), 112–126.
[51] L. Biferale, F. Mantovani, M. Pivanti, et al. 2013. An optimized D2Q37 Lattice Boltzmann code on GP-GPUs. *Computers & Fluids* 80 (2013), 55–62.
[52] J. W. Bos and D. Stefan. 2010. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. *Cryptographic Hardware and Embedded Systems* (2010), 279–293.
[53] V. Boyer, D. E. Baz, and M. Elkihel. 2012. Solving knapsack problems on GPU. *Computers & Operations Research* 39, 1 (2012), 42–47.
[54] G. van d. Braak, B. Mesman, and H. Corporaal. 2010. Compile-time GPU memory access optimizations. *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation* (2010).
[55] A. Brahmakshatriya, Y. Zhang, C. Hong, et al. 2021. Compiling graph applications for GPU s with GraphIt. *IEEE/ACM International Symposium on Code Generation and Optimization.*
[56] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. 2012. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel Distrib. Comput.* 0 (2012).
[57] P. Bruel, M. Amarís, and A. Goldman. 2017. Autotuning CUDA compiler parameters for heterogeneous applications using the OpenTuner framework. *Concurr Comput* 29, 22 (2017), e3973.
[58] H. M. Bucker, R. Seidler, D. Neuhauser, et al. 2015. The approximate discrete radon transform: A case study in autotuning of OpenCL implementations. *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip* (2015).
[59] M. Burtscher and K. Pingali. 2011. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. *GPU Computing Gems Emerald Edition* (2011), 75–92.
[60] F. Busato and N. Bombieri. 2016. An efficient implementation of the bellman-ford algorithm for kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2016), 2222–2233.
[61] F. Busato, O. Green, N. Bombieri, et al. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. *IEEE High Performance Extreme Computing Conference* (2018).
[62] J. Carabaño, J. Westerholm, and T. Sarjakoski. 2017. A compiler approach to map algebra: Automatic parallelization, locality optimization, and GPU acceleration of raster spatial analysis. *GeoInformatica* 22, 2 (2017), 211–235.
[63] S. Carrillo, J. Siegel, and X. Li. 2009. A control-structure splitting optimization for GPGPU. In *Proceedings of the 6th ACM Conference on Computing Frontiers.*
[64] J. M. Cecilia, J. M. García, A. Nisbet, et al. 2013. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing* 73, 1 (2013), 42–51.
[65] J. M. Cecilia, A. Llanes, J. L. Abellán, et al. 2018. High-throughput ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing* 113 (2018), 261–274.

[66] C. Cecka, A. J. Lew, and E. Darve. 2010. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering* 85, 5 (2010), 640–669.
[67] G. Chakrabarti, V. Grover, B. Aarts, et al. 2012. CUDA: Compiling and optimizing for a GPU platform. *Procedia Computer Science* 9 (2012), 1910–1919.
[68] I. Chakroun, M. Mezmaz, N. Melab, et al. 2012. Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. *Concurr Comput* 25, 8 (2012), 1121–1136.
[69] L. Chang, J. A. Stratton, H. Kim, et al. 2012. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. *International Conference for High Performance Computing, Networking, Storage and Analysis* (2012).
[70] A. Charara, D. Keyes, and H. Ltaief. 2017. A framework for dense triangular matrix kernels on various manycore architectures. *Concurr Comput* 29, 15 (2017), e4187.
[71] A. Chatterjee, S. Radhakrishnan, and J. K. Antonio. 2013. On analyzing large graphs using GPUs. *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013).
[72] S. Che, J. W. Sheaffer, and K. Skadron. 2011. Dymaxion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
[73] C. Chen, C. Wang, J. Hou, et al. 2018. Optimizing data transmission and access of the incremental clustering algorithm using CUDA: A case study. *Journal of Computational Methods in Sciences and Engineering* 18, 4 (2018), 989–1005.
[74] G. Chen and X. Shen. 2015. Free launch. *Proceedings of the 48th International Symposium on Microarchitecture*.
[75] J. Chen, Z. Ju, C. Hua, et al. 2013. Accelerated implementation of adaptive directional lifting-based discrete wavelet transform on GPU. *Signal Process. Image Commun.* 28, 9 (2013), 1202–1211.
[76] X. Chen, D. Z. Chen, and X. S. Hu. 2018. moDNN: Memory optimal DNN training on GPUs. *Design, Automation & Test in Europe Conference & Exhibition*.
[77] X. Chen, P. Li, J. Fang, et al. 2016. Efficient and high-quality sparse graph coloring on GPUs. *Concurr Comput* 29, 10 (2016), e4064.
[78] J. W. Choi, A. Singh, and R. W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
[79] M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. *IEEE International Parallel & Distributed Processing Symposium* (2011).
[80] M. Daga, T. Scogland, and W. Feng. 2011. Architecture-aware mapping and optimization on a 1600-core GPU. *IEEE 17th International Conference on Parallel and Distributed Systems* (2011).
[81] S. Dalton, S. Baxter, D. Merrill, et al. 2015. Optimizing sparse matrix operations on GPUs using merge path. *IEEE International Parallel and Distributed Processing Symposium* (2015).
[82] S. Dalton, L. Olson, and N. Bell. 2015. Optimizing sparse matrix—matrix multiplication for the GPU. *ACM Transactions on Mathematical Software* 41, 4 (2015), 1–20.
[83] K. Datta, M. Murphy, V. Volkov, et al. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *SC - International Conference for High Performance Computing, Networking, Storage and Analysis* (2008).
[84] A. Davidson, S. Baxter, M. Garland, et al. 2014. Work-efficient parallel GPU methods for single-source shortest paths. *IEEE 28th International Parallel and Distributed Processing Symposium* (2014).
[85] A. Deftu and A. Murarasu. 2013. Optimization techniques for dimensionally truncated sparse grids on heterogeneous systems. *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2013).
[86] N. Delbosc, J. L. Summers, A. I. Khan, et al. 2014. Optimized implementation of the lattice boltzmann method on a graphics processing unit towards real-time fluid simulation. *Computers & Mathematics with Applications* 67, 2 (2014), 462–475.
[87] A. Delévacq, P. Delisle, M. Gravel, et al. 2013. Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing* 73, 1 (2013), 52–61.
[88] L. Deng, H. Bai, D. Zhao, et al. 2015. Kepler GPU vs. Xeon Phi: Performance case study with a high-order CFD application. *IEEE International Conference on Computer and Communications* (2015).
[89] Y. Deng, B. D. Wang, and S. Mu. 2009. Taming irregular EDA applications on GPUs. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD* (2009), 539–546.
[90] Y. Djenouri, A. Bendjoudi, Z. Habbas, et al. 2016. Reducing thread divergence in GPU-based bees swarm optimization applied to association rule mining. *Concurr Comput* 29, 9 (2016), e3836.
[91] Y. Djenouri, A. Bendjoudi, M. Mehdi, et al. 2015. Data reordering for minimizing threads divergence in GPU-based evaluating association rules. *Distributed Computing and Artificial Intelligence, 12th International Conference* (2015), 47–54.
[92] Y. Do, H. Kim, P. Oh, et al. 2019. SNU-NPB 2019: Parallelizing and optimizing NPB in OpenCL and CUDA for modern GPUs. *IEEE International Symposium on Workload Characterization (IISWC)* (2019).

[93] M. Doerksen, P. Thulasiraman, and R. K. Thulasiram. 2012. Optimizing option pricing algorithms and profiling power consumption on VLIW APU architecture. *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications* (2012).
[94] E. Domazet, M. Gusev, and S. Ristov. 2016. Optimizing high-performance CUDA DSP filter for ECG signals. *Proceedings of the 27th International DAAAM Symposium* (2016), 0623–0632.
[95] J. M. Domínguez, A. J. C. Crespo, and M. Gómez-Gesteira. 2013. Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method. *Comput. Phys. Commun.* 184, 3 (2013), 617–627.
[96] T. Dong, A. Haidar, P. Luszczek, et al. 2014. LU factorization of small matrices: Accelerating batched DGETRF on the GPU. In *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst* (2014).
[97] Y. Dotsenko, N. K. Govindaraju, P. Sloan, et al. 2008. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd International Conference on Supercomputing*.
[98] C. Du, J. Yuan, J. Dong, et al. 2020. GPU based parallel optimization for real time panoramic video stitching. *Pattern Recognit. Lett.* 133 (2020), 62–69.
[99] S. Dwivedi and A. Heumann. 2020. Profiling and optimization of CT reconstruction on Nvidia Quadro GV100. *IEEE High Performance Extreme Computing Conference* (2020).
[100] A. Dziekonski, A. Lamecki, and M. Mrozowski. 2011. A memory efficient and fast sparse matrix vector product on a GPU. *Progress In Electromagnetics Research* 116 (2011), 49–63.
[101] I. R. Eguly and M. Giles. 2012. Efficient sparse matrix-vector multiplication on cache-based GPUs. *Innovative Parallel Computing, InPar* (2012).
[102] M. Fan, H. Jia, Y. Zhang, et al. 2015. Optimizing image sharpening algorithm on GPU. *44th International Conference on Parallel Processing* (2015).
[103] J. Fang, H. Fu, and G. Yang. 2016. Cache-friendly design for complex spatially-variable coefficient stencils on many-core architectures. *IEEE 23rd International Conference on High Performance Computing* (2016).
[104] J. Fang, H. Fu, H. Zhang, et al. 2015. Optimizing complex spatially-variant coefficient stencils for seismic modeling on GPU. *IEEE 21st International Conference on Parallel and Distributed Systems.* (2015).
[105] J. Fang, A. L. Varbanescu, J. Shen, et al. 2012. Accelerating cost aggregation for real-time stereo matching. *IEEE 18th International Conference on Parallel and Distributed Systems* (2012).
[106] J. Fang, A. L. Varbanescu, and H. Sips. 2011. An auto-tuning solution to data streams clustering in OpenCL. *14th IEEE International Conference on Computational Science and Engineering* (2011).
[107] M. Fang, J. Fang, W. Zhang, et al. 2018. Benchmarking the GPU memory at the warp level. *Parallel Comput.* 71 (2018), 23–41.
[108] F. Feinbube, B. Rabe, M. v. Löwis, et al. 2010. NQueens on CUDA: Optimization issues. *Ninth International Symposium on Parallel and Distributed Computing* (2010).
[109] X. Feng, H. Jin, R. Zheng, et al. 2011. Optimization of sparse matrix-vector multiplication with variant CSR on GPUs. *IEEE 17th International Conference on Parallel and Distributed Systems* (2011).
[110] X. Feng, H. Jin, R. Zheng, et al. 2012. Implementing smith-waterman algorithm with two-dimensional cache on GPUs. *2nd International Conference on Cloud and Green Computing* (2012).
[111] J. Filipovič, M. Madzin, J. Fousek, et al. 2015. Optimizing CUDA code by kernel fusion: Application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.
[112] P. Fortin, M. Gouicem, and S. Graillat. 2016. GPU-accelerated generation of correctly rounded elementary functions. *ACM Transactions on Mathematical Software* 43, 3 (2016), 1–26.
[113] S. Funasaka, K. Nakano, and Y. Ito. 2017. Single kernel soft synchronization technique for task arrays on CUDA-enabled GPUs, with applications. *5th International Symposium on Computing and Networking* (2017).
[114] W. W. L. Fung and T. M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. *IEEE 17th International Symposium on High Performance Computer Architecture* (2011), 25–36.
[115] A. Gaihre, Z. Wu, F. Yao, et al. 2019. XBFS. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*.
[116] H. Gamaarachchi, C. W. Lam, G. Jayatilaka, et al. 2020. GPU accelerated adaptive banded event alignment for rapid comparative nanopore signal analysis. *BMC Bioinformatics* 21, 1 (2020).
[117] J. Gao, Z. Li, R. Liang, et al. 2016. Adaptive optimization l_1-minimization solvers on GPU. *International Journal of Parallel Programming* 45, 3 (2016), 508–529.
[118] L. Gao, F. Zheng, N. Emmart, et al. 2020. DPF-ECC: Accelerating elliptic curve cryptography with floating-point computing power of GPUs. *IEEE International Parallel and Distributed Processing Symposium* (2020).
[119] J. D. Garvey and T. S. Abdelrahman. 2018. A strategy for automatic performance tuning of stencil computations on GPUs. *Sci. Program.* 2018 (2018), 1–24.

[120] M. B. Giles, G. R. Mudalige, Z. Sharif, et al. 2011. Performance analysis and optimization of the OP2 framework on many-core architectures. *The Computer Journal* 55, 2 (2011), 168–180.
[121] J. Gmys, M. Mezmaz, N. Melab, et al. 2016. A GPU-based branch-and-bound algorithm using integer–vector–matrix data structure. *Parallel Comput.* 59 (2016), 119–139.
[122] M. Goldfarb, Y. Jo, and M. Kulkarni. 2013. General transformations for GPU execution of tree traversals. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.*
[123] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, et al. 2012. An optimized approach to histogram computation on GPU. *Machine Vision and Applications* 24, 5 (2012), 899–908.
[124] J. Gómez-Luna, J. M. González-Linares, J. I. B. Benitez, et al. 2013. Performance modeling of atomic additions on GPU scratchpad memory. *IEEE Transactions on Parallel and Distributed Systems* 24, 11 (2013), 2273–2282.
[125] J. Gómez-Luna, I. Sung, L. Chang, et al. 2016. In-place matrix transposition on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2016), 776–788.
[126] B. Goodarzi, F. Khorasani, V. Sarkar, et al. 2019. High performance multilevel graph partitioning on GPU. *International Conference on High Performance Computing & Simulation* (2019).
[127] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, et al. 2008. High performance discrete Fourier transforms on graphics processors. *SC - International Conference for High Performance Computing, Networking, Storage and Analysis* (2008).
[128] M. Gowanlock, C. M. Rude, D. M. Blair, et al. 2017. Clustering throughput optimization on the GPU. *IEEE International Parallel and Distributed Processing Symposium* (2017).
[129] J. L. Greathouse and M. Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis* (2014).
[130] D. Guo, W. Gropp, and L. N. Olson. 2015. A hybrid format for better performance of sparse matrix-vector multiplication on a GPU. *The International Journal of High Performance Computing Applications* 30, 1 (2015), 103–120.
[131] J. Guo, W. Liu, W. Wang, et al. 2019. A GPU memory efficient speed-up scheme for training ultra-deep neural networks. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming.* (2019).
[132] K. Gupta, J. A. Stuart, and J. D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. *Innovative Parallel Computing* (2012).
[133] P. Gwosdek, H. Zimmer, S. Grewenig, et al. 2012. A highly efficient GPU implementation for variational optic flow based on the euler-lagrange framework. *Trends and Topics in Computer Vision* (2012), 372–383.
[134] S. Ha and T. Han. 2013. A scalable work-efficient and depth-optimal parallel scan for the GPGPU environment. *IEEE Transactions on Parallel and Distributed Systems* 24, 12 (2013), 2324–2333.
[135] J. Habich, C. Feichtinger, H. Köstler, et al. 2013. Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results. *Computers & Fluids* 80 (2013), 276–282.
[136] J. Habich, T. Zeiser, G. Hager, et al. 2011. Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA. *Advances in Engineering Software* 42, 5 (2011), 266–272.
[137] B. Hagedorn, A. S. Elliott, H. Barthels, et al. 2020. Fireiron. *Proc. ACM International Conference on Parallel Architectures and Compilation Techniques* (2020).
[138] A. Haidar, A. Abdelfatah, S. Tomov, et al. 2017. High-performance Cholesky factorization for GPU-only execution. *Proceedings of the General Purpose GPUs.*
[139] A. Haidar, T. Dong, P. Luszczek, et al. 2015. Batched matrix computations on hardware accelerators based on GPUs. *The International Journal of High Performance Computing Applications* 29, 2 (2015), 193–208.
[140] T. D. Han and T. S. Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units.*
[141] T. D. Han and T. S. Abdelrahman. 2013. Reducing divergence in GPGPU programs with loop merging. *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units.*
[142] M. Harris. 2013. Unified Memory in CUDA 6. Retrieved June 2021 from https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.
[143] G. He, J. Gao, and J. Wang. 2018. Efficient dense matrix-vector multiplication on GPU. *Concurrency and Computation: Practice and Experience* 30, 19 (2018), e4705.
[144] S. Heldens, P. Hijma, B. van Werkhoven, et al. 2020. The landscape of exascale research: A data-driven literature analysis. *ACM Computing Surveys* 53, 2 (2020), 1–43.
[145] S. Heldens, A. Sclocco, and H. Dreuning. 2019. NLeSC/litstudy.
[146] T. Henriksen, S. Hellfritzsch, P. Sadayappan, et al. 2020. Compiling generalized histograms for GPU. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020).
[147] T. Henriksen, F. Thorøe, M. Elsman, et al. 2019. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming.*
[148] P. Hill, A. Jain, M. Hill, et al. 2017. DeftNN. *Proceedings of the International Symposium on Microarchitecture.*

[149] J. Holewinski, L. 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*.
[150] T. Honda, Y. Ito, and K. Nakano. 2015. A warp-synchronous implementation for multiple-length multiplication on the GPU. *3rd International Symposium on Computing and Networking* (2015).
[151] T. Honda, Y. Ito, and K. Nakano. 2016. GPU-accelerated bulk execution of multiple-length multiplication with warp-synchronous programming technique. *IEICE Transactions on Information and Systems* E99.D, 12 (2016), 3004–3012.
[152] C. Hong, A. Sukumaran-Rajam, J. Kim, et al. 2018. GPU code optimization using abstract kernel emulation and sensitivity analysis. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[153] J. Hong and K. Chung. 2015. Parallel LDPC decoding on a GPU using OpenCL and global memory for accelerators. *IEEE International Conference on Networking, Architecture and Storage* (2015).
[154] S. Hong, S. K. Kim, T. Oguntebi, et al. 2011. Accelerating CUDA graph algorithms at maximum warp. *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*.
[155] N. Hou, F. He, Y. Zhou, et al. 2020. An efficient GPU-based parallel tabu search algorithm for hardware/software co-design. *Frontiers of Computer Science* 14, 5 (2020).
[156] C. Hsieh, L. Vespa, and N. Weng. 2016. A high-throughput DPI engine on GPU via algorithm/implementation co-optimization. *J. Parallel Distrib. Comput.* 88 (2016), 46–56.
[157] C. Hsu, I. Wu, and J. J. Shann. 2014. Dynamic memory optimization and parallelism management for OpenCL. *International Conference on Information Science, Electronics and Electrical Engineering* (2014).
[158] R. Hu, W. Yang, X. Zhou, et al. 2020. Performance analysis and optimization for MTTKRP of sparse tensor on CPU and GPU. *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2020).
[159] X. Hu, J. Xi, and D. Tang. 2020. Optimization for multi-join queries on the GPU. *IEEE Access* 8 (2020), 118380–118395.
[160] Y. Hu, H. Liu, and H. H. Huang. 2018. TriCore: Parallel triangle counting on GPUs. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (2018).
[161] B. Huang, J. Gao, and X. Li. 2009. An empirically optimized radix sort for GPU. *IEEE International Symposium on Parallel and Distributed Processing with Applications* (2009).
[162] J. Huang, C. D. Yu, and R. A. van d. Geijn. 2020. Strassen's algorithm reloaded on GPUs. *ACM Transactions on Mathematical Software* 46, 1 (2020), 1–22.
[163] M. M. Hussain and N. Fujimoto. 2020. GPU-based parallel multi-objective particle swarm optimization for large swarms and high dimensional problems. *Parallel Computing* 92 (2020), 102589.
[164] M. M. Hussain, H. Hattori, and N. Fujimoto. 2016. A CUDA implementation of the standard particle swarm optimization. *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2016).
[165] H. Igarashi, F. Takano, and T. Moriyoshi. 2016. Highly parallel transformation and quantization for HEVC encoder on GPUs. *Visual Communications and Image Processing* (2016).
[166] Y. Ito and K. Nakano. 2013. A GPU implementation of dynamic programming for the optimal polygon triangulation. *IEICE Transactions on Information and Systems* E96.D, 12 (2013), 2596–2603.
[167] Y. Ito, K. Ogawa, and K. Nakano. 2011. Fast ellipse detection algorithm using hough transform on the GPU. *2nd International Conference on Networking and Computing* (2011).
[168] L. M. Itu, C. Suciu, F. Moldoveanu, et al. 2011. GPU optimized computation of stencil based algorithms. *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research* (2011).
[169] B. Jang, S. Do, H. Pien, et al. 2009. Architecture-aware optimization targeting multithreaded stream computing. *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units* (2009).
[170] B. Jang, D. Schaa, P. Mistry, et al. 2011. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1 (2011), 105–118.
[171] W. Jiang, Y. Ma, B. Liu, et al. 2019. Layup. *ACM Transactions on Architecture and Code Optimization* 16, 4 (2019), 1–23.
[172] W. Jiang, H. Mei, F. Lu, et al. 2016. A novel parallel deblocking filtering strategy for HEVC/H.265 based on GPU. *Concurrency and Computation: Practice and Experience* 28, 16 (2016), 4264–4276.
[173] J. Jiménez and J. R. d. Miras. 2013. Optimizations with CUDA: A case study on 3D curve-skeleton extraction from voxelized models. *Communications in Computer and Information Science* (2013), 82–96.
[174] G. Jin, T. Endo, and S. Matsuoka. 2013. A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU. *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013).
[175] J. Jin, S. Lai, S. Hu, et al. 2015. GPUSGD: A GPU-accelerated stochastic gradient descent algorithm for matrix factorization. *Concurrency and Computation: Practice and Experience* 28, 14 (2015), 3844–3865.
[176] M. Jin, H. Fu, Z. Lv, et al. 2016. Graph-oriented code transformation approach for register-limited stencils on GPUs. *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2016).

[177] W. A. R. Jradi, H. A. D. d. Nascimento, and W. S. Martins. 2020. A GPU-based parallel reduction implementation. *Communications in Computer and Information Science* (2020), 168–182.
[178] S. Jun and O. Ha. 2015. Increasing GPU-speedup of volume rendering for images with high complexity. *8th International Conference on u- and e-Service, Science and Technology* (2015).
[179] D. Junger, C. Hundt, and B. Schmidt. 2018. WarpDrive: Massively parallel hashing on multi-GPU nodes. *IEEE International Parallel and Distributed Processing Symposium* (2018).
[180] V. Kelefouras, A. Kritikakou, I. Mporas, et al. 2016. A high-performance matrix–matrix multiplication methodology for CPU and GPU architectures. *The Journal of Supercomputing* 72, 3 (2016), 804–844.
[181] M. Khairy, A. G. Wassal, and M. Zahran. 2019. A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity. *Journal of Parallel and Distributed Computing* 127 (2019), 65–88.
[182] A. u. H. K. Khan, M. Al-Mouhamed, A. Fatayer, et al. 2014. Padding free bank conflict resolution for CUDA-based matrix transpose algorithm. *International Journal of Networked and Distributed Computing* 2, 3 (2014), 124.
[183] F. Khorasani, R. Gupta, and L. N. Bhuyan. 2015. Efficient warp execution in presence of divergence with collaborative context collection. In *Proceedings of the 48th International Symposium on Microarchitecture*.
[184] F. Khorasani, R. Gupta, and L. N. Bhuyan. 2015. Scalable SIMD-efficient graph processing on GPUs. *International Conference on Parallel Architecture and Compilation* (2015).
[185] F. Khorasani, B. Rowe, R. Gupta, et al. 2016. Eliminating intra-warp load imbalance in irregular nested patterns via collaborative task engagement. *IEEE International Parallel and Distributed Processing Symposium* (2016).
[186] F. Khorasani, K. Vora, R. Gupta, et al. 2014. CuSha. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*.
[187] K. Kim, K. Kim, and Q. Park. 2011. Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model. *Computer Physics Communications* 182, 6 (2011), 1201–1207.
[188] M. Korch and T. Werner. 2018. Accelerating explicit ODE methods on GPUs by kernel fusion. *Concurrency and Computation: Practice and Experience* 30, 18 (2018), e4470.
[189] T. Kovac, T. Haber, F. van Reeth, et al. 2019. Improving ODE integration on graphics processing units by reducing thread divergence. *International Conference on Computational Science* (2019), 450–456.
[190] M. Kowarschik and C. Weiß. 2003. *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*. Springer, Berlin, 213–232.
[191] S. Kozacik, P. Fox, J. Humphrey, et al. 2014. Optimization techniques for OpenCL-based linear algebra routines. *Modeling and Simulation for Defense Systems and Applications IX* (2014).
[192] M. Kreutzer, G. Hager, G. Wellein, et al. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423.
[193] M. Krotkiewski and M. Dabrowski. 2013. Efficient 3D stencil computations using CUDA. *Parallel Computing* 39, 10 (2013), 533–548.
[194] M. Kruliš and M. Kratochvíl. 2020. Detailed analysis and optimization of CUDA K-means algorithm. *49th International Conference on Parallel Processing* (2020).
[195] P. Kumar, A. Singhal, S. Mehta, et al. 2013. Real-time moving object detection algorithm on high-resolution videos using GPUs. *Journal of Real-Time Image Processing* 11, 1 (2013), 93–109.
[196] M. Labschutz, S. Bruckner, M. E. Groller, et al. 2016. JiTTree: A just-in-time compiled sparse GPU volume data structure. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 1025–1034.
[197] S. Lade, P. Kulkarni, P. Saraf, et al. 2021. Optimization of ray-tracing algorithm for simulation of PMD sensors. *Machine Learning and Information Processing* (2021), 267–280.
[198] M. E. Lalami and D. El-Baz. 2012. GPU implementation of the branch and bound method for knapsack problems. *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (2012).
[199] Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
[200] D. Lee, I. Dinov, B. Dong, et al. 2012. CUDA optimization strategies for compute- and memory-bound neuroimaging algorithms. *Computer Methods and Programs in Biomedicine* 106, 3 (2012), 175–187.
[201] D. Lee, M. Wolf, and H. Kim. 2010. Design space exploration of the turbo decoding algorithm on GPUs. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*.
[202] J. Lee, S. Kang, Y. Yu, et al. 2020. Optimization of GPU-based sparse matrix multiplication for large sparse networks. *IEEE 36th International Conference on Data Engineering* (2020).
[203] V. W. Lee, P. Hammarlund, R. Singhal, et al. 2010. Debunking the 100X GPU vs. CPU myth. In *Proceedings of the 37th International Symposium on Computer Architecture* (2010).
[204] W. Lee, B. Goi, and R. C. W. Phan. 2018. Terabit encryption in a second: Performance evaluation of block ciphers in GPU with Kepler, Maxwell, and Pascal architectures. *Concurrency and Computation: Practice and Experience* 31, 11 (2018), e5048.

[205] M. Lefebvre, P. Guillen, J. M. L. Gouez, et al. 2012. Optimizing 2D and 3D structured Euler CFD solvers on graphical processing units. *Computers & Fluids* 70 (2012), 136–147.
[206] A. Leist, D. P. Playne, and K. A. Hawick. 2009. Exploiting graphical processing units for data-parallel scientific applications. *Concurrency and Computation: Practice and Experience* 21, 18 (2009), 2400–2437.
[207] H. Li, K. Li, J. An, et al. 2018. MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 29, 7 (2018), 1530–1544.
[208] P. Li, X. Chen, Z. Quan, et al. 2016. High performance parallel graph coloring on GPGPUs. *IEEE International Parallel and Distributed Processing Symposium Workshops* (2016).
[209] Y. Li, J. Dongarra, and S. Tomov. 2009. A note on auto-tuning GEMM for GPUs. *International Conference on Computational Science* (2009), 884–892.
[210] Y. Li, L. Schwiebert, E. Hailat, et al. 2016. Improving performance of GPU code using novel features of the NVIDIA kepler architecture. *Concurrency and Computation: Practice and Experience* 28, 13 (2016), 3586–3605.
[211] Z. Li and Y. Che. 2020. Parallelization and optimization of a combustion simulation application on GPU platform. *Proceedings of the 4th International Conference on High Performance Compilation, Computing and Communications.*
[212] Z. Li, H. Jia, Y. Zhang, et al. 2019. Efficient parallel optimizations of a high-performance SIFT on GPUs. *Journal of Parallel and Distributed Computing* 124 (2019), 78–91.
[213] S. Liang, Y. Liu, C. Wang, et al. 2010. Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU. *IEEE 2nd Symposium on Web Society* (2010).
[214] Y. Liang, W. T. Tang, R. Zhao, et al. 2017. Scale-free sparse matrix-vector multiplication on many-core architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 12 (2017), 2106–2119.
[215] L. Ligowski and W. Rudnicki. 2009. An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. *IEEE International Symposium on Parallel & Distributed Processing* (2009).
[216] C. Lin, A. Cheng, and B. Lai. 2017. A software technique to enhance register utilization of convolutional neural networks on GPGPUs. *International Conference on Applied System Innovation* (2017).
[217] C. Lin, J. Liu, and P. Yang. 2020. Performance enhancement of GPU parallel computing using memory allocation optimization. *14th International Conference on Ubiquitous Information Management and Communication* (2020).
[218] H. Lin and C. Wang. 2020. On-GPU thread-data remapping for nested branch divergence. *Journal of Parallel and Distributed Computing* 139 (2020), 75–86.
[219] H. Lin, C. Wang, and H. Liu. 2018. On-GPU thread-data remapping for branch divergence reduction. *ACM Transactions on Architecture and Code Optimization* 15, 3 (2018), 1–24.
[220] Y. Lin and W. Niu. 2014. High throughput LDPC decoder on GPU. *IEEE Communications Letters* 18, 2 (2014), 344–347.
[221] E. Lindholm, J. Nickolls, S. Oberman, et al. 2008. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (2008), 39–55.
[222] J. D. C. Little. 1961. A proof for the queuing formula. *Operations Research* 9, 3 (1961), 383–387.
[223] B. Liu, C. Wen, A. D. Sarwate, et al. 2017. A unified optimization approach for sparse tensor operations on GPUs. *IEEE International Conference on Cluster Computing* (2017).
[224] H. Liu and H. H. Huang. 2019. SIMD-X: Programming and processing of graph algorithms on GPUs. *USENIX Annual Technical Conference* (2019), 411–428.
[225] H. Liu, H. Kuo, K. Chen, et al. 2013. Memory capacity aware non-blocking data transfer on GPGPU. *IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation* (2013), 395–400.
[226] H. Liu, S. Pai, and A. Jog. 2020. Why GPUs are slow at executing NFAs and how to make them faster. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems* (2020).
[227] J. Liu, W. Ding, O. Jang, et al. 2013. Data layout optimization for GPGPU architectures. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*
[228] S. Liu, J. Tang, Z. Zhang, et al. 2017. Computer architectures for autonomous driving. *Computer* 50, 8 (2017), 18–25.
[229] W. Liu and B. Vinter. 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data. *IEEE 28th International Parallel and Distributed Processing Symposium* (2014).
[230] Y. Liu, D. L. Maskell, and B. Schmidt. 2009. CUDASW++: Optimizing smith-waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes* 2, 1 (2009), 73.
[231] Y. Liu, B. Schmidt, and D. L. Maskell. 2010. CUDASW++2.0: Enhanced smith-waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes* 3, 1 (2010).
[232] P. A. C. Lopes, S. S. Yadav, A. Ilic, et al. 2019. Fast block distributed CUDA implementation of the Hungarian algorithm. *Journal of Parallel and Distributed Computing* 130 (2019), 50–62.
[233] G. Lu, W. Zhang, and Z. Wang. 2020. Optimizing GPU memory transactions for convolution operations. *IEEE International Conference on Cluster Computing* (2020).

[234] Y. Lu, F. Ino, and K. Hagihara. 2016. Cache-aware GPU optimization for out-of-core cone beam CT reconstruction of high-resolution volumes. *IEICE Transactions on Information and Systems* E99.D, 12 (2016), 3060–3071.
[235] Y. Lu, I. Yamazaki, F. Ino, et al. 2020. Reducing the amount of out-of-core data access for GPU-accelerated randomized SVD. *Concurr Comput* 32, 19 (2020).
[236] L. Luo, M. Wong, and W. Hwu. 2010. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*.
[237] A. Ma, J. Cai, Y. Cheng, et al. 2009. Performance optimization strategies of high performance computing on GPU. *International Workshop on Advanced Parallel Processing Technologies* (2009), 150–164.
[238] Y. Ma, J. Li, X. Wu, et al. 2019. Optimizing sparse tensor times matrix on GPUs. *Journal of Parallel and Distributed Computing* 129 (2019), 99–109.
[239] M. Maggioni and T. Berger-Wolf. 2013. AdELL: An adaptive warp-balancing ELL format for efficient sparse matrix-vector multiplication on GPUs. *42nd International Conference on Parallel Processing* (2013).
[240] M. Maggioni and T. Berger-Wolf. 2013. An architecture-aware technique for optimizing sparse matrix-vector multiplication on GPUs. *Procedia Computer Science* 18 (2013), 329–338.
[241] M. Maggioni and T. Berger-Wolf. 2014. CoAdELL: Adaptivity and compression for improving sparse matrix-vector multiplication on GPUs. *IEEE International Parallel & Distributed Processing Symposium Workshops* (2014).
[242] M. Maggioni and T. Berger-Wolf. 2016. Optimization techniques for sparse matrix–vector multiplication on GPUs. *Journal of Parallel and Distributed Computing* 93–94 (2016), 66–86.
[243] A. Magni, C. Dubach, and M. F. O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–11.
[244] A. Magni, C. Dubach, and M. O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 455–466.
[245] D. Man, K. Uda, Y. Ito, et al. 2011. A GPU implementation of computing euclidean distance map with efficient memory access. *2nd International Conference on Networking and Computing* (2011).
[246] D. Man, K. Uda, Y. Ito, et al. 2013. Accelerating computation of Euclidean distance map using the GPU with efficient memory access. *International Journal of Parallel, Emergent and Distributed Systems* 28, 5 (2013), 383–406.
[247] S. A. Manavski and G. Valle. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics* 9, S2 (2008).
[248] S. Manoochehri, B. Goodarzi, and D. Goswami. 2017. An efficient transaction-based GPU implementation of minimum spanning forest algorithm. *International Conference on High Performance Computing & Simulation* (2017).
[249] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli. 2012. Sparse matrix-matrix multiplication on modern architectures. *19th International Conference on High Performance Computing* (2012).
[250] K. Matsumoto, N. Nakasato, T. Sakai, et al. 2011. Multi-level optimization of matrix multiplication for GPU-equipped systems. *Procedia Computer Science* 4 (2011), 342–351.
[251] K. Matsumura, H. R. Zohouri, M. Wahib, et al. 2020. AN5D: Automated stencil framework for high-degree temporal blocking on GPUs. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (2020).
[252] M. J. Mawson and A. J. Revell. 2014. Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs. *Computer Physics Communications* 185, 10 (2014), 2566–2574.
[253] M. Mendez-Lojo, M. Burtscher, and K. Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
[254] M. Mendez-Lojo, M. Burtscher, and K. Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices* 47, 8 (2012), 107.
[255] D. Merrill, M. Garland, and A. Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
[256] D. Merrill and A. Grimshaw. 2011. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters* 21, 02 (2011), 245–272.
[257] P. Micikevicius. 2009. 3D finite difference computation on GPUs using CUDA. *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*.
[258] Y. Miki, D. Takahashi, and M. Mori. 2012. A fast implementation and performance analysis of collisionless N-body code based on GPGPU. *Procedia Computer Science* 9 (2012), 96–105.
[259] Y. Misaki, F. Ino, and K. Hagihara. 2017. Cache-aware, in-place rotation method for texture-based volume rendering. *IEICE Transactions on Information and Systems* E100.D, 3 (2017), 452–461.
[260] S. Mittal. 2019. A survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *Journal of Systems Architecture* 97 (2019), 428–442.
[261] S. Mittal and S. Vaishay. 2019. A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture* 99 (2019), 101635.

[262] T. Mo and R. Li. 2014. A new memory mapping mechanism for GPGPUs' stencil computation. *Computing* 97, 8 (2014), 795–812.
[263] M. Moazeni, A. Bui, and M. Sarrafzadeh. 2009. A memory optimization technique for software-managed scratchpad memory in GPUs. *IEEE 7th Symposium on Application Specific Processors* (2009).
[264] A. Monakov, A. Lokhmotov, and A. Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. *High Performance Embedded Architectures and Compilers* (2010), 111–125.
[265] P. Montero, V. M. Gulías, J. Taibo, et al. 2012. Optimising lossless stages in a GPU-based MPEG encoder. *Multimedia Tools and Applications* 65, 3 (2012), 495–520.
[266] T. Muhammed, R. Mehmood, A. Albeshri, et al. 2019. SURAA: A novel method and tool for loadbalanced and coalesced SpMV computations on GPUs. *Applied Sciences* 9, 5 (2019), 947.
[267] D. Mukunoki, T. Imamura, and D. Takahashi. 2015. Fast implementation of general matrix-vector multiplication (GEMV) on kepler GPUs. *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2015).
[268] D. Mukunoki and D. Takahashi. 2013. Optimization of sparse matrix-vector multiplication for CRS format on NVIDIA kepler architecture GPUs. *International Conference on Computational Science and Its Applications* 7975 LNCS, PART 5 (2013), 211–223.
[269] A. Murarasu, J. Weidendorfer, G. Buse, et al. 2011. Compact data structure and scalable algorithms for the sparse grid technique. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*.
[270] G. S. Murthy, M. Ravishankar, M. M. Baskaran, et al. 2010. Optimal loop unrolling for GPGPU programs. *IEEE International Symposium on Parallel & Distributed Processing* (2010).
[271] J. M. Nadal-Serrano and M. Lopez-Vallejo. 2016. A performance study of CUDA UVM versus manual optimizations in a real-world setup: Application to a Monte Carlo wave-particle event-based interaction model. *IEEE Transactions on Parallel and Distributed Systems* 27, 6 (2016), 1579–1588.
[272] Y. Nagasaka, A. Nukada, and S. Matsuoka. 2016. Adaptive multi-level blocking optimization for sparse matrix vector multiplication on GPU. *Procedia Computer Science* 80 (2016), 131–142.
[273] J. Naghmouchi, D. P. Scarpazza, and M. Berekovic. 2010. Small-ruleset regular expression matching on GPGPUs. In *Proceedings of the 24th ACM International Conference on Supercomputing*.
[274] R. Nasre, M. Burtscher, and K. Pingali. 2013. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*.
[275] R. Nasre, M. Burtscher, and K. Pingali. 2013. Data-driven versus topology-driven irregular computations on GPUs. *IEEE 27th International Symposium on Parallel and Distributed Processing* (2013).
[276] R. Nasre, M. Burtscher, and K. Pingali. 2013. Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
[277] R. Nath, S. Tomov, T. "Tim" Dong, et al. 2011. Optimizing symmetric dense matrix-vector multiplication on GPUs. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
[278] R. Nath, S. Tomov, and J. Dongarra. 2010. An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications* 24, 4 (2010), 511–515.
[279] B. Neelima, G. R. M. Reddy, and P. S. Raghavendra. 2014. Predicting an optimal sparse matrix format for SpMV computation on GPU. *IEEE International Parallel & Distributed Processing Symposium Workshops* (2014).
[280] A. Nguyen, N. Satish, J. Chhugani, et al. 2010. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010).
[281] I. Nisa, J. Li, A. Sukumaran-Rajam, et al. 2019. Load-balanced sparse MTTKRP on GPUs. *IEEE International Parallel and Distributed Processing Symposium* (2019).
[282] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, et al. 2018. Sampled dense matrix multiplication for high-performance machine learning. *IEEE 25th International Conference on High Performance Computing* (2018).
[283] K. Nishida, Y. Ito, and K. Nakano. 2011. Accelerating the dynamic programming for the matrix chain product on the GPU. *Second International Conference on Networking and Computing* (2011).
[284] K. Nishida, K. Nakano, and Y. Ito. 2012. Accelerating the dynamic programming for the optimal polygon triangulation on the GPU. *Algorithms and Architectures for Parallel Processing* (2012), 1–15.
[285] A. E. Nocentino and P. J. Rhodes. 2010. Optimizing memory access on GPUs using morton order indexing. In *Proceedings of the 48th Annual Southeast Regional Conference*.
[286] R. Novak. 2015. Loop optimization for divergence reduction on GPUs with SIMT architecture. *IEEE Transactions on Parallel and Distributed Systems* 26, 6 (2015), 1633–1642.
[287] C. Nugteren and V. Codreanu. 2015. CLTune: A generic auto-tuner for OpenCL kernels. *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip* (2015).
[288] NVIDIA. 2010. NVIDIA's Next Generation CUDA Compute Architecture: Fermi Whitepaper.

[289] NVIDIA. 2012. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210 Whitepaper.
[290] NVIDIA. 2014. NVIDIA GeForce GTX 980 Whitepaper.
[291] NVIDIA. 2016. NVIDIA Tesla P100 Whitepaper.
[292] NVIDIA. 2017. NVIDIA Tesla V100 Whitepaper.
[293] NVIDIA. 2018. NVIDIA Turing GPU Architecture Whitepaper.
[294] NVIDIA. 2020. CUDA C++ Programming Guide. Retrieved July 2022 from https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[295] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture.
[296] D. Okada, F. Ino, and K. Hagihara. 2015. Accelerating the smith-waterman algorithm with interpair pruning and band optimization for the all-pairs comparison of base sequences. *BMC Bioinformatics* 16, 1 (2015).
[297] A. Ozsoy, A. Chauhan, and M. Swany. 2013. Achieving TeraCUPS on longest common subsequence problem using GPGPUs. *International Conference on Parallel and Distributed Systems* (2013).
[298] S. Pai and K. Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* (2016).
[299] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. *ACM SIGPLAN Notices* 48, 4 (2013), 407–418.
[300] V. K. Pallipuram, M. Bhuiyan, and M. C. Smith. 2011. A comparative study of GPU programming models and architectures using neural networks. *The Journal of Supercomputing* 61, 3 (2011), 673–718.
[301] A. Paukštė. 2013. Genetic algorithm on GPU performance optimization issues. *Intelligent Data Engineering and Automated Learning – IDEAL* (2013), 529–536.
[302] P. J. Pavan, M. S. Serpa, E. D. Carreño, et al. 2019. Improving performance and energy efficiency of geophysics applications on GPU architectures. *Communications in Computer and Information Science* (2019), 112–122.
[303] D. Petre, A. T. Lake, and A. Hux. 2016. OpenCL™ FFT optimizations for intel® processor graphics. In *Proceedings of the 4th International Workshop on OpenCL.*
[304] F. Petrovič, D. Střelák, J. Hozzová, et al. 2020. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit. *Future Generation Computer Systems* 108 (2020), 161–177.
[305] T. Y. Phuong, D. Lee, and J. Lee. 2017. Impacts of optimization strategies on performance, power/energy consumption of a GPU based parallel reduction. *Journal of Central South University* 24, 11 (2017), 2624–2637.
[306] M. Plauth, F. Feinbube, F. Schlegel, et al. 2015. Using dynamic parallelism for fine-grained, irregular workloads: A case study of the N-queens problem. *Third International Symposium on Computing and Networking.* (2015).
[307] S. F. Portegies Zwart, R. G. Belleman, and P. M. Geldof. 2007. High-performance direct gravitational N-body simulations on graphics processing units. *New Astronomy* 12, 8 (2007), 641–650.
[308] W. Qiu, Z. Gong, Y. Guo, et al. 2016. GPU-based high performance password recovery technique for hash functions. *Journal of Information Science and Engineering* 32, 1 (2016), 97–112.
[309] J. Ragan-Kelley, C. Barnes, A. Adams, et al. 2013. Halide. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.*
[310] P. S. Rawat, C. Hong, M. Ravishankar, et al. 2016. Resource conscious reuse-driven tiling for GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation.*
[311] C. Reddy, M. Kruse, and A. Cohen. 2016. Reduction drawing. In *Proceedings of the International Conference on Parallel Architectures and Compilation.*
[312] L. Reis, R. Nobre, and J. M. P. Cardoso. 2018. Impact of vectorization over 16-bit data-types on GPUs. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms.*
[313] N. Reissmann, T. L. Falch, B. A. Bjornseth, et al. 2016. Efficient control flow restructuring for GPUs. *International Conference on High Performance Computing & Simulation* (2016).
[314] T. Remmelg, T. Lutz, M. Steuwer, et al. 2016. Performance portable GPU code generation for matrix multiplication. *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit.*
[315] C. Rivera, J. Chen, N. Xiong, et al. 2021. TSM2X: High-performance tall-and-skinny matrix–matrix multiplication on GPUs. *Journal of Parallel and Distributed Computing* 151 (2021), 70–85.
[316] K. Rojek, R. Wyrzykowski, and L. Kuczynski. 2016. Systematic adaptation of stencil-based 3D MPDATA to GPU architectures. *Concurrency and Computation: Practice and Experience* 29, 9 (2016), e3970.
[317] J. W. Romein. 2016. A comparison of accelerator architectures for radio-astronomical signal-processing algorithms. In *Proceedings of the 2016 45th International Conference on Parallel Processing.* IEEE, 484–489.
[318] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, et al. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), 73–82.

[319] S. Ryoo, C. I. Rodrigues, S. S. Stone, et al. 2008. Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing* 68, 10 (2008), 1389–1401.
[320] S. Ryoo, C. I. Rodrigues, S. S. Stone, et al. 2008. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization.*
[321] M. Samadi, J. Lee, D. A. Jamshidi, et al. 2013. SAGE. In *Proceedings of the International Symposium on Microarchitecture.*
[322] E. F. d. O. Sandes and A. C. M. A. d. Melo. 2013. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Transactions on Parallel and Distributed Systems* 24, 5 (2013), 1009–1021.
[323] S. Sarkar, S. Mitra, and A. Srinivasan. 2012. Reuse and refactoring of GPU kernels to design complex applications. *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications* (2012).
[324] J. Sartori and R. Kumar. 2013. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia* 15, 2 (2013), 279–290.
[325] N. Satish, M. Harris, and M. Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. *IEEE International Symposium on Parallel & Distributed Processing* (2009).
[326] F. E. Sayadi, M. Chouchene, H. Bahri, et al. 2019. CUDA memory optimisation strategies for motion estimation. *IET Computers & Digital Techniques* 13, 1 (2019), 20–27.
[327] S. Sengupta, M. Harris, Y. Zhang, et al. 2007. Scan primitives for GPU computing. *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2007), 97–106.
[328] M. Sha, Y. Li, and K. Tan. 2019. GPU-based graph traversal on compressed graphs. In *Proceedings of the International Conference on Management of Data.*
[329] S. Shi, Q. Wang, and X. Chu. 2020. Efficient sparse-dense matrix-matrix multiplication on GPUs using the customized sparse storage format. *IEEE 26th International Conference on Parallel and Distributed Systems* (2020).
[330] Y. Shin, B. Sohn, and H. Kye. 2021. Acceleration techniques for cubic interpolation MIP volume rendering. *Multimedia Tools and Applications* 80, 14 (2021), 20971–20989.
[331] J. Siegel, J. Ributzka, and X. Li. 2009. CUDA memory optimizations for large data-structures in the gravit simulator. *International Conference on Parallel Processing Workshops* (2009).
[332] E. A. Sitaridi and K. A. Ross. 2015. GPU-accelerated string matching for database applications. *The VLDB Journal* 25, 5 (2015), 719–740.
[333] B. Sofranac, A. Gleixner, and S. Pokutta. 2020. Accelerating domain propagation: An efficient GPU-parallel algorithm over sparse matrices. *IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms* (2020).
[334] J. Soman, K. Kishore, and P. J. Narayanan. 2010. A fast GPU algorithm for graph connectivity. *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum.*
[335] K. Sopyła, P. Drozda, and P. Górecki. 2012. SVM with CUDA accelerated kernels for big sparse problems. *Artificial Intelligence and Soft Computing* (2012), 439–447.
[336] S. Srungarapu, D. P. Reddy, K. Kothapalli, et al. 2011. Fast two dimensional convex hull on the GPU. *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications Workshops.*
[337] J. A. Stratton, N. Anssari, C. Rodrigues, et al. 2012. Optimization and architecture effects on GPU computing workload performance. *Innovative Parallel Computing* (2012).
[338] D. Střelák, C. Ó. S. Sorzano, J. M. Carazo, et al. 2019. A GPU acceleration of 3-D Fourier reconstruction in cryo-EM. *The International Journal of High Performance Computing Applications* 33, 5 (2019), 948–959.
[339] B. Su and K. Keutzer. 2012. clSpMV. In *Proceedings of the 26th ACM International Conference on Supercomputing.*
[340] H. Su, N. Wu, M. Wen, et al. 2013. On the GPU performance of 3D stencil computations implemented in OpenCL. *International Supercomputing Conference* (2013), 125–135.
[341] Y. Sugimoto, F. Ino, and K. Hagihara. 2014. Improving cache locality for GPU-based volume rendering. *Parallel Computing* 40, 5–6 (2014), 59–69.
[342] X. Sun, Y. Zhang, T. Wang, et al. 2011. Optimizing SpMV for diagonal sparse matrices on GPU. *International Conference on Parallel Processing* (2011).
[343] I. Sung, J. Gómez-Luna, J. M. González-Linares, et al. 2014. In-place transposition of rectangular matrices on accelerators. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*
[344] I. Sung, J. A. Stratton, and W. W. Hwu. 2010. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques.*
[345] K. Świrydowicz, N. Chalmers, A. Karakus, et al. 2019. Acceleration of tensor-product operations for high-order finite element methods. *The International Journal of High Performance Computing Applications* 33, 4 (2019), 735–757.
[346] S. Tabik, G. Ortega, and E. M. Garzón. 2014. Performance evaluation of kernel fusion BLAS routines on the GPU: Iterative solvers as case study. *The Journal of Supercomputing* 70, 2 (2014), 577–587.

[347] T. Takahashi, C. Cecka, W. Fong, et al. 2011. Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. *International Journal for Numerical Methods in Engineering* 89, 1 (2011), 105–133.
[348] G. Tan, L. Li, S. Triechle, et al. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
[349] W. Tan, L. Cao, and L. Fong. 2016. Faster and cheaper. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*.
[350] H. Tang, K. Komatsu, M. Sato, et al. 2020. An efficient skinny matrix-matrix multiplication method by folding input matrices into tensor core operations. *8th International Symposium on Computing and Networking Workshops* (2020).
[351] W. T. Tang, W. J. Tan, R. Krishnamoorthy, et al. 2013. Optimizing and auto-tuning iterative stencil loops for GPUs with the in-plane method. *IEEE 27th International Symposium on Parallel and Distributed Processing* (2013).
[352] W. T. Tang, W. J. Tan, R. Ray, et al. 2013. Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
[353] Y. Tang, O. Selvitopi, D. T. Popovici, et al. 2020. A high-throughput solver for marginalized graph kernels on GPU. *IEEE International Parallel and Distributed Processing Symposium* (2020).
[354] C. Tezcan. 2021. Optimization of advanced encryption standard on graphics processing units. *IEEE Access* 9 (2021), 67315–67326.
[355] S. Tomov, J. Dongarra, and M. Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing* 36, 5–6 (2010), 232–240.
[356] S. Tomov, R. Nath, H. Ltaief, et al. 2010. Dense linear algebra solvers for multicore with GPU accelerators. *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2010).
[357] H. Tran and E. Cambria. 2018. A survey of graph processing on graphics processing units. *The Journal of Supercomputing* 74, 5 (2018), 2086–2115.
[358] N. Tran, M. Lee, and D. H. Choi. 2015. Memory-efficient parallelization of 3D lattice boltzmann flow solver on a GPU. *IEEE 22nd International Conference on High Performance Computing* (2015).
[359] N. Tran, M. Lee, S. Hong, et al. 2013. Performance optimization of aho-corasick algorithm on a GPU. *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (2013).
[360] S. Tzengy, A. Patney, and J. D. Owens. 2010. Task management for irregular-parallelworkloads on the GPU. *High-Performance Graphics - ACM SIGGRAPH / Eurographics Symposium Proc., HPG* (2010), 29–37.
[361] A. Uchida, Y. Ito, and K. Nakano. 2012. An efficient GPU implementation of ant colony optimization for the traveling salesman problem. *Third International Conference on Networking and Computing* (2012).
[362] D. U. Ufuk, A. Temizel, and A. M. Ozbayoglu. 2018. Optimized GPU implementation of JPEG 2000 for satellite image decompression. *IEEE International Conference on Computational Science and Engineering* (2018).
[363] J. Vanek, J. Trmal, J. V. Psutka, et al. 2012. Optimized acoustic likelihoods computation for NVIDIA and ATI/AMD graphics processors. *IEEE Transactions on Audio, Speech, and Language Processing* 20, 6 (2012), 1818–1828.
[364] G. Vasiliadis, S. Antonatos, M. Polychronakis, et al. 2008. Gnort: High performance network intrusion detection using graphics processors. *Recent Advances in Intrusion Detection* (2008), 116–134.
[365] F. Vázquez, J. J. Fernández, and E. M. Garzón. 2010. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 23, 8 (2010), 815–826.
[366] F. Vázquez, G. Ortega, J. J. Fernández, et al. 2010. Improving the performance of the sparse matrix vector product with GPUs. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*.
[367] B. Veenboer, M. Petschow, and J. W. Romein. 2017. Image-domain gridding on graphics processors. *IEEE International Parallel and Distributed Processing Symposium* (2017).
[368] B. Veenboer and J. W. Romein. 2019. Radio-astronomical imaging: FPGAs vs GPUs. In *Proceedings of the European Conference on Parallel Processing*. Springer, 509–521.
[369] L. Vespa, A. Bauman, and J. Wells. 2015. Algorithm flattening: Complete branch elimination for GPU requires a paradigm shift from CPU thinking. *IEEE High Performance Extreme Computing Conference* (2015).
[370] C. Vetter and R. Westermann. 2011. Optimized GPU histograms for multi-modal registration. *IEEE International Symposium on Biomedical Imaging: From Nano to Macro* (2011).
[371] B. Videau, V. Marangozova-Martin, L. Genovese, et al. 2013. Optimizing 3D convolutions for wavelet transforms on CPUs with SSE Units and GPUs. *Euro-Par Parallel Processing* (2013), 826–837.
[372] A. Vizitiu, L. Itu, L. Lazar, et al. 2014. Double precision stencil computations on Kepler GPUs. *18th International Conference on System Theory, Control and Computing* (2014).
[373] V. Volkov. 2010. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference* 10 (2010), 16.
[374] V. Volkov. 2016. *Understanding Latency Hiding on GPUs*. Ph.D. Dissertation. UC Berkeley.
[375] V. Volkov and J. W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. *SC - International Conference for High Performance Computing, Networking, Storage and Analysis, SC* (2008).

[376] A. Wakatani. 2016. Evaluation of splitting-up conjugate gradient method on GPUs. *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2016).
[377] L. Wan, K. Li, J. Liu, et al. 2015. Efficient CPU-GPU cooperative computing for solving the subset-sum problem. *Concurr Comput* 28, 2 (2015), 492–516.
[378] G. Wang, Y. Lin, and W. Yi. 2010. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. *IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing* (2010).
[379] G. Wang, T. Tang, X. Fang, et al. 2009. Program optimization of array-intensive SPEC2k benchmarks on multithreaded GPU using CUDA and Brook+. *15th International Conference on Parallel and Distributed Systems* (2009).
[380] J. Wang, X. Ma, Y. Zhu, et al. 2013. Auto-tuning of thread assignment for matrix-vector multiplication on GPUs. *IEICE Transactions on Information and Systems* E96.D, 11 (2013), 2319–2326.
[381] J. Wang, X. Xie, and J. Cong. 2017. Communication optimization on GPU: A case study of sequence alignment algorithms. *IEEE International Parallel and Distributed Processing Symposium* (2017).
[382] L. Wang, Z. Chen, Y. Liu, et al. 2019. A unified optimization approach for CNN model inference on integrated GPUs. *Proceedings of the 48th International Conference on Parallel Processing*.
[383] S. Wang, Z. Li, and Y. Che. 2021. Memory access optimization of high-order CFD stencil computations on GPU. *Parallel and Distributed Computing, Applications and Technologies* (2021), 43–56.
[384] Y. Wang, A. Davidson, Y. Pan, et al. 2015. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
[385] Y. Wang, J. Qiao, S. Lin, et al. 2016. Performance optimization for CPU-GPU heterogeneous parallel system. *12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery* (2016).
[386] Z. Wang, X. Xu, W. Zhao, et al. 2010. Optimizing sparse matrix-vector multiplication on CUDA. *2nd International Conference on Education Technology and Computer* (2010).
[387] K. Wei, X. Sun, H. Chu, et al. 2017. Reconstructing permutation table to improve the tabu search for the PFSP on GPU. *Journal of Supercomputing* 73, 11 (2017), 4711–4738.
[388] Z. Wei and J. JaJa. 2010. Optimization of linked list prefix computations on multithreaded GPUs using CUDA. *IEEE International Symposium on Parallel & Distributed Processing* (2010).
[389] B. van Werkhoven. 2019. Kernel tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 90 (2019), 347–358.
[390] B. van Werkhoven, J. Maassen, H. E. Bal, et al. 2014. Optimizing convolution operations on GPUs using adaptive tiling. *Future Generation Computer Systems* 30 (2014), 14–26.
[391] B. van Werkhoven, J. Maassen, and F. J. Seinstra. 2011. Optimizing convolution operations in CUDA with adaptive tiling. In *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors*.
[392] B. van Werkhoven, J. Maassen, F. J. Seinstra, et al. 2014. Performance models for CPU-GPU data transfers. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* 11–20.
[393] A. Wijs, T. Neele, and D. Bošnački. 2016. GPUexplore 2.0: Unleashing GPU explicit-state model checking. *FM: Formal Methods* (2016), 694–701.
[394] S. Williams, A. Waterman, and D. Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52, 4 (2009), 65–76.
[395] C. Wu, J. Ke, H. Lin, et al. 2011. Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism. *IEEE 17th International Conference on Parallel and Distributed Systems* (2011).
[396] C. Wu, K. Wei, and T. Lin. 2012. Optimizing dynamic programming on graphics processing units via data reuse and data prefetch with inter-block barrier synchronization. *IEEE 18th International Conference on Parallel and Distributed Systems* (2012).
[397] H. Wu, G. Diamos, J. Wang, et al. 2012. Optimizing data warehousing applications for GPUs using kernel fusion/fission. *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (2012).
[398] J. Wu and J. JaJa. 2012. Optimized strategies for mapping three-dimensional FFTs onto CUDA GPUs. *Innovative Parallel Computing* (2012).
[399] Y. Xia, P. Jiang, and G. Agrawal. 2019. Enabling prefix sum parallelism pattern for recurrences with principled function reconstruction. In *Proceedings of the 28th International Conference on Compiler Construction*.
[400] S. Xiao and W. Feng. 2010. Inter-block GPU communication via fast barrier synchronization. *IEEE International Symposium on Parallel & Distributed Processing* (2010).
[401] S. Xiao, H. Lin, and W. Feng. 2011. Accelerating protein sequence search in a heterogeneous computing system. *IEEE International Parallel & Distributed Processing Symposium* (2011).
[402] J. Xu, H. Fu, L. Gan, et al. 2016. Generalized GPU acceleration for applications employing finite-volume methods. *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2016).

[403] S. Xu, W. Xue, and H. X. Lin. 2011. Performance modeling and optimization of sparse matrix-vector multiplication on NVIDIA CUDA platform. *The Journal of Supercomputing* 63, 3 (2011), 710–721.
[404] W. Xu, Z. Liu, J. Wu, et al. 2012. Auto-tuning GEMV on many-core GPU. *IEEE 18th International Conference on Parallel and Distributed Systems* (2012).
[405] K. Yamashita, Y. Ito, and K. Nakano. 2018. Bulk execution of the dynamic programming for the optimal polygon triangulation problem on the GPU. *Concurrency and Computation: Practice and Experience* 31, 19 (2018), e4947.
[406] D. Yan, H. Cao, X. Dong, et al. 2011. Optimizing algorithm of sparse linear systems on GPU. *6th Annual Chinagrid Conference* (2011).
[407] D. Yan, W. Wang, and X. Chu. 2020. Demystifying tensor cores to optimize half-precision matrix multiply. *IEEE International Parallel and Distributed Processing Symposium* (2020).
[408] D. Yan, W. Wang, and X. Chu. 2020. Optimizing batched winograd convolution on GPUs. *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*
[409] S. Yan, C. Li, Y. Zhang, et al. 2014. yaSpMV: Yet another SpMV framework on GPUs. *19th ACM SIGPLAN Symp. on Principles and Pract. of Parallel Programm.* (2014).
[410] S. Yan, G. Long, and Y. Zhang. 2013. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization. *18th ACM SIGPLAN Symp. on Principles and Pract. of Parallel Program.* (2013).
[411] C. Yang, A. Buluç, and J. D. Owens. 2018. Design principles for sparse matrix multiplication on the GPU. *Euro-Par : Parallel Processing* (2018), 672–687.
[412] S. Yang, T. Lin, and S. Chien. 2009. Real-time motion estimation for 1080p videos on graphics processing units with shared memory optimization. *IEEE Workshop on Signal Processing Systems* (2009).
[413] X. Yang, S. Parthasarathy, and P. Sadayappan. 2011. Fast sparse matrix-vector multiplication on GPUs. *Proc. VLDB Endowment* 4, 4 (2011), 231–242.
[414] Y. Yang, P. Xiang, J. Kong, et al. 2010. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*
[415] Y. Yang, P. Xiang, J. Kong, et al. 2012. A unified optimizing compiler framework for different GPGPU architectures. *ACM Transactions on Architecture and Code Optimization* 9, 2 (2012), 1–33.
[416] Y. Yang and H. Zhou. 2012. The implementation of a high performance GPGPU compiler. *International Journal of Parallel Programming* 41, 6 (2012), 768–781.
[417] Y. Yang and H. Zhou. 2014. CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications. *19th ACM SIGPLAN Symp. on Principles and Prac. of Parallel Program.* (2014).
[418] W. Yi, Y. Tang, G. Wang, et al. 2010. A case study of SWIM: Optimization of memory intensive application on GPGPU. *3rd International Symposium on Parallel Architectures, Algorithms and Programming* (2010).
[419] K. K. Yong and S. S. O. Talib. 2016. Histogram optimization with CUDA. *IEEE Industrial Electronics and Applications Conference* (2016).
[420] H. Yoshizawa and D. Takahashi. 2012. Automatic tuning of sparse matrix-vector multiplication for CRS format on GPUs. *IEEE 15th International Conference on Computational Science and Engineering* (2012).
[421] Y. You, H. Fu, S. L. Song, et al. 2014. Evaluating multi-core and many-core architectures through accelerating the three-dimensional Lax–Wendroff correction stencil. *The International Journal of High Performance Computing Applications* 28, 3 (2014), 301–318.
[422] L. Yu, A. Goldsmith, and S. D. Cairano. 2017. Efficient convex optimization on GPUs for embedded model predictive control. *Proceedings of the General Purpose GPUs.*
[423] W. Yu, T. Chen, F. Franchetti, et al. 2010. High performance stereo vision designed for massively data parallel platforms. *IEEE Transactions on Circuits and Systems for Video Technology* 20, 11 (2010), 1509–1519.
[424] Y. Yuttakonkit, S. Takamaeda-Yamazaki, and Y. Nakashima. 2016. Performance optimization of light-field applications on GPU. *IEICE Transactions on Information and Systems* E99.D, 12 (2016), 3072–3081.
[425] K. Zhai, T. Banerjee, A. Wijayasiri, et al. 2020. Batched small tensor-matrix multiplications on GPUs. *IEEE 27th International Conference on High Performance Computing, Data, and Analytics* (2020).
[426] E. Z. Zhang, Y. Jiang, Z. Guo, et al. 2010. Streamlining GPU applications on the fly. *Proceedings of the 24th ACM International Conference on Supercomputing.*
[427] E. Z. Zhang, Y. Jiang, Z. Guo, et al. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems.*
[428] F. Zhang, P. Di, H. Zhou, et al. 2016. RegTT: Accelerating tree traversals on GPUs by exploiting regularities. *45th International Conference on Parallel Processing* (2016).
[429] F. Zhang, J. Su, W. Liu, et al. 2021. YuenyeungSpTRSV: A thread-level and warp-level fusion synchronization-free sparse triangular solve. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2321–2337.
[430] Q. Zhang, H. An, G. Liu, et al. 2010. The optimization of parallel Smith-Waterman sequence alignment using on-chip memory of GPGPU. *IEEE 5th International Conference on Bio-Inspired Computing: Theories and Applications* (2010).

[431] T. Zhang, W. Shu, and M. Wu. 2011. Optimization of N-queens solvers on graphics processors. *International Workshop on Advanced Parallel Processing Technologies* (2011), 142–156.
[432] Y. Zhang, S. Li, S. Yan, et al. 2016. A cross-platform SpMV framework on many-core architectures. *ACM Transactions on Architecture and Code Optimization* 13, 4 (2016), 1–25.
[433] Z. Zhao, L. Song, R. Xie, et al. 2016. GPU accelerated high-quality video/image super-resolution. *IEEE International Symposium on Broadband Multimedia Systems and Broadcasting* (2016).
[434] J. Zhong and B. He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552.
[435] Y. Zhou, F. He, and Y. Qiu. 2016. Optimization of parallel iterated local search algorithms on graphics processing unit. *The Journal of Supercomputing* 72, 6 (2016), 2394–2416.
[436] H. Zhu, L. He, M. Leeke, et al. 2019. WolfGraph: The edge-centric graph processing on GPU. *Future Generation Computer Systems* (2019).
[437] M. Zhu, Y. Zhuo, C. Wang, et al. 2017. Performance evaluation and optimization of HBM-Enabled GPU for data-intensive applications. *Design, Automation & Test in Europe Conference & Exhibition* (2017).
[438] V. Zois, A. Panangadan, and V. Prasanna. 2016. Accelerating support count for association rule mining on GPUs. *IEEE International Parallel and Distributed Processing Symposium Workshops* (2016).
[439] D. Zou, Y. Dou, and F. Xia. 2011. Optimization schemes and performance evaluation of Smith-Waterman algorithm on CPU, GPU and FPGA. *Concurrency and Computation: Practice and Experience* 24, 14 (2011), 1625–1644.

Received 30 July 2021; revised 22 July 2022; accepted 17 October 2022