# Optimization and Architecture Effects on GPU Computing Workload Performance

John A. Stratton* stratton@illinois.edu
Nasser Anssari* anssari1@illinois.edu
Christopher Rodrigues* cirodrig@illinois.edu
I-Jui Sung* sung10@illinois.edu
Nady Obeid*† nady.obeid@gmail.com
Liwen Chang* lchang20@illinois.edu
Geng Daniel Liu* gengliu2@illinois.edu
Wen-mei Hwu* w-hwu@illinois.edu

*University of Illinois
Electrical and Computer Engineering
Urbana, Illinois, USA

†KLA Tencor
Milpitas, California, USA

## ABSTRACT

It is unquestionable that successive hardware generations have significantly improved GPU computing workload performance over the last several years. Moore's law and DRAM scaling have respectively increased single-chip peak instruction throughput by 3X and off-chip bandwidth by 2.2X from NVIDIA's GeForce 8800 GTX in November 2006 to its GeForce GTX 580 in November 2010. However, raw capability numbers typically underestimate the improvements in real application performance over the same time period, due to significant architectural feature improvements.

To demonstrate the effects of architecture features and optimizations over time, we conducted experiments on a set of benchmarks from diverse application domains for multiple GPU architecture generations to understand how much performance has truly been improving for those workloads. First, we demonstrate that certain architectural features make a huge difference in the performance of unoptimized code, such as the inclusion of a general cache which can improve performance by 2-4× in some situations. Second, we describe what optimization patterns have been most essential and widely applicable for improving performance for GPU computing workloads across all architecture generations. Some important optimization patterns included data layout transformation, converting scatter accesses to gather accesses, GPU workload regularization, and granularity coarsening, each of which improved performance on some benchmark by over 20%, sometimes by a factor of more than 5×. While hardware improvements to baseline unoptimized code can reduce the speedup magnitude, these patterns remain important for even the most recent GPUs. Finally, we identify which added architectural features created significant new optimization opportunities, such as increased register file capacity or reduced bandwidth penalties for misaligned accesses, which increase performance by 2× or more in the optimized versions of relevant benchmarks.

## Keywords

GPU, CUDA, Optimization

## 1. INTRODUCTION

While no community or field springs out of nothing, the modern field of GPU computing had a major inflection point approximately five years ago with the first support for C-based programming languages for general computation on GPUs. Very quickly, the community discovered and published what worked well on GPU platforms and what didn't at first. As the years progressed, GPU architects and application researchers continually pushed at the boundaries of what GPUs could do effectively, significantly improving performance for many workloads. At this juncture, with five years of experience and a new academic conference explicitly dedicated to novel parallel computing platforms and applications, we would like to examine some GPU computing workloads and see how far we have come, and what is most important to learn from the current state of the art as we continue to move forward.

We would like to focus on two major aspects of the GPU computing field over the last five years. The first is the optimization and programming patterns that have shaped optimized applications for GPU architectures. Several design principles of GPU architectures have been and will likely continue to be very consistent, such as SIMT and high degrees of multithreading. We have surveyed many GPU computing applications and kernels and distilled what we believe to be several key optimization techniques and design considerations for high-performance GPU-computing workloads. These techniques deserve to be covered in some detail and in a way that can be understood across domains, because they reflect the common patterns a new GPU programmer in any domain should learn. In addition, we believe that these optimization patterns will continue to gain broader relevance over time, because the optimization patterns we present are fundamentally about implementing scalable, efficient parallel programs on an architecture with many cores, vector execution, and limited memory bandwidth. We present our focused discussion on such optimization patterns in section 3.

Second, we present new experiments exploring the design space of GPU architecture itself, which has been both remarkably consistent and increasingly friendly to application programmers over the last several years. The tipping point of GPU computing five years ago coincided with the GPU

978-1-4673-2633-9/12/$31.00 

Table 1: GPU Architecture Summaries

| | 9800 GX2 (One GPU) | S1070 (One GPU) | GTX 480 |
|---|---|---|---|
| Released | Mar. 2008 | Nov. 2008 | Mar. 2010 |
| Compute capability | 1.1 | 1.3 | 2.0 |
| SP Ops/sec | 576 GFLOPS | 1037 GFLOPS | 1345 GFLOPS |
| DRAM bandwidth | 64 GB/s | 102 GB/s | 177 GB/s |
| Features | global memory atomics, shared & constant memory | shared memory atomics, reduced coalescing penalties | general cache, increased shared memory capacity |

industry shift to *unified shader cores*, and for good reason. Unified compute cores running a common instruction set enabled a single SPMD kernel to use all available processing on the GPU with no special effort. However, the first compute-capable GPUs still lacked many features, restricting the applications that could effectively use them. We select a benchmark suite of GPU workloads, track its performance through the years of GPU architecture revisions, and analyze when and how performance improved most for each kind of application. Our experiments, presented in section 4 should give us some insight into what architecture features in the accelerator design space truly mattered for workload performance. In section 5, we summarize our insights from both the optimization patterns and architecture studies. But first, we shall describe some necessary background and methodology information.

## 2. ARCHITECTURE EVOLUTION

GPU computing workloads are extremely diverse. Literally hundreds of GPU application optimization papers have been published over the past few years alone. The GPU Computing Gems books alone document dozens, from very diverse fields [4, 5]. The SHOC OpenCL benchmark suite (with some equivalent CUDA benchmarks) has multiple "levels", benchmarking increasingly complicated codes [3]. The SHOC level 0 benchmarks are essentially microbenchmarks for stress-testing things like memory bandwidth. Level 1 benchmark are simple, common primatives such as reduction, scan, SGEMM, and others. Level 2 benchmarks (of which there is only one at time of writing) are considered full applications. The Rodinia [2] and Parboil [8] benchmark suites overlap some with each other and with the SHOC benchmarks, mixing "primitives" with "applications". Both Parboil and Rodinia have CUDA and OpenCL implementations, except for two Rodinia benchmarks lacking OpenCL support at time of writing. Because part of our goals was to reimplement and reoptimize our benchmarks for multiple GPU devices, we chose to work with the Parboil benchmark suite, which has both reasonably-sized benchmarks for reimplementation and scripting support for multiple implementations and platforms.

All our experiments were conducted on one of three GPUs, which are summarized in Table 1. The first system houses an NVIDIA GeForce 9800 GX2, of which we only use one of the included GPU chips. We chose this device as opposed to slightly older G80-generation device because of the added support for atomic operations to global memory, without which some of our benchmarks would be practically impossible to implement. Atomic operations to shared memory are not supported in this device, but can often by emulated by barriers or clever uses of shared memory consistency and warp scheduling assumptions valid for that device. Such emulated atomic operations come at a higher software development, performance, and portability cost. Additionally, the 9800 GX2 architecture carries a harsh penalty to any imperfectly coalesced memory accesses, generating many DRAM line accesses even for contiguous but simply misaligned accesses.

Experiments on a second system executed on a single GPU of an NVIDIA Tesla S1070. That GPU, and others of compute capability 1.2 and higher, support atomic operations to shared memory, allowing for more robust and efficient communication among threads in a block for certain programming patterns. In addition, the S1070 removes the harshest penalties for uncoalesced accesses, such that a misaligned access will only transfer the two DRAM lines straddled by the contiguous addresses. In general, the S1070's coalescing unit will generate the fewest number of DRAM transactions necessary to satisfy the requests from a single warp-instruction.

Finally, the third system consists of an NVIDIA GTX 480. The "Fermi" GPU generation, designated by compute capabilities 2.0 and higher and including the GTX 480, adds a general cache hierarchy to the global memory system, and increased shared memory capacity for those kernels needing it. In addition, an addition to its instruction set allows the CUDA compiler to automatically target certain access patterns to the constant memory cache for high broadcast performance.

Clearly, at time of publication all of these devices will be two or more years old, which for scientific computing implies that the performance results may not be directly relevant for future hardware. However, the insights gained from analyzing the feature sets should continue to be informative. While NVIDIA is unlikely to remove standardized hardware features in future generations, determining which features proved most impactful can help shape the accelerator market as a whole towards the most useful feature set for accelerated workloads.

## 3. OPTIMIZATION PATTERNS FOR PARALLEL CHIP ARCHITECTURES

A segment of the parallel programming community has long been interested in characterizing the programming patterns that are effective for parallel systems [7, 6]. However, in our conversations with some of authors in that field, they have confided that they sometimes struggle with the fact that once a parallel program is implemented, the optimization process involves software development practices completely outside the domain of their structural patterns. We would therefore like to begin the academic discussion of a set of patterns systematizing the *optimization* of parallel programs. The optimization patterns were drawn in particular from our informal survey of the GPU Computing Gems contributions [4, 5], and from a focused and detailed analysis of the Parboil benchmarks.

Table 2 summarizes our proposed collection of optimiza-

| Technique | Contention | Bandwidth | Locality | Efficiency | Load Imbalance | CPU Leveraging |
|---|---|---|---|---|---|---|
| Tiling | | X | X | | | |
| Privatization | X | | X | | | |
| Scatter to Gather Conversion | X | | | | | |
| Binning | | X | X | X | | X |
| Regularization | | | | X | X | X |
| Compaction | | X | | | | |
| Data Layout Transformation | X | | X | | | |
| Granularity Coarsening | X | X | X | X | | |

Table 2: Issues Addressed by Optimization Pattern

tion patterns and the goals or benefits accomplished by each. One interesting note is that for certain goals, more than one potential optimization technique may apply. The choice of which technique best fits the application currently requires significant human analysis of the code, a costly effort.

Because accelerators are highly parallel devices, many of the techniques specifically address general performance issues that arise from programming a highly parallel shared-memory architecture, such as contention and load imbalance. Some techniques are not specific to highly parallel architectures, but avoid especially severe performance cliffs given the design of today's accelerator architectures, such as the especially software-driven approaches to effective bandwidth utilization and locality management. Still other techniques are specifically targeted towards leveraging the benefits of a hybrid system, using the versatility of the CPU to not only process necessarily sequential code regions but to also precondition GPU kernel inputs such that kernels can be further optimized than would be possible for general input.

Finally, parallel architectures are fundamentally a collection of sequential processing units. When a parallel architecture is well used, the performance limitation of a program on that architecture is the efficiency of the sequential programs running on each execution unit. Therefore, sequential program performance optimization is still an area of interest for the SPMD code executed on the accelerator. We will not discuss those techniques here, as they are well studied and not unique to parallel programming systems, but acknowledge that immature compiler technology sometimes will necessitate direct programmer implementations of "trivial" code optimizations.

We firmly believe that every one of the patterns we describe here has been explained in previous work, but note that previous descriptions of these patterns as they apply to GPU workloads are typically embedded within implementations of specific workloads. While we do not take credit for being the first to discover any of these individual transformations, we believe that there is useful insight to be gained by consolidating summaries of all those we found applied to the Parboil benchmarks in a way that highlights their generality to a variety of GPU computing workloads. By gathering the optimization patterns together, anchored by the real benchmarks using them, we can study how they interact with one another, their variations among different applications, and their individual and cumulative results on real hardware systems.

We demonstrate the impact of each individual pattern by presenting, for benchmarks where that pattern was particularly relevant, performance improvements from the highest-performing code we could write without that optimization to the highest-performing code we have. Except where noted otherwise, results in this section are collected from the NVIDIA Tesla S1070 system described in Section 2.

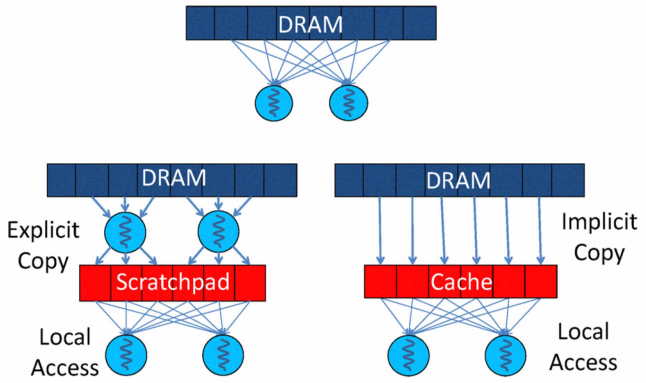


Figure 1: Tiling diagram for implicit storage (e.g. cache) and explicit storage (e.g. scratchpad)

## 3.1 Tiling

Tiling is perhaps the most widely used and understood technique for best utilizing a tiered memory hierarchy. While the technique is fundamentally the same in sequential code optimization, the actual implementation can vary with the design of an architecture's memory hierarchy, as shown in Figure 1. Tiling in the context of a CPU architecture with a large-capacity, implicitly managed cache hierarchy typically means writing regions of code that operate intensively on smaller sections of memory. The regions could then be repeated many times for different sections, or tiles, of data. The application need not explicitly define the region of memory being operated on, as the hardware should automatically respond to the heavy usage of certain regions and retain those regions in the cache.

One of the most obvious differences of current GPU architecture is explicitly managed on-chip memory, such as on the right side of Figure 1. To use the small-capacity, high-bandwidth scratchpad, software must explicitly move data into it before use. The threads themselves are mediators between DRAM and scratchpad, under the direction of source code written by application programmers.

Recent GPUs have also added small implicit caches to their general memory system, providing a hybrid of implicitly and explicitly managed locality mechanisms. What

Table 3: Tiling results

| Benchmark | Pattern performance impact |
|---|---|
| Stencil | 3.15× |
| TPACF | 1.12× |
| SGEMM | 6.18× |

makes even cached GPUs significantly different from typical CPUs is that the ratio of cache capacity to the number of potentially active threads is incredibly small for GPUs. Indeed, the overall predicted trend for highly multithreaded processors is towards more limited resources per thread []. For instance, if all thread contexts were active in an NVIDIA Fermi GPU and cache space were partitioned among active threads, each thread would have a mere 32 bytes of L1 or L2 cache space.

Clearly, neither caches nor scratchpads in current GPUs were designed for CPU-style temporal locality and thread-private tiles, but for overlapping accesses among threads. A block of threads may collectively have 16kB of private cache or scratchpad space even when all thread contexts are active, which is often sufficient to hold worthwhile-sized tiles of data. Therefore, the software technique of tiling is still applicable for GPUs, but very often must take the form of cooperative tiling using the shared resources of several threads for sufficient impact.

The performance impacts of tiling are significant, as shown in Table 3. The 3× improvement in performance for the stencil benchmark corresponds to the fact that memory tiling reduces the number of bytes accessed from global memory per iteration from 5 words per thread to 1.25 words per thread on average. Performance does not increase by a full factor of 4× primarily because some accesses are still misaligned, not fully utilizing DRAM bandwidth. Although the results in Table 3 are for a cacheless GPU, our experiments in Section 4.3 verify, as any CPU high-performance programmer will assert, that software tiling is still critical for architectures with implicit caches.

## 3.2 Privatization

Privatization is the transformation of taking some data that was once common or shared among parallel tasks and duplicating it such that different parallel tasks have a private copy on which to operate. Parallel threads typically operate most efficiently when they can operate completely independently, avoiding coordination with other threads, but many parallel algorithms require threads to interact to obtain a final result. Privatization is applied to isolate regions of code where threads can operate independently and efficiently, before eventually combining results.

Figure 2 shows a common multi-level privatization pattern reflecting the hierarchical task decomposition common among highly parallel systems such as clusters or single-chip GPUs. Working up from the bottom of Figure 2, a global result is built from the partial results from many independent tasks (thread blocks in the case of a GPU.) Those partial results are each in turn constructed from many more "private" results. This kind of privatization has applications in many different kinds of algorithms. Collective operations such as sorting or reductions will use this pattern, as will data structures such as histograms or queues.

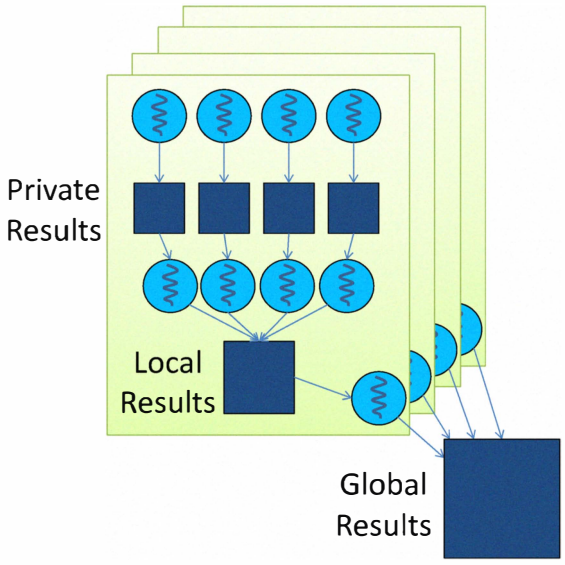


Figure 2: Example of the common hierarchical privatization pattern

Table 4: Privatization results

| Benchmark | Pattern performance impact |
|---|---|
| BFS | 3.15× |
| Histo | 2.26× (GTX 480) |

One limitation of privatization is that the data footprint of the copies and the overhead of combining the copies scale with the amount of parallelism being exploited. This is why privatization is an extremely powerful technique for today's CMPs, with a relatively small number of threads, but somewhat limited for the levels of thread parallelism in highly multithreaded architectures. Often the "private" results are still shared by several GPU threads due to resource limitations, but are intended to be constructed with as little inter-thread cooperation as possible. As shown in Table 4, the BFS Parboil benchmark privatizes the output work queues, resulting in a 3× performance improvement over an unprivatized implementation. Privatization allows the BFS kernels to exchange more costly global memory atomic operations for shared memory atomic operations, and also collects irregular updates in shared memory before bulk-committing results to the global queue in a more regular pattern, improving bandwidth. For the histogram benchmark, the privatization transformation was ineffective for the S1070 due to shared memory capacity limitations; we therefore report GTX 480 speedups in Table 4 for that benchmark.

## 3.3 Scatter to Gather Transformation

A few Parboil applications demonstrate a computation pattern where an input datum would either contribute to many output elements, or contribute to one or more statically unknown output elements, such as shown in Figure 3. In both either cases, a common pattern for sequential implementation is to examine each input element, determine the output elements it affects, and update each one before moving on to the next input element.

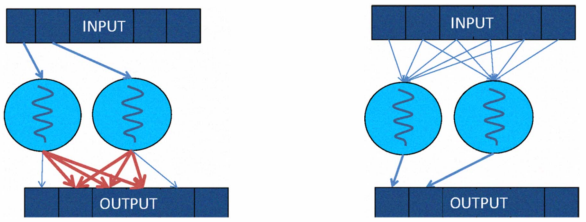


Figure 3: Depiction of a scatter-to-gather transformation

Table 5: Scatter-to-Gather results

| Benchmark | Pattern performance impact (GTX 480) |
|---|---|
| Histo | 1.22× |

This method works poorly as parallelism scales, because the output accesses are either contentious or random or both. Examining the previous techniques, we see that tiling is very effective on input data, and privatization is very effective on output data. However, a kernel implemented with a scattering approach has no input read sharing to tile, and no outputs with multiple updates from the same thread to privatize. In these situations, it is often very important to transform the code such that input elements are read-shared, but output elements are private to a parallel task. This is more palatable than the converse case because shared reads can be much more efficiently handled than conflicting writes, which typically require more costly atomic operations and coherence enforcement. A conversion to gather accesses means that privatization can be applied to output writes, reducing their cost, while techniques such as tiling can be applied to improve shared read efficiency.

Scatter-to-Gather transformation works exceptionally well when the range of inputs affecting an output can be found without direct examination of the input data contents. If this input-to-output mapping cannot be done statically, sometimes the transformation must be supplemented with a binning operation. The Parboil Histogram benchmark gains about 20% performance on a Fermi architecture by using a gather-based approach instead of a scattering approach. Results are presented on the GTX 480 because the gather approach for the Parboil Histogram benchmark is only effective for GPUs with sufficient shared memory space to privatize a reasonable portion of the output histogram. The S1070 system does not have sufficient scratchpad capacity for the scatter-to-gather transformation to improve histogram benchmark performance, and is therefore inapplicable as an optimization for that architecture.

## 3.4 Binning

A gather operation can be difficult to orchestrate without a method of determining, based on output location, which inputs contribute to that location. In the Parboil Histogram benchmark, for instance, a set of work-groups redundantly reads a section of the input data from off-chip DRAM, but each only processes the set falling within its own output range.[1] In general, the bandwidth cost of reexamining data scales with the amount of parallelism. Therefore, for some applications it is beneficial to first create a data structure creating a map from output locations to a small subset of the input data that may affect that output location, reducing the redundant reading of data. This data structure creation is called "binning", because it often reflects a sorting of input elements into bins representing a region of space containing those input elements. In the example of Figure 4, the unsorted data keys are examined and sorted into an array. If the input dataset were very regular, the sorting by key alone would likely create an efficiently accessible data structure. However, in the presence of irregularity, there will either be empty or overflowing bins for any fixed bin size, which should be addressed by some combination of the following two optimization patterns: regularization and compaction.

Table 6: Binning results

| Benchmark | Pattern performance impact |
|---|---|
| CutCP | 12.0× |

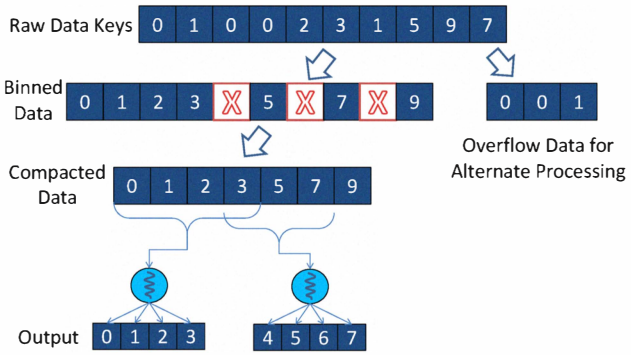


Figure 4: An example showing the binning, regularization, and compaction optimization patterns

Binning can improve system performance in several ways. If the GPU is performing both the binning and the computation, the overhead of binning can be outweighed by the improved efficiency of the main compute kernels. Alternatively, the binning operation could be offloaded to the CPU, potentially making better use of all available system resources. Binning is applicable in particular for the CutCP benchmark, as shown in Table 6. The speedups from binning are often very high, because binning input data for a kernel's input changes the fundamental computational complexity of the kernel algorithm. A scatter-based kernel may not need binning to get comparable computational complexity, but even for scattering kernels, binning is important because it can facilitate privatization of tiles of output data.

## 3.5 Regularization

Load imbalance has been one of the banes of parallel processing throughout its history. Typically load imbalance is exacerbated when the level of parallelism being exploited increases. Architectures exploiting SIMD or SIMT vector processing suffer from low-level imbalance if the tasks as-

[1] An example of a Scatter-to-Gather transformation without binning. Binning is not a useful technique for the Histogram benchmark because the actual histogram contribution is no more expensive in computation or bandwidth than the operation of sorting the data into bins would be itself.

Table 7: Regularization results

| Benchmark | Pattern performance impact |
|---|---|
| SpMV | 2.4× |
| MRI-Gridding | 2.62× |

Table 8: Compaction results

| Benchmark | Memory buffer size reduction |
|---|---|
| SpMV | 49% |
| MRI-Gridding | 68% |

signed to different execution lanes process different amounts or kinds of work. GPU architectures are no exception. Furthermore, if threads co-executing in a thread block have imbalanced loads, the shared resources of the entire thread block may be occupied until the last thread completes, potentially reducing the real amount of thread-level parallelism available for the architecture to exploit.

Some applications that exhibit load imbalance can predict at run-time where and how the load imbalance will occur. In the example of Figure 4, we assume that the program can count the number of data elements for each key for much less cost that actually calculating its contribution to its particular output. A preprocessing step can limit the amount of imbalance in work units executed on the GPU by identifying regions of load imbalance and proactively addressing them. In our example, during the binning process, elements that "overflow" a bin can be put in a separate data structure, which can be processed by some method less sensitive to load imbalance.

Regularization is the optimization pattern of preconditioning GPU kernel input to improve performance. Among the Parboil benchmarks, there are examples of processing work separately using a GPU kernel insensitive to imbalance, offloading irregular work for the CPU to process concurrently with the accelerated kernel. Other cases have no visible impact on the kernel code except that load imbalance and warp divergence are on average improved, resulting in higher performance. Regularization increases the efficiency of the primary accelerated kernels handling the majority of the processing, resulting in higher system performance overall, with impact as listed in Table 7.

## 3.6 Compaction

Compaction has been a technique within extremely parallel, shared-memory systems and programming models for quite some time as well. The fundamental issue is that when parallel work units produce a varying number of output elements into statically allocated output buffers, the buffer size must be overprovisioned. Because tasks determine output locations statically, unused holes or spaces in the output are the consequence of overprovision, such as those bins marked by X's in Figure 4. Output gaps interleaved with useful data cause bandwidth efficiency to drop for DRAM and cache architectures operating on transactions of larger, contiguous memory chunks. Compaction is a method of coordinating parallel tasks to dynamically determine output locations such that no holes are introduced.

If compaction were a separate processing step, as depicted in Figure 4, it would simply move all the useful data elements into contiguous addresses, filling in the holes, while keeping track of where each output section begins, as it will be data-dependent [1]. More often, and in the MRI-Gridding and BFS Parboil benchmarks where GPU computation produces compacted output, the compaction is integrated into the kernel producing output itself.

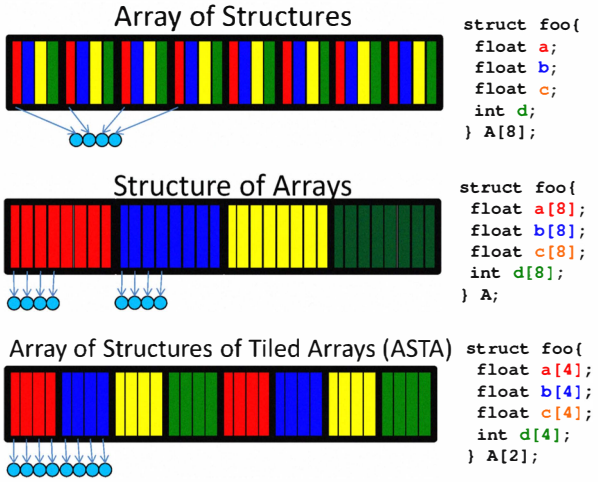


Figure 5: Data layout example for a collection of structures. Different layouts affect the coalescing of each warp access and the locality of multiple accesses.

The benefits from compaction primarily stem from the reduced memory footprint of the compacted data format. Performance impacts are typically only meaningful for bandwidth-bound kernels, and even then only minimally if the overprovisioned regions of the buffers are mostly contiguous. Thus, we quote not performance results but memory capacity reduction effects in Table 8. Compaction is essential for the MRI-Gridding benchmark in particular, for which we cannot even run uncompacted versions of reasonable datasets on most GPUs due to insufficient global memory capacity.

## 3.7 Data Layout Transformation

DRAM systems supporting both CPU and GPU architectures are designed to transfer data in large, contiguous lines or rows. Poor usage of CPU cache lines or GPU coalesced bursts will result in poor performance. However, GPU coalescing rules are somewhat harsher, because of the shorter time window over which the software could make use of a data burst from DRAM before any unused data is "dropped", requiring retransmission if needed at a future time. In some GPU architectures, the window is instantaneous, only exposed to a single SIMD instruction. More recent architectures introduce a small degree of caching extending this window, but because of the high degree of threading and the cache's low capacity, the window in practice is still very small. This is in contrast to CPU cache lines, which will typically sit in the cache for a longer period of time before being replaced.

Programmers work within the DRAM system design with well chosen data traversal orders or task index organization. If the elements in question are single-word data and closely associated with task indexes, a good choice of task index to element index mapping is typically sufficient to get good

Table 9: Data Layout results

| Benchmark | Pattern performance impact |
|---|---|
| LBM | 11.0× |
| SpMV | 1.21× |

memory system performance. However, that pleasant situation is not always feasible. Sometimes, the data elements needed within a particular time window are not naturally adjacent to each other in the memory address space. Take, for example, the diagram in Figure 5, which shows a warp accessing fields from a set of cells for various layouts. In the top case, using C standard data structure layout, the warp access addresses with a large stride between them, requiring multiple memory lines of mostly unused data to fulfill the requests. The middle case of Figure 5 shows equivalent accesses with a structure-of-arrays layout, a common transformation. However, even the middle case results in a large distance between the addresses of two fields, which are likely to happen very close together in time.

Depending on the memory system design, performance can be improved further by more complex layout transformation [9], perhaps resulting in a layout like that depicted at the bottom of Figure 5 where accesses to multiple fields will request adjacent, contiguous regions of memory. Specific examples of data layout transformation in the Parboil benchmarks include several instances of array-of-structure to structure-of-array transformations, a matrix transposition in SGEMM, and a transposed sparse matrix data storage format in SpMV. We specifically isolate the data layout transformation effect for the LBM benchmark, with an order-of-magnitude speedup as shown in Table 9. Overall, transformations for the purposes of achieving coalescing often achieve very high performance gains, such as the LBM, while layout transformations for improving memory level parallelism or avoiding moderate partition camping can effect a more modest improvement. Sung et al. report speedups ranging from 5% to 30% for already coalesced accesses in different benchmarks [9].

## 3.8 Granularity Coarsening

Granularity coarsening has been anecdotally described in many application optimization papers, perhaps most rigorously by Volkov in regards to linear algebra kernels [10]. When a larger task is decomposed into a set of fine-grained work-items, there is almost invariably some amount of overhead introduced in the problem decomposition. The overhead may vary for different algorithms and kernels, but almost every kernel will exhibit some inefficiencies in recalculating values like address offsets or other seemingly "small" operations in many threads. The finer the decomposition, typically the larger the overhead incurred. In addition to innate implementation inefficiencies, most real systems incur some fixed costs creating or scheduling parallel tasks, and communication operations tend to become more costly as the number of communicating tasks grows.

The CUDA and OpenCL programming models lend themselves to an "elemental" style of decomposition, where the source code of the kernel is scalar, processing a single element, with as many threads created as there are elements to process. With this extreme level of decomposition, the level

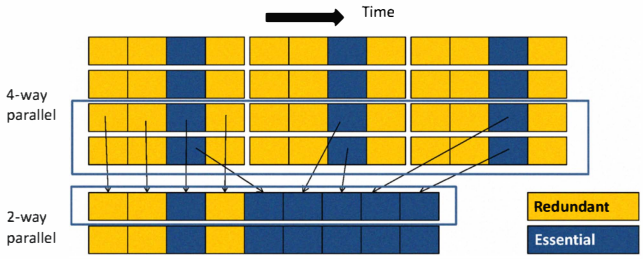


Figure 6: Granularity coarsening and resulting efficiency gains. Each shaded box represents an executed instruction or operation.

Table 10: Granularity Coarsening results

| Benchmark | Pattern performance impact |
|---|---|
| SGEMM | 1.96× |
| CutCP | 1.3× |

of redundancy and other inefficiencies can be surprisingly high, but difficult to address within the elemental-function methodology as the cost of communicating between different tasks is still higher than the cost of redundant computation.

Granularity coarsening is essentially a de-parallelization of a program. Instead of executing code where each thread processes one element, each thread processes several. Figure 6 shows a coarsening transformation by a factor of six. By putting several threads together, redundant operations that were previously executed once by each original thread have their redundant executions reduced by a factor of the degree of coarsening. Furthermore, what had been shared reads or conflicting writes to a variable in the untransformed code become private uses of data. In the example of Figure 6, although task parallelism was reduced by a factor of six, the total number of operations required to compute the full output was reduced by nearly two-thirds. The efficiency gains make incremental coarsening worthwhile so long as the amount of parallelism is still sufficient to occupy the parallel resources of the device. Examples of specific efficiency gains are shown in Table 10.

## 3.9 Summary

Table 11 shows a compact representation of which optimization patterns were relevant for each benchmark. Note that the table does not convey the relative importance of each optimization pattern to each benchmark. In our experience, a given benchmark's performance improvement due to optimization is typically dominated by one or two optimizations, with others making smaller contributions. Also, note that certain optimization patterns are widely applicable, such as granularity coarsening, while others are only applicable to applications with certain characteristics, such as binning. Finally, we would like to point out that some of the optimization patterns are clustered. Regularization and compaction, for instance, are typically combined rather than applied separately, because both are applicable for similar workload characteristics.

We are not necessarily convinced that these optimization

Table 11: Applicability of optimization patterns to each benchmark

| Benchmark | Tiling | Privatization | Scatter to Gather | Binning | Regularization | Compaction | Data Layout Transformation | Granularity Coarsening |
|---|---|---|---|---|---|---|---|---|
| cutcp | X | | X | X | X | | | X |
| mri-q | X | | | | | | X | X |
| mri-gridding | X | | X | X | X | X | | X |
| sad | X | | | | | | | X |
| stencil | X | | | | | | | X |
| tpacf | X | X | | | | | | X |
| lbm | | | | | | | X | |
| sgemm | X | | | | | | X | X |
| spmv | | | | | X | X | X | X |
| bfs | | X | | | X | X | | |
| histo | | X | X | | | | | X |

patterns are a comprehensive list, and would not be surprised to see other application domains introduce optimization patterns not represented in our studied benchmarks. However, given the generality of the patterns that we have seen so far, we do suggest that at least some of these optimization patterns will be applicable to almost any GPU application workload.

# 4. PERFORMANCE IMPACT OF ARCHITECTURES

Having explored the workload and optimization characteristics demonstrated by the Parboil benchmarks, we can now discuss in more detail the impact of different GPU architecture features on workload performance. Such a study could be approached multiple ways, with each method leading to particular insights. Due to space constraints, we would like to focus on three primary methods of inquiry. In the first, we assume that GPU workloads are in a state of perpetual hardware-software co-design and optimization. Our experimental results under this methodology will show how optimized software targets new features in each successive hardware generation, and how architecture changes amplify the benefit of particular optimization patterns. Secondly, we examine the other end of the optimization spectrum, to see how a simple, unoptimized implementation of each of the Parboil benchmarks improves with successive hardware generations. These results will tell us how well implicit or compiler-targeted hardware features are finding ways to improve performance without explicit software support. Finally, we would like to compare the performance gains of code optimization for each architecture generation, to understand how different architectures change or preserve the optimization process. Unfortunately, the Parboil datasets for the MRI-Gridding benchmarks were large enough that the GPU global memory capacity of both the 9800 GX2 and the S1070 were insufficient to collect baseline results. This both highlights the necessity of the compaction optimization for this benchmark, and prevents us from analyzing its performance any further.

## 4.1 Hardware-Software Co-Design Results

We optimized each of the benchmarks for each of the GPU architectures studied, and recorded the speedups achieved by successive architecture generations in Figure 7. Because the implementations for the earlier generations were already optimized around most of the performance cliffs of those architectures, the advances made by successive generations are typically near the increase in raw bandwidth or instruction throughput. Architecture feature improvements with

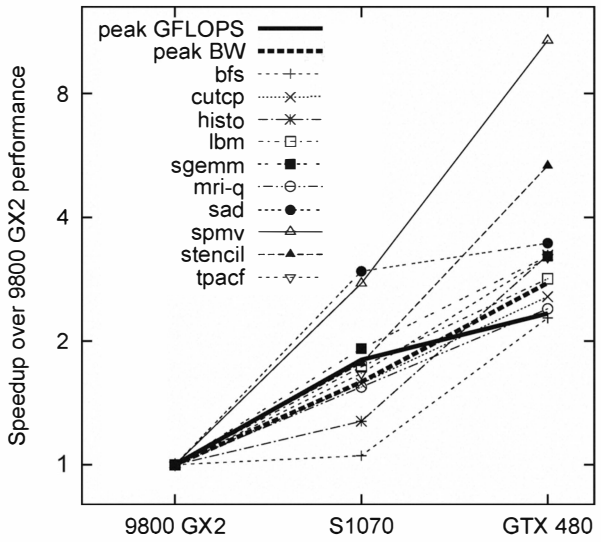


Figure 7: Performance of code optimized for each successive GPU generation, plotted against raw throughput and bandwidth scaling for comparison

a moderate impact on optimized workload performance include increased register file capacity, which boosted the performance of applications such as SGEMM and SAD in particular because of the extensive register tiling of those benchmarks. The performance improvement for register tiled benchmarks came less from the opportunity of additional register tiling, which reaches asymptotically low incremental benefits and had little impact in practice, but more from the architecture's ability to increase occupancy for the same degree of register tiling.

The single feature with the most performance impact overall was the global memory cache added in the GTX 480 generation. Even for optimized codes, scratchpad usage can introduce meaningful inefficiencies into the software. That overhead is typically overcome by the performance improvement due to captured locality otherwise unattainable in the absence of a cache, but does put scratchpad at a disadvantage to implicit caches for certain workloads. Furthermore, the GTX 480's cache captures what private scratchpads never can: shared locality among different thread blocks and access patterns with irregular locality. The spmv benchmark performance increases for the GTX 480 primarily from the caching of irregular accesses to the dense vector. The stencil benchmark benefits from caching because any mem-

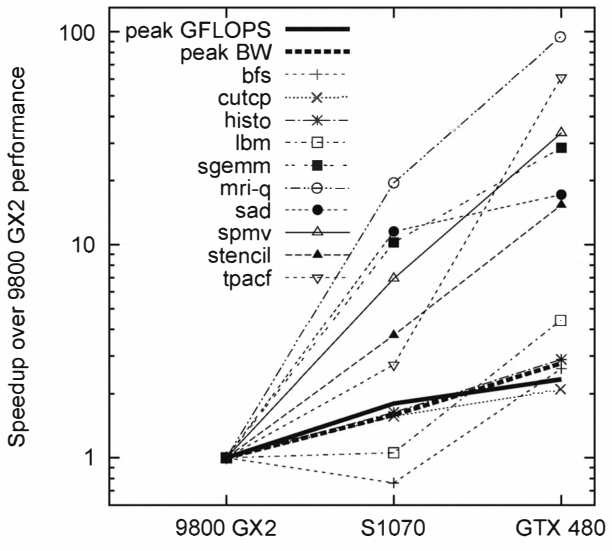


Figure 8: Performance of unoptimized code across GPU generations, plotted against raw throughput and bandwidth scaling for comparison

ory tiling approach in that benchmark must address the fact that the sizes of input tile needed to compute an output tile does not match the output tile size. Thread blocks sizes must be chosen to fit either the input or output tile size, resulting in inefficiencies from idle threads or increased software complexity for explicitly copying input tiles, respectively. In addition the tile borders overlap with the working sets of other thread blocks, exposing locality that cannot be captured with private scratchpad memory. The version of the stencil benchmark optimized for the GTX 480 actually avoids memory tiling, improving the efficiency of the instruction stream by relying on the cache and thread block scheduling policy to capture locality. The cache also significantly improves the BFS benchmark's performance by caching the end of the output queue while threads in a block incrementally add to its tail.

Surprisingly, atomic operations to shared memory had less performance impact than we expected. On further analysis, we found that privatization optimizations had reduced contention on shared memory locations requiring atomic updates to the point that the overhead of our software atomic updates, which scales with contention, was not so high as to make hardware assisted atomics indispensable. While our iterative atomic update methods were limited to certain situations, the versions optimized for the 9800 GX2 targeted those situations specifically, resulting in sufficient atomic update performance.

Finally, we note that the BFS benchmark in particular does not scale very well with regards to the number of SMs in the system. The BFS kernel that fills the GPU and performs device-wide barrier synchronizations in certain kernels does not perform as well on the S0170 as on the narrower 9800 GX2 and GTX 480 devices. As it is likely that machine widths will be increasing on average in the future, it seems reasonable to expect that using atomic operations for chip-wide synchronizations will become increasingly inefficient, and should perhaps be avoided if possible.

## 4.2 Baseline Performance Improvements

Figure 8 shows the performance improvement of a single, optimization-agnostic implementation across the different GPU generations, again plotted against the raw throughput and bandwidth improvements of the devices themselves. The definition of "unoptimized" is somewhat slippery, because it is always possible to write less efficient code by doing some kind of useless computation. Our philosophy while writing these baseline versions was to write the simplest functional code that seemed reasonable to us. We cannot claim that the baseline versions of all the benchmarks are equivalently unoptimized, but believe we can still learn some useful insights by paying attention to what "inefficiencies" are automatically mitigated or eliminated by particular architectures.

Generally, we can see that the performance trends are definitely positive, and significantly higher in magnitude than the improvement of optimized code versions. In several instances, one architecture generation brings order-of-magintude speedups over the previous generation, mostly for benchmarks with artificially poor memory bandwidth performance for uniform or misaligned accesses on the 9800 GX2 surging in performance when those limitations were removed in the S1070. The Fermi generation improved global memory broadcast accesses further by automatically promoting them to use the constant memory cache. Broadcast accesses are those where each thread in a warp loads from exactly the same address in a particular instruction. The GPU's constant cache supports this access pattern with very high performance. The constant cache design of the GTX 480 architecture enables the CUDA compiler to automatically transform accesses to use it under certain conditions, which reduces pressure on the general global memory cache and resultx in significant speedups for unoptimized mri-q, tpacf, and sgemm implementations.

Despite the raw bandwidth improvements of the S1070 over the 9800 GX2, the strided access pattern of the unoptimized lbm benchmark saw practically no performance improvement. It was not until the cache of the GTX 480 that its performance meaningfully improved. The GTX 480 cache also had significant impact on the performance of codes with had shared locality in the accesses among thread blocks that was not exploited by explicit memory tiling, in particular the stencil benchmark.

## 4.3 Optimization & Architecture Interactions

Finally, we examine the performance improvements of optimization for each benchmark and GPU generation, with results presented in Figure 9. Overall trend is significantly downward, implying that optimizations in general are becoming less critical over time. Conversely, we can say that many of the performance cliffs avoided by optimization are becoming less steep with successive architecture generations. However, there are some exceptions. The binning optimization pattern, exemplified by the cutcp benchmark in particular, results in consistently high speedups due to the change in fundamental algorithmic complexity, as should be expected. Also, while architectures are becoming slightly less sensitive to imperfect access patterns, good data layout remains extremely important, as exemplified by the lbm benchmark's 5× performance improvement from layout transformation, even on the Fermi architecture. For the sgemm benchmark, register tiling results in consistently high speedups. For

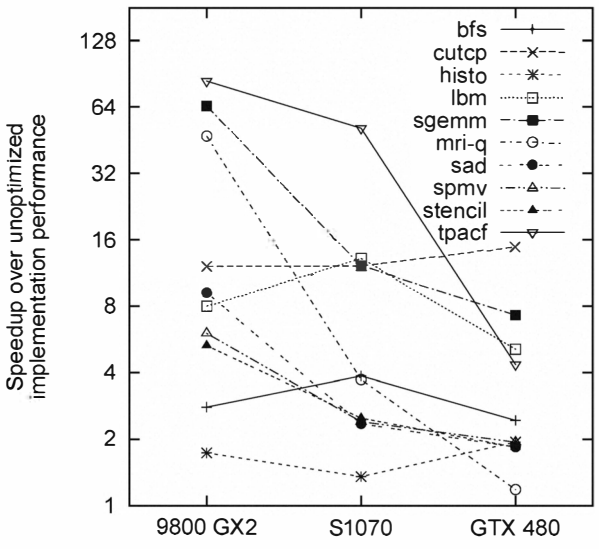


Figure 9: Speedup of optimizations for each GPU generation

such "simple" codes, the primary bottleneck is instruction stream efficiency: how many instructions compute necessary floating-point operations relative to how many instructions calculate addresses or move memory around. Even when artificial bandwidth inefficiencies are addressed by the Fermi architecture, a significant speedup can be expected from good register tiling.

# 5. CONCLUSIONS

Hundreds of articles have been published on optimizing applications for GPUs, and for good reason. In this paper, we have verified that for nearly all applications, hand-optimization has great performance rewards for GPU architectures. Additionally, those application optimizations are worth explaining and sharing, because certain patterns of optimization are applicable for a wide range of workloads. Each optimization pattern discussed in this paper was somewhat applicable for at least two benchmarks, and critically important for one or two as well.

We can also verify that some of the "worst" days of GPU computing are now behind us. Although legacy GPUs will still linger in the marketplace for several years, NVIDIA and other vendors seem to be getting on track with the design philosophy that unoptimized code exists, matters, and must be addressed. While major optimizations like binning or good choice of data layout should continue to be forefront in the minds of developers, others we are beginning to think of as good things to do if there is time instead of essential for getting any kind of reasonable performance.

Finally, based on experiments on past architectures, we can say with some confidence that if there were some way of making these optimization patterns unnecessary, it would have been done by now. Many of the optimization patterns could be applied to any parallel system, and are still relevant for today's multicore CPUs after decades of research and experience with high-performance parallel systems.

While innovation may still surprise us, it seems like manual program optimization, and in particular the optimization patterns we have presented in this paper, will continue to be relevant forparallel architectures in general, and GPUs specifically, for years to come. Software developers for high-performance applications would do well to brush up on these optimization patterns, and to continue to publish optimization insights either applying these general patterns to specific contexts, or possibly describing new optimization patterns.

# References

[1] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the 2009 Conference on High Performance Graphics*, HPG '09, pages 159–166, New York, NY, USA, Aug. 2009. ACM.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization 2009*, IISWC '09, pages 44–54, Washington, DC, USA, Oct. 2009. IEEE Computer Society.

[3] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable HeterOgeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, Mar. 2010. ACM.

[4] W.-m. W. Hwu, editor. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, San Francisco, CA, USA, Feb. 2011.

[5] W.-m. W. Hwu, editor. *GPU Computing Gems Jade Edition*. Morgan Kaufmann, San Francisco, CA, USA, Oct. 2011.

[6] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders. A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[7] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, Boston, MA, USA, first edition, Sept. 2004.

[8] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-m. W. Hwu. The Parboil benchmarks. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[9] I.-J. Sung, J. A. Stratton, and W. mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Conference on Parallel Architectures and Compilation Techniques*, pages 513–522, Sept. 2010.

[10] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, Nov. 2008. IEEE Press.