Policy-based Tuning for Performance Portability and Library Co-optimization

Duane Merrill
NVIDIA Corporation
Santa Clara
California
USA
dumerrill@nvdia.com

Michael Garland
NVIDIA Corporation
Santa Clara
California
USA
mgarland@nvidia.com

Andrew Grimshaw
University of Virginia
Charlottesville
Virginia
USA
grimshaw@virginia.edu

ABSTRACT

Although modular programming is a fundamental software development practice, software reuse within contemporary GPU kernels is uncommon. For GPU software assets to be reusable across problem instances, they must be inherently flexible and tunable. To illustrate, we survey the performance-portability landscape for a suite of common GPU primitives, evaluating thousands of reasonable program variants across a large diversity of problem instances (microarchitecture, problem size, and data type). While individual specializations provide excellent performance for specific instances, we find no variants with "universally reasonable" performance.

In this paper, we present a policy-based design idiom for constructing reusable, tunable software components that can be co-optimized with the enclosing kernel for the specific problem and processor at hand. In particular, this approach enables flexible granularity coarsening which allows the expensive aspects of communication and the redundant aspects of data parallelism to scale with the width of the processor rather than the problem size. From a small library of tunable device subroutines, we have constructed the fastest, most versatile GPU primitives for reduction, prefix and segmented scan, duplicate removal, reduction-by-key, sorting, and sparse graph traversal.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent programming; D.2.13 [**Software Engineering**]: Reusable Software; D.3.4 [**Programming Languages**]: Processors – *optimization*

General Terms

Performance, Design, Languages

Keywords

Performance, performance portability, software reuse, library design, auto tuning, policy, metaprogramming

1 INTRODUCTION

Parallel computing is fundamentally motivated by performance. However, absolute performance is not always an exclusive design objective. Sustainable software development also values design practices that emphasize program modularity, software reuse, and other productivity enhancing methodologies.

It can be difficult to develop modular programs for massively parallel machines like GPUs. Achieving high performance while maintaining good software abstractions is often challenging. The addition of parallelism creates further complexity. Despite the existence of many libraries of reusable CUDA kernels, we see little reuse of components within kernels themselves [22]. Furthermore, these libraries often require re-tuning effort for new GPU architectures. As GPUs evolve and existing programs mature, we are increasingly motivated to simultaneously satisfy the design objectives of (1) absolute performance and (2) portability and reuse, despite their often adversarial relationship.

We believe the dearth of reusable kernel components corresponds to a lack of performance portability. The absolute performance of many kernels is significantly influenced by problem type, problem size, and specific GPU microarchitecture. As a consequence, performance-sensitive applications derive little value from libraries of reusable device subroutines that cannot be tailored for the specific problem and processor at hand. In this paper, we present a software design methodology for *policy-based tuning* where authors of reusable software components express the "general shape" of their solutions, leaving many of the performance sensitive details unbound.

This approach was developed for the Back40 library of GPU computing [3]. To our knowledge, Back40 provides the fastest and most performance-portable implementations of reduction, prefix scan, segmented scan, duplicate removal, histogram, reduce-by-key, sorting of numeric primitives, and sparse graph traversal. These primitives are all constructed from a core library of reusable, tunable device subroutines for common kernel activities (e.g., workload management, data movement, variants of local reduction and prefix scan, etc.).

We motivate our policy-based design methodology with an examination of performance portability landscapes and the performance benefits of implementing flexible algorithmic granularity.

1.1 Investigation of performance portability

We evaluate the performance portability landscapes for the following archetypal sequence-processing primitives: copy-transform, reduction, prefix scan, and reduce-by-key. We explore the tuning spaces for these problems across a variety of data types, problem sizes, and NVIDIA microarchitecture (GF100, GT200, and G92). Our results show:

- A large performance variation among program instances representative of manually authored code.
- We can identify optimal or near-optimal program variants for many combinations of problem type and microarchitecture.
- No single program variant provides "universally reasonable" performance across all data types, problem

sizes, and architectures. While we expected over-fitting of program variants to individual problem instances, we were surprised by the lack of well-rounded program variants.

1.2 Policy-based tuning

These empirical observations motivate our software design methodology centered on policy-based tuning. The premise of policy-based tuning is to insulate both programmers and reusable software components from cementing implementation decisions having opaque performance consequences. By leaving such decisions unbound within the program text, we construct generic implementations that can later be specialized for specific problem instances and target microarchitectures. This allows us to obtain excellent absolute performance as well as performance portability from reusable software components.

The approach we describe is based on C++ metaprogramming. More specifically, our methodology incorporates procedural interfaces having parametric *policy types*. The text for a given procedure refers to these reflective policy parameters to describe how the compiler should expand, couple, and select from various phases of sequential and parallel computation. This provides us with the interface flexibility to adapt reusable components within the context of the enclosing implementation, problem, and processor.

1.3 Flexible granularity coarsening

Our policy-based tuning methodology was principally developed to permit *flexible granularity coarsening*, the expression of programs that accommodate a sliding scale of parallel versus sequential computation. By limiting the amount of concurrency expressed by the program, we force the costs of thread cooperation and parallelism to scale with the width of the processor rather than the problem size.

Flexible granularity can complicate parallel programs because the program text must combine code for both serial and parallel phases. Furthermore, we want to leave unbound both: (1) the number of steps each phase is to be run; and (2) the width of parallelism for each phase. Policy-based tuning allows us to specialize a program text by adapting the amount of exposed concurrency to individual processors, including those that may not yet be known.

Our work illustrates the benefit of matching appropriate task granularities with the width of the underlying hardware. At the global scheduling level, we show that increasing the granularity of work performed by threadblocks provides computational savings of 67% for global reduction, 42% for global prefix sum, and 27% for radix sorting passes. At the local level, increasing the sequential workloads of individual threads yields computational savings of 67% and 44% for intra-threadblock reduction and scan, respectively.

2 GPU COMPUTING

Contemporary processor architecture provides increasing parallelism in order to deliver higher throughput while maintaining energy efficiency. Modern GPUs are at the leading edge of this trend, provisioning tens of multiprocessor cores per chip, each of which manages on the order of a thousand hardware-scheduled threads. Each core employs data parallel SIMD (single instruction, multiple data) techniques in which a single instruction stream is executed by a fixed-size grouping of threads called a *warp*. A *threadblock* is a group of threads that will be co-located on the same core and share a local on-chip scratch memory. Parallel threads are used to execute a single program, or *kernel*. Coherence within shared memory spaces follows the bulk-synchronous parallel model [30].

3 TUNING AS AN EXPLICIT DESIGN METHODOLOGY

We can generalize the inherent challenges of parallel programming as stemming from two related sources: *expressing* parallelism, and *mapping* the expression of parallelism onto real hardware. The former encapsulates the creative aspects of devising and authoring a clean, concise, and correct description of parallel computation. The latter comprises the practical aspects of compiling and scheduling such descriptions of computation and data movement onto the underlying hardware for efficient execution.

The twin burdens of expression and mapping have historically fallen separately upon the shoulders of the programmer and the compiler/runtime, respectively. For sequential programs, compilers and dynamic CPU pipelines have largely succeeded in providing performance-portability without explicit guidance from the programmer.

However, the effectiveness of this arrangement is unlikely to continue as contemporary processor architecture embraces ever-increasing parallelism. As we discuss in this section, a philosophy of complete insulation from the mapping process is less useful for achieving both portability and performance. At worst, it is counterproductive. In particular, there are three aspects of mapping that would benefit from explicit guidance from the programmer: variable concurrency, algorithm selection, and resource scheduling.

3.1 Variable concurrency

Parallel computing adds an important facet to the process of mapping programs onto hardware: the amount of concurrency expressed by the program text. Many programming abstractions are designed for the program to specify *all available concurrency*. For example, SISAL [17], MultiLisp [13], and VHDL [14] are well-known declarative languages for expressing the data dependences that expose which computations can proceed in parallel. Similarly, the abstractions for data parallelism within frameworks such as OpenMP [7], CUDA [5], and MapReduce [8] require the specification of independent operations to be performed on every data element.

In this vein, GPU programmers are encouraged to construct data-parallel task decompositions that instantiate a unique logical thread for every data item. The abstract machine model supports this idiom through thread virtualization, i.e., the decoupling of logical threads from hardware threads. This idiom simplifies development: programmers need only express a single algorithmic strategy that encodes the smallest granularity of parallel computation.

On the surface, this idiom is also attractive for mapping programs onto hardware. First, the approach ensures that the concurrency expressed by a given program is both maximal and scales with problem size. These two properties are useful for achieving strong and weak scaling, respectively. Second, the idiom provides good portability. It abstracts away the physical details of processor cores and SIMD widths that may vary across GPUs. Finally, the oversubscription of processing elements with short-lived tasks helps ensure good load balancing and overall utilization.

However, this style of thread decomposition has important performance consequences for cooperative problems, i.e.,

parallelizations with sharing dependences. When logical threads scale with input size, so does the amount of communication through memory. Communication between logical threads often results in the same data being loaded back into registers on the same processor core, yet at the expense of many clock cycles and costly synchronization for correctness. We would prefer not to move such data at all. This implies that communication overhead should scale with physical processing elements, not problem size.

Furthermore, a portion of the overall instruction workload also scales with logical threads. Local computation within a threadblock typically involves computing conditional predicates, performing offset calculations, initializing local variables and shared memory, etc. Many of these operations are identical across threadblocks. For example, thread t_i in one threadblock is likely to have the same activation schedule and access the same shared memory locations as thread t_i in all other threadblocks. These identical instructions are effectively redundant when they are ultimately executed on the same SIMD lanes. When the number of threadblocks scales with problem size, this redundant computation does as well.

We can reduce the presence of unnecessary computation and communication by increasing the granularity, i.e., amount of serial work performed by each thread, warp, and threadblock. Our goal is to construct parallelizations where logical threads are a multiple of machine width, not problem size.

3.2 Algorithm selection

For many problems, no single parallelization is best across all processor architectures and input sizes, types, and data. The preference of one algorithm over another can depend on problem size and data type [2]. Ideally, we would like our compilers to be able to: (1) detect that a program implements a particular algorithm; and (2) synthesize an alternative parallelization that might be better suited for the underlying hardware.

However, it is extremely difficult to implement such compiler intelligence, particularly for problems having non-trivial data dependences. In the general case, it is impossible [24, 25]. This motivates programming methodologies having a less opaque relationship between the expression of the parallel program and its compilation, e.g., one in which the programmer explicitly supplies algorithmic alternatives and rules for guiding selection among them based upon problem type and target processor.

3.3 Resource scheduling

The challenges of mapping programs onto parallel hardware extend beyond algorithmic choice and granularity. Even when the basic outline of an algorithm is a good fit for the underlying machine model, an efficient scheduling of threads on one processor can result in significant underutilization on another. This is exacerbated on contemporary GPUs, where the hardware resources provisioned for each thread (registers, shared memory, etc.) are intimately intertwined with co-scheduling, i.e., the arrangement of threads within threadblocks and of threadblocks within multiprocessor cores.

Logical threads are dispatched onto processor cores by threadblock. The number of resident, active threadblocks per core is limited by the core's resources, namely the aggregate register file, local shared memory, and scheduling contexts. For example, the NVIDIA GF100 architecture provisions 32K 32-bit registers, 48KB shared memory, and scheduling resources for 1,536 threads per core. The configuration space for thread blocking is quite large, including such alternatives as:

- Three resident 512-thread threadblocks (1536 threads/core), 16KB shmem per threadblock, 21 registers per thread
- Six resident 128-thread threadblocks (768 threads/core), 8KB shmem per threadblock, 42 registers per thread
- Eight resident 64-thread threadblocks (512 threads/core), 6KB shmem per threadblock, 64 registers per thread

What should the program specify? The performance consequences are opaque. A higher number of resident threads per core does not necessarily imply greater throughput if computation or memory is already saturated. Larger residency also results in increased register pressure per thread and can result in costly spills to off-chip memory. Having a large number of small threadblocks can provide a greater diversity of instantaneous thread behavior for better core utilization. The same diversity, however, can be harder on read-only cache hierarchies. More resident threadblocks also reduces the amount of shared memory available to each threadblock for local cooperation.

Furthermore, these co-scheduling relationships explicitly affect the expression of thread behavior within program text. In particular, the degree of local parallelism affects the layout of shared memory within which threads communicate. On one hand, we can encode these relationships directly within our kernel programs, having each thread dynamically compute many of the derivative details it will need (e.g, offsets, strides, etc.) from parameters supplied by the host program. Alternatively, we can encode these relationships statically using the type system, allowing the much of this information to be computed at compile time.

3.4 Related work

Without precise analytical models for complex and data-dependent scheduling interactions on specific target architecture, the automation of empirical performance tuning (*autotuning*) is a common approach for program optimization. The tuning of sequential code has largely focused on various aspects of adaptive inlining and loop transformations. The former can increase the scope and quality of program optimization and the latter can improve the utilization of deep and diverse CPU cache hierarchies. [9, 12, 32]

Performance tuning for parallel programs has typically followed one of three methodologies. The first pairs a parallelizing compiler with an autotuning framework for mapping sequential loop nests onto parallel hardware. The considerations for both parallelization and tuning are often transparent to the programmer, or minimally influenced via code annotation or ancillary "recipes." [26, 28]

The second approach employs a separate metalanguage or code synthesizer to assemble program specializations from fragments of an explicitly-parallel language. Such frameworks are typically *ad hoc* in nature and/or are constructed for specific applications. [15, 16, 23]

Under the third methodology, the parallel programming language serves as its own metalanguage. Sequoia [11] and Petabricks [2] are example languages that provide their own mechanisms for expressing tunable parameters and variants. Our policy-based approach also falls within this category: we leverage template features of the CUDA C++ type system for constructing program text that is capable of manipulating its own compilation.

Our methodology has two important distinctions within this third category. The first is that we make use of reflective tuning types across procedural interfaces to facilitate co-optimization of

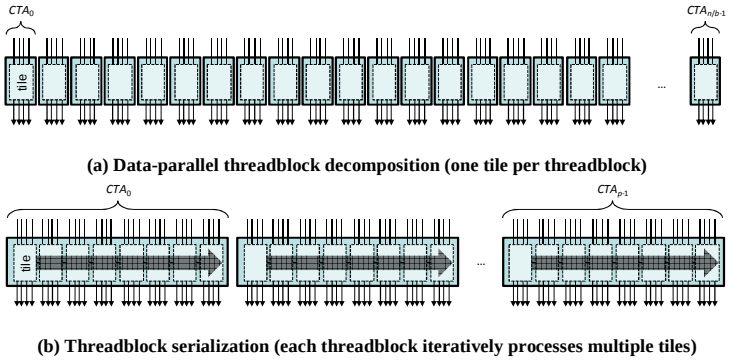


Fig. 1. Example threadblock decompositions for a data-parallel transformation. Tile size b=4 elements.

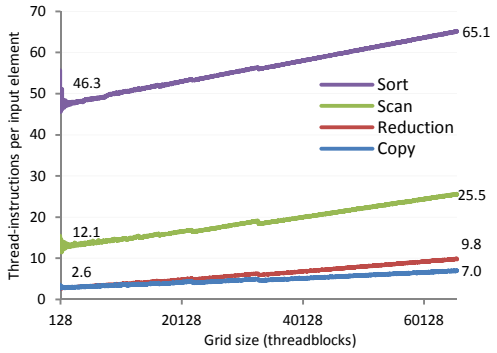


Fig. 2. Instruction overhead vs. threadblock granularity (GTX480)

reusable library components. The second is that we use the same type system for expressing tuning policy as well as for the data structures that govern the layout of shared memory. Thus we can express layout in terms of policy. This is particularly useful when authoring cooperative algorithms where a tunable number of threads must communicate with each other through shared memory while adhering to architecture-specific rules for avoiding bank conflicts.

The practice of increasing the granularity of work performed by threads and threadblocks is a common GPU performance optimization. The linear algebra implementations by Volkov *et al.* employ *register blocking* to improve instruction level parallelism by increasing the number of items locally processed per thread [31]. The global reduction implementations by Harris have threadblocks that process more than one tile[1] of input data [21]. Our prefix scan and sorting implementations invoke kernels whose threadblocks scale with the number of GPU multiprocessor cores [19, 20]. Similarly, the software graphics pipelines by Aila *et al.* [1] and Tzeng *et al.* [29] implement long running, *persistent threadblocks* that interact through global work queues. This paper investigates tunable granularity coarsening in the context of performance portability and software reuse.

4 GRANULARITY COARSENING

This section illustrates two important applications of granularity coarsening: *threadblock serialization* and *thread serialization*. The Back40 library of primitives makes extensive use of policy-based tuning to implement these two patterns.

4.1 Threadblock serialization

The CUDA programming model encourages data-parallel decompositions where the number of threads, and thus the number of threadblocks, scales with problem size. Fig. 1a illustrates this for a simple data-parallel transformation (e.g., copy). Each threadblock processes exactly one tile of data, typically where the number of data elements b in a tile corresponds to the number of threads in a threadblock. For a given problem of size n and scheduling granularity b, the kernel will launch a grid of $C = n/b$ threadblocks.

Fig. 1b illustrates alternative threadblock decomposition for the same data-parallel problem in which the number of threadblocks launched C is constant. The tile-processing logic for each threadblock wrapped within in a while-loop. When C is a fixed multiple of cores p, each threadblock is responsible for serially processing $O(n/(pb))$ tiles. Because C is $O(p)$, the number of logical threads scales with processor width instead of problem size.

We illustrate the effectiveness of this technique for a trivial data-parallel "copy" kernel. Threads simply read and write their 32-bit elements from global input and output arrays. We use 64M-element arrays, large enough to saturate the GTX480 memory subsystem. Fig. 2 plots the dynamic instruction overhead per input element as a function of the number of threadblocks launched by the kernel[2]. We vary the threadblock count from the minimum number needed to occupy the processor ($8p$=120 threadblocks) to fully data-parallel (n/b = 64K threadblocks where b=1024).

We observe that the computational overhead increases linearly with the number of threadblocks invoked. With fewer threadblocks, the computational savings from reduced concurrency and increased serial processing are substantial. Compared to the strictly data-parallel extreme on the right hand side, restricting the amount of concurrency to the width of the processor reduces the overall computational workload by 57%.

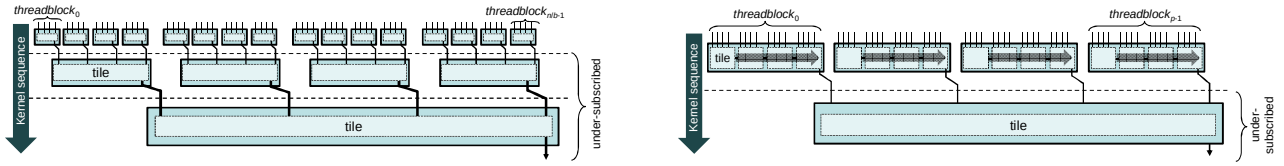
Two factors contribute to these savings. First, the reduced number of logical threads lowers the overall thread-setup overhead. This includes instructions for loading the kernel parameters into registers, computing the offset of the threadblock's first tile, the offset of the thread into that tile, etc. Second, the compiler can hoist operations out of the tile-processing loop, further reducing the workload per input element.

This threadblock serialization idiom is also particularly effective for recursive decompositions. Fig. 3a illustrates the traditional recursive data-parallel decomposition for parallel reduction. Each threadblock computes a partial reduction from its tile of b elements. The host program further invokes $\log_b n - 1$ reduction kernels to reduce these partial reductions into a single aggregate result.

However, GPUs are only efficient when the problem size is large enough to saturate the processor. This is rarely true for the interior of the reduction tree. For example, the second level of a 64M element reduction tree with branching factor b=1024 contains only 64K elements. Unfortunately the memory subsystem for the GTX480 only saturates for inputs larger than 8M elements. The second and third kernel invocations leave the GPU undersubscribed. Only the first kernel instance is capable of fully utilizing the processor.

[1] To avoid further overloading of the term "block", we use *tile* to describe a block of input data that a threadblock is designed to process to completion before terminating or obtaining another block of input.

[2] We normalize instruction counts per thread (as opposed to SIMD instructions per warp).



(a) $\log_b$-level tree: each threadblock processes one tile **(b) Two-level: threadblocks are reused to process multiple tiles**

Fig. 3. Example threadblock decompositions for global reduction. Threadblocks are comprised of four threads. Tile size b=4 elements.

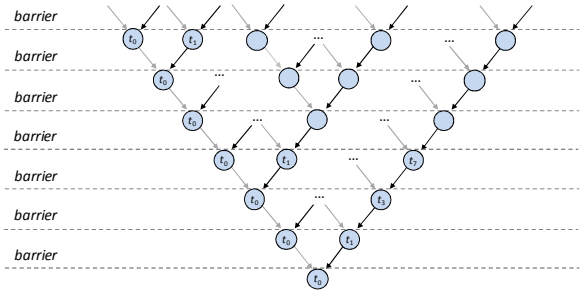


Fig. 4. Recursive, pair-wise parallelization of local threadblock reduction. Lighter dataflow arrows indicate partials left in registers

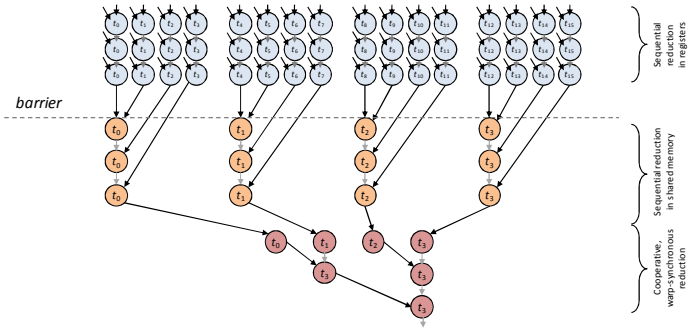


Fig. 5. Recursive, three-phase parallelization for locally reducing a tile of b = 64 elements where $p_{\text{threadblock}}$ = 16 and w_{SIMD} = 4. Lighter dataflow arrows indicate partials left in registers.

As an alternative, Fig. 3b illustrates the threadblock serialization idiom as applied to our reduction example [21]. In the first kernel, C threadblocks are given an even share of input tiles. Each threadblock sequentially processes its tiles, maintaining the accumulated partial reduction locally until its last tile has been processed. When C is a constant multiple of p, a single threadblock invoked by the second kernel can quickly reduce the C partial reductions output by the first kernel. This approach requires less global data movement and finishes the inefficient part of parallel reduction as quickly as possible.

The two-level threadblock serialization idiom extends to other cooperative, recursive parallelizations. Fig. 2 also illustrates the effectiveness of threadblock serialization for the cooperative problems of global reduction, prefix sum, and multi-way partitioning (for radix sorting) [19, 20]. By only invoking as many threadblocks as can be actively resident on the processor, we demonstrate computational savings of 67% for reduction, 42% for prefix sum, and 27% for partitioning.

4.2 Thread serialization

In this subsection, we discuss the merits of granularity coarsening for local cooperation with the threadblock. When expressed at their finest granularity, the task dependences for many cooperative parallelizations comprise binary trees of communication through shared memory spaces. Reduction and prefix sum are commonplace examples. At each timestep, the expressed concurrency is geometrically decreasing (or increasing). To illustrate, Fig. 4 presents a mapping of pairwise reduction onto parallel threads.

Despite its simplicity and abundant concurrency, this parallelization is quite inefficient on GPU architecture. Each of the b-1 reduction operators has an operand that needs to be written, synchronized, and read from shared memory. After performing an operator, threads must also evaluate a conditional to determine whether they will be active in the subsequent level. For example, a 1024-thread threadblock requires 4,224 thread-instructions[3] to reduce a tile of b=1024 elements.

A much better fit is the generic, three-phase construction illustrated in Fig. 5. Each phase seeks to either increase the amount of sequential work within a given storage class (e.g., registers, shared memory, etc.) or exploit a particular aspect of the abstract machine model (e.g., lock-step thread progress within the warp):

1) ***Sequential reduction in registers***. This phase decouples the tile size b from the number of threads $p_{\text{threadblock}}$. Each thread loads $b/p_{\text{threadblock}}$ items. It is important that this phase be wide enough to saturate the global memory subsystem with requests. The loaded elements are sequentially reduced in registers without read, write, and barrier instructions.
2) ***Sequential reduction in shared memory***. We place the partials from the previous step into shared memory, barrier, and then reduce the parallelism to the SIMD width w_{SIMD} of the processor core. One warp then serially *rakes*[4] over the shared partials for $p_{\text{threadblock}}/w_{\text{SIMD}}$ steps without write and barrier instructions.
3) ***Cooperative, warp-synchronous reduction***. Finally, the single raking warp performs a synchronization-free, pair-wise reduction in shared memory of the partial reductions computed in the previous phase. We exploit the lock-step SIMD behavior of threads within the same warp to avoid explicit barrier synchronization.

This construction only requires one barrier-synchronized exchange through shared memory that is accompanied by a single

[3] The actual width of the final five reduction levels is the warp-width w_{SIMD}=32, regardless of deactivated threads.

[4] *Raking* [4] is a strategy for assigning a set of threads p to process a much larger data set. Each thread is assigned an even-share of consecutive inputs to process serially, i.e., the stride between threads is p and the stride between elements for a given thread is 1.

Listing 1. A straightforward kernel sub-procedure for threadblocks to copy a tile of 32-bit floats from one global array to another

Template parameters: None
Formal parameters:
- Global input and output arrays *d_in*, *d_out*
- Offset *tile_offset* into *d_in*/*d_out* of the tile to be copied
- Optional limit *guarded_elements* on the number of tile elements to copy

Other:
- Global variable *thread_id* for thread identifier
- Global variable *cta_size* for threadblock-size in threads

```
1   __device__ void CopyTile(
2     float *d_in,
3     float *d_out,
4     size_t cta_offset,
5     size_t guarded_elements = cta_size)
6   {
7     if (thread_id < guarded_elements) {
8
9       // Load tile data
10      float data =
11        d_in[tile_offset + thread_id];
12
13      // Store tile data
14      d_out[tile_offset + thread_id] = data;
15    }
16  }
```

Listing 2. A tuning policy type for data-parallel copy, followed by an example parameterization of that type specialized for large-problems of 8-byte elements on the GF100 architecture.

```
1   // Tuning policy type
2   template <
3     // Problem instance type parameters
4     typename T,
5     int ARCHITECTURE,
6
7     // Tunable parameters
8     int LOG_THREADS,
9     int LOG_LOAD_VEC_SIZE,
10    int LOG_LOADS_PER_TILE,
11    ld::CacheModifier READ_MODIFIER,
12    st::CacheModifier WRITE_MODIFIER,
13    bool WORK_STEALING>
14      struct Policy;
15
16  // Example policy parameterization tuned
17  // for 8-byte data, large-size problems
18  typedef Policy<unsigned long long, GF100,
19    8, 7, 1, 0, ld::cg, st::cg, true>
20      LargeProblemPolicy8B;
```

conditional for reducing the degree of parallelism. All other steps are free of conditionals, and the bulk of the reduction operators (first phase) are free of any shared memory overhead. Compared with the pair-wise example, this three-phase construction only requires 1,440 thread-instructions to reduce a tile of b=1024 elements using a 128-thread threadblock with w_{SIMD}=32, a savings of 67%.

This example of local reduction serves to illustrate the importance of expressing the "general shape" of cooperation from multiple algorithmic phases. However, we do not want to bind these phases to any particular widths and depths when authoring our programs. In this example, the tile size, threadblock size, and warp size are the unbound tuning parameters that ultimately dictate the number of steps to statically unroll each phase. They also dictate the size and layout of shared memory needed for thread communication. We prefer to bind these parameters after empirically tuning for a specific problem and target architecture.

We also apply the same thread-serialization techniques for constructing local implementations of parallel prefix sum. The ability for parallel threads to cooperatively reserve space within shared data structures is a fundamental aspect of parallel computing. For GPU architecture, prefix sum is a much more efficient mechanism for implementing dynamic data placement than atomic instructions [18]. As a result of thread serialization, the Back40 implementations of local prefix sum exhibit a 44% reduction in dynamic instruction overhead from the recursive pairwise implementation within CUDPP [6].

5 TUNING VIA THE TYPE SYSTEM

Our design idiom for tuning via the type system uses C++ support for template-based meta-programming to ease the burden of granularity selection and algorithmic choice. We construct our parallel algorithms such that they can be specialized by tuning policy types.

By parameterizing kernel subroutines with policy types, we can author the "general shape" of an implementation, leaving many of the performance-sensitive details unbound. For example, we can use policy to specify the degree of parallelism, to govern algorithmic or threshold specialization, to dictate iteration and unrolling, and for declaring local variable types such as array sizes and shared memory layouts. Kernel subroutines can be reused by binding them with different tuning configuration policies that co-optimize them for the specific problem at hand.

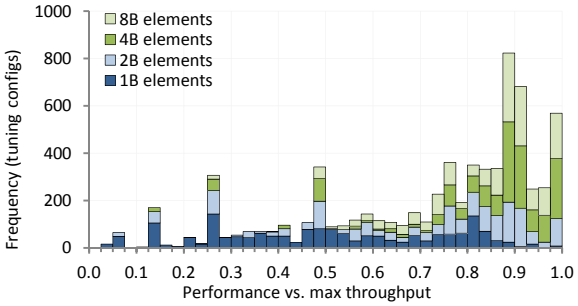
Because the policy is statically known to the compiler, we often obviate the need for any runtime decision-making with each logical thread. The cumulative overhead of runtime decision-making (e.g., how many loads to unroll) is particularly costly on GPU-like architectures having tens or hundreds-of-thousands of resident threads.

5.1 A simple example: data-parallel copy

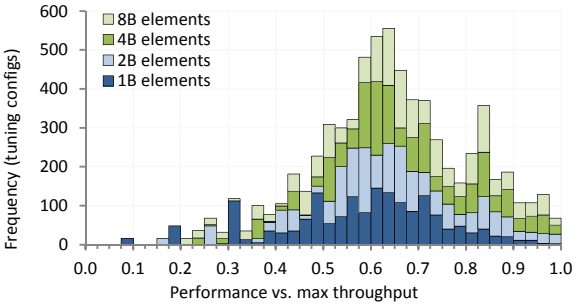
Consider data-parallel copy as a trivial example. As one of the simplest stencil kernels, threads simply load elements from a global input array and write them to equivalent locations within output array. Listing 1 illustrates a "concrete" tile-copying sub-procedure in which a threadblock copies a tile of 32-bit floats. Each thread loads and stores exactly one float.

In practice, the ostensibly-simple copy operation incorporates quite a few tuning decisions that are opaque in terms of their performance impact for any given architecture and problem type. Lines 2-14 in Listing 2 illustrate a parametric type *Policy* that can be specialized in the following tuning dimensions:

a) ***The number of loads per thread per tile***. This allows us to increase the number of outstanding loads issued before stores at the expense of increased register pressure. Reasonable configurations include 2^0, 2^1, and 2^2 loads per thread per tile.
b) ***The number of items per load***. Current NVIDIA GPUs support vector-loads of up to four component elements. Reasonable configurations include 2^0, 2^1, and 2^2 elements per vector load.
c) ***The number of threads per threadblock***. Reasonable configurations include powers-of-twos ranging from 2^5 to 2^{10} threads.
d) ***Work-stealing***. As algorithmic variants, we can either: (a) provide each threadblock with an even-share of input tiles; or (b) allow threadblocks to "steal" tiles of work using coarse-grained atomic-addition.



(a) Large problem size = 128MB, max throughput = 164 GB/s



(b) Small problem size = 128 KB, max throughput = 65 GB/s

Fig. 6. "Copy" kernel performance histograms of tuning configurations binned by normalized slowdown with respect to the maximum throughput achieved (NVIDIA GTX 480).

Listing 3. A generalized, policy-based kernel sub-procedure for threadblocks to copy a tile of elements from one global array to another.

Template parameters:
- Tuning policy type *Policy* as per Listing 2

Formal parameters:
- Global input and output arrays *d_in*, *d_out*
- Offset *tile_offset* into *d_in*/*d_out* of the tile to be copied
- Optional limit *guarded_elements* on the number of tile elements to copy

Other:
- Device function *LoadTileValid*() for reading each thread's tile portion
- Device function *StoreTileValid*() for writing each thread's tile portion

```
1    template <typename Policy>
2    __device__ void CopyTile(
3      typename Policy::T *d_in,
4      typename Policy::T *d_out,
5      typename size_t tile_offset,
6      typename size_t guarded_elements =
7        Policy::ELEMENTS_PER_TILE)
8    {
9      // Tile data
10     typename Policy::T
11       data[1 << Policy::LOG_LOADS_PER_TILE]
12           [1 << Policy::LOG_LOAD_VEC_SIZE];
13
14     // Load tile
15     LoadTileValid<Policy>(
16       data, d_in + tile_offset, guarded_elements);
17
18     // Store tile
19     StoreTileValid<Policy>(
20       data, d_out + tile_offset, guarded_elements);
21   }
```

e) ***Caching directives***. These modifiers affect cache behavior during loads and stores. Current NVIDIA GPUs expose up to four variants: default caching at L2 and L1 levels; no caching; cache in global L2 using smaller cache lines; and tagging for preferential eviction.

Listing 3 illustrates a templated copy subroutine that expresses the "general shape" of tile-copying. This procedure is not bound to a specific type of copy-element. In addition, each thread loads and stores a tunable number of elements. Such tuning details are encapsulated within the template parameter type *Policy*. Furthermore, Listing 2 (lines 18-20) presents an example policy type instance that has been tuned for copying large lists of 8-byte elements.

Fig. 6 illustrates the diversity of the corresponding performance landscape for the current NVIDIA GF100 architecture (GTX480). These tuning options enumerate a configuration space of 1,728 tuning variants per data type, per problem size. We evaluate these specializations for a pair of "large" and "small" representative workloads: 128MB and 128KB. Furthermore, we explore the configuration space for 1-byte, 2-byte, 4-byte, and 8-byte data types for each problem size. We normalize the throughputs of each tuning configuration against the maximum observed for its problem size and plot the resulting slowdown histograms.

Table 1. Max achievable DRAM bandwidth (10^9 Bytes/s)

	GTX480	*GTX280*	*9800 GTX+*
Unidirectional (even-share)	163.4	135.6	67.8
Unidirectional (steal)	168.6	63.6	42.6
Bidirectional (even-share)	153.6	125.4	61.7
Bidirectional (steal)	163.7	85.3	55.5

The large problem size (Fig. 6a) is representative of datasets large enough to saturate the memory subsystem. In general, the GTX480 is somewhat forgiving at this problem size, i.e., it is skewed to the right. On average, 25% of all configurations achieve more than 90% of the maximum achievable throughput (164 GB/s). However, we observe that it is relatively much more difficult to achieve this performance when copying 1-byte characters. Only 2% of configurations achieve more than 90% of maximum on 1B problem instances.

The performance for the small problem size (Fig. 6b) is much more diverse. Only 6% of all specializations fall within 90% of the maximum throughput (65 GB/s). For the various problems discussed throughout this paper, we generally observe that it is comparatively harder to find tuning configurations that are well-suited to small, fleeting workloads.

We also observed the configurations corresponding to the straightforward implementation specified in Listing 1 were not particularly competitive. For the large 128MB problems instances, the best 4-byte, 1-load, vector-1 configurations perform at less than 90% of maximum-achievable. For the small 128KB instances, these configurations only muster 65% of maximum-achievable. It is not obvious to the programmer that this "concrete" implementation would perform so poorly.

Finally, we use this tunable kernel to determine the maximum-achievable DRAM bandwidths for each of our three of our evaluation GPUs (GTX480, GTX280, and 9800 GTX+). We use these throughputs, listed in Table 1, to evaluate the absolute performance of memory-bound implementations.

Table 2. Performance-portability landscape benchmarks			
Benchmark	***Kernel tuning dimensions***	***Tuning configs per problem instance***	***Total sample evaluations***
Copy transform	Copy: a, b, c, d, e	1,728	124,416
Reduction	Upsweep: a, b, c, d Spine: a, b, c	8,748	104,976
Prefix sum	Upsweep: a, b, c Spine: a, b, c Downsweep: a, b, c	157,464	11,337,408
Reduce-by-key	Upsweep: a, b, c Spine: a, b, c Downsweep: a, b, c	157,464	11,337,408

Table 3. Between-configs slowdown variance (s^2_B)				
	GTX 480	*GTX 280*	*9800 GTX+*	*All GPUs*
Copy	0.52	0.08	0.48	0.40
Reduction	0.74	0.15	0.31	0.41
Prefix sum	0.58	0.42	0.31	0.83
Reduce-by-key	0.53	0.38	0.25	0.91

Table 4. Within-configs slowdown variance (s^2_W)				
	GTX480	*GTX280*	*9800 GTX+*	*All GPUs*
Copy	0.03	0.04	0.14	0.07
Reduction	0.03	0.04	0.11	0.06
Prefix sum	0.03	0.02	0.09	0.06
Reduce-by-key	0.01	0.01	0.03	0.02

5.2 Analysis of performance landscape across GPU architecture

In this section, we explore the cumulative tuning landscape for several data-parallel and cooperative problems across the last three generations of NVIDIA GPU architecture. Our evaluation is comprised of the following four benchmark problems: *copy-transform*, *reduction*, *prefix sum*, and *reduce-by-key*[5]. Global copy is the simplest performance proxy for any memory-bound data parallel transformation. Prefix sum is a performance proxy for kernels that compute recurrence relations or partition data (e.g., sorting). Reduce-by-key is a performance proxy for dataset contraction (e.g., list-compaction and duplicate-removal) and can be used to implement map-reduce computation (after mapping and sorting stages).

Table 2 lists the kernels that comprise each benchmark and the dimensions along which we can tune each kernel. For example, the *reduce-by-key* benchmark has three kernels, each of which can tuned by loads-per-thread, items-per-load, and number-of-threads-per-threadblock (*a*, *b*, and *c* from the previous section). With three kernels and 54 tuning specializations per kernel, the benchmark has an overall tuning domain of 157,464 tuning configurations.

Our investigation evaluates how different tuning policies respond to different problem instances (where a problem instance is a specific combination of data type, problem size, and GPU architecture). We evaluate the performance of each tuning configuration across a sample space of 72 problem instances constructed from combinations of the following:

- Four data types (1-byte, 2-byte, 4-byte, and 8-byte elements)
- Six problem sizes (128 KB, 512 KB, 2MB, 8MB, 32MB, and 128 MB)
- Three GPU architectures (NVIDIA GF100, GT200, G92 represented by GTX480, GTX280, and 9800 GTX+ GPUs)

We are interested in gauging how performance varies *between* configurations as well as *within* configurations. These two metrics intuitively correspond to the "strength" and "consistency" of individual tuning configurations, respectively.

We normalize our performance samples to the interval [0,1] so that we may generalize behavior across problem instances. For every problem instance, we identify the tuning configuration that provides the best sample performance. (For example, reducing 128 MB of 4-byte integers on GT200 maximally proceeds at 169 GB/s.) We then normalize the performance samples of all configurations for that problem instance in terms of relative slowdown against this "best" performance.

We use the statistical metrics *between-group variance* (s^2_B) and *within-group variance* (s^2_W) for analyzing the diversities of configuration strength and consistency, respectively [10]. The between-group variance is a measure of the variability of configuration means around the grand mean. The within-group variance is a weighted average of configuration variance, with weights determined by the number of problem instance samples in each configuration.

Between-group analysis. Table 3 and Table 4 present the between-group and within-group variances, respectively. The large ratios of s^2_B/s^2_W indicate that the broad majority of overall variation between pairings of configurations and problem instances is due to differences *between* configurations, i.e., certain configurations are innately better or worse than others. The performance-slowdown histograms in Fig. 7 graphically illustrate the ample performance variation amongst tuning configurations by binning configurations by their average slowdown.

Furthermore, Table 3 also reveals that some architectures are relatively more pliant than others. For example, the variances among tuning configurations are much lower for problem instances on the GTX280 than for the newer GTX480, particularly for the *reduction* benchmark.

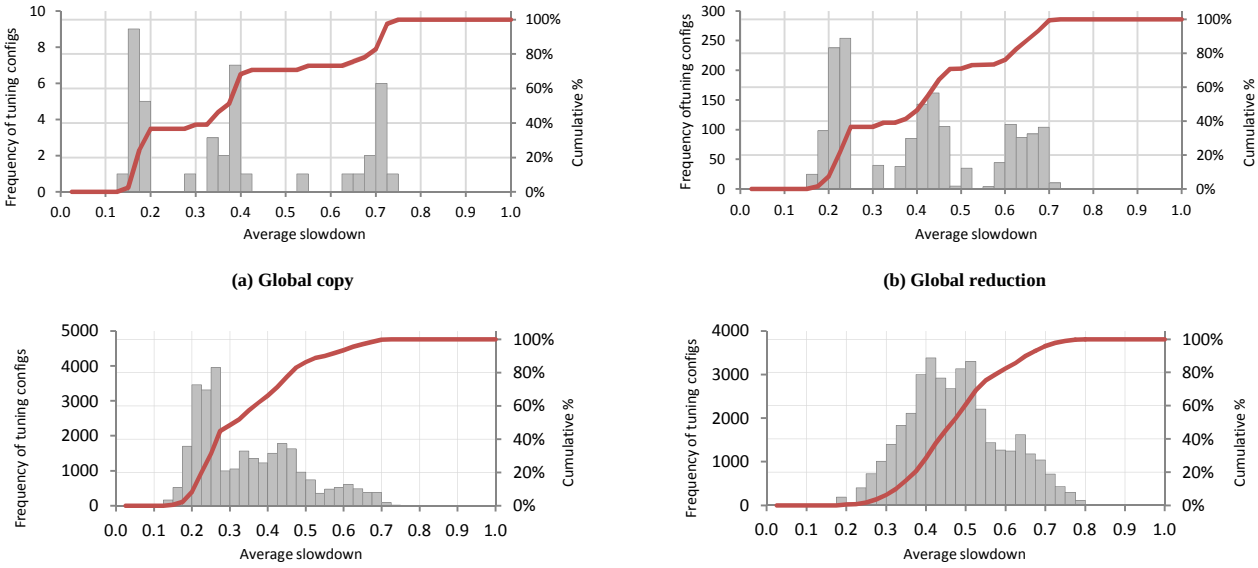
Within-group analysis. Despite being dwarfed by between-groups variance, the within-groups variance s^2_W is also fairly significant. For example, the within-groups deviation s_W for *prefix sum* across all GPUs is $\sqrt{0.6} = 24\%$. This implies that performance is also strongly related to problem instance, and that it will be relatively difficult to find tuning configurations that are universally better than others.

The histograms in Fig. 7 corroborate the absence of tuning configurations that perform well across the entire sample space of problem instances. "Well-rounded" tuning configurations do not exist. For example, no single configuration for *copy* averages more than 83% of the maximum-achievable performance across problem instances. For *reduction*, *prefix-sum*, and *reduce-by-key*, the best all-purpose configurations only average 73%, 73%, and 83% of what we can maximally achieve.

5.3 Effectiveness of auto-tuning

For large saturating problem sizes, we would like our memory-bound problems (namely *copy*, *reduction*, and *prefix sum*) to proceed at the maximum-achievable DRAM bandwidth for each device. Because of the heavily overlapped nature of the GPU, we would expect that all memory-bound specializations would yield equal performance. Table 5 reveals this not to be the case. It presents the average bandwidth utilization across pairings of

[5] Given a list of key-value pairs, reduce-by-key is analogous to a segmented reduction over the values where the segments are defined by regions of consecutive, identical keys.



(a) Global copy **(b) Global reduction**

(a) Global prefix sum **(b) Global reduce-by-key**

Fig. 7. Performance histograms of tuning configuration "strength." Configurations are binned by their harmonic mean slowdown across all problem instances. (For a specific problem instance, the slowdown for a given tuning configuration is relative to the maximum performance achieved by any configuration on that problem instance.)

Table 5. Average bandwidth utilization of *all* 128MB tuning configurations

	GTX480	*GTX280*	*9800 GTX+*
Copy	0.72	0.43	0.45
Reduction	0.61	0.32	0.35
Prefix sum	0.59	0.46	0.47
Reduce-by-key	0.31	0.16	0.16

Table 6. Average bandwidth utilization of *best* 128MB tuning configurations

	GTX480	*GTX280*	*9800 GTX+*
Copy	1.00	0.99	0.99
Reduction	0.96	0.88	0.95
Prefix sum	0.97	0.97	0.94
Reduce-by-key	0.67	0.38	0.33

configurations with 128MB problem instances, normalized to the DRAM bandwidth presented in Table 1. The three implementations that should be bandwidth-bound at this problem size are nowhere near maximum bandwidth utilization.

However, our autotuning search is quite effective at finding specific configurations that perform at peak or near-peak bandwidth. Selecting among only the best-performing configurations for each of the 128MB problem instances, Table 6 shows that we can identify policy configurations that perform exceptionally well for each data type (1B – 8B). Even for our compute-bound problem (*reduce-by-key*), our best-performing configurations are more than twice as fast.

We further illustrate the need for specialization by comparing our tuned global reduction kernels against those provided by the Thrust library of GPU primitives [27]. To this point, we have emphasized the mediocre performance of our average program variants. This raises the question of whether our average specializations are representative of concrete implementations "in the wild." The Thrust implementation of global reduction is a good point of comparison because it shares the same overall parallelization strategy.

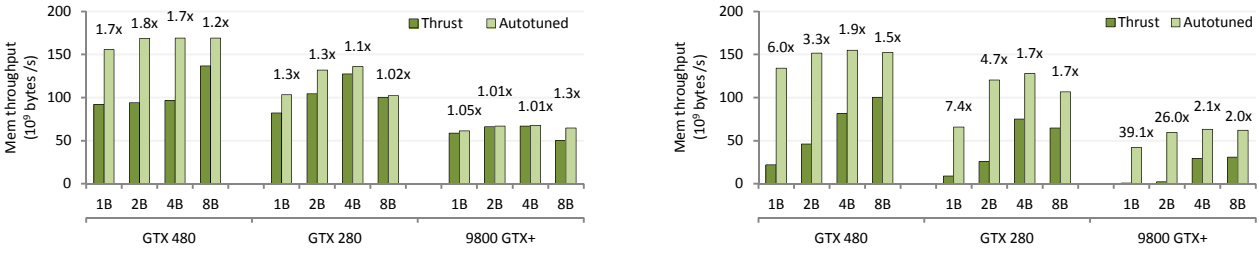
Fig. 8 illustrates our autotuned reduction performance advantage over the Thrust implementation for both saturating 128MB and fleeting 128KB problem instances. For large, GF100-based problems instances, the Thrust performances align with our average configuration performance. In relation, our tuned specializations achieve a harmonic mean speedup of 1.6x. Their large-problem performance is relatively much better for the older GT200 and G92 architectures. We only achieve 1.14x and 1.08x speedups for those GPUs, respectively.

Fig. 8b illustrates the importance of autotuning for small problem sizes. For this subset of problem instances, the Thrust performance is representative of our grand-mean configuration slowdown of 0.6 across all reduction problems. In relation, our tuned specializations achieve harmonic mean speedups of 2.4x, 2.6x, and 3.9x for the GF100, GT200, and G92 architectures, respectively.

6 CONCLUSION

In constructing the Back40 library of high performance CUDA primitives, it became clear that "concrete" implementations were simply not performance portable. Our tuning analyses illustrated the dire performance portability landscapes of such program instances, showing them to be incapable of delivering good performance across the domain of problem instances they might be expected to address. A recurring observation is the difficulty of achieving good performance from a single implementation on both large, saturating workloads and small, fleeting workloads.

To achieve performance portability, we developed a design methodology for *policy-based tuning* where reusable components express the "general shape" of their solution, leaving many of the performance sensitive details unbound. By incorporating policy types within procedural interfaces, we enable the co-optimization reusable software components with the enclosing kernel



(a) 128 MB problem instances

(b) 128 KB problem instances

Fig. 8. Global reduction performance comparison between our autotuned and the "concrete" Thrust implementations.

application. We found the C++ type system to be useful as a mechanism for specializing code generation via template metaprogramming, particularly as many tuning decisions affect data structure and layout within shared memory. Our autotuning results demonstrate the ability to consistently discover excellent specializations for the specific problem instance at hand.

An important application of such specialization is the selection of the proper granularity of concurrent work. We showed that parallelizations that achieve a proper balance between serial and parallel phases of computation provide significantly better efficiency and performance than those that simply express all available concurrency.

7 REFERENCES

[1] Aila, T. and Laine, S. 2009. Understanding the efficiency of ray traversal on GPUs. *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), 145–149.

[2] Ansel, J. et al. 2009. PetaBricks: a language and compiler for algorithmic choice. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), 38–49.

[3] Back40 Computing: Fast and efficient software primitives for GPU computing: *http://code.google.com/p/back40computing/*. Accessed: 2011-08-25.

[4] Blelloch, G.E. et al. Solving linear recurrences with loop raking. 416–424.

[5] CUDA: *http://www.nvidia.com/object/cuda_home_new.html*. Accessed: 2011-08-25.

[6] cudpp - CUDA Data Parallel Primitives Library - Google Project Hosting: *http://code.google.com/p/cudpp/*. Accessed: 2011-07-12.

[7] Dagum, L. and Menon, R. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*. 5, (Mar. 1998), 46–55.

[8] Dean, J. and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM*. 51, 1 (Jan. 2008), 107–113.

[9] Demmel, J. et al. 2005. Self-Adapting Linear Algebra Algorithms and Software. *Proceedings of the IEEE*. 93, 2 (Feb. 2005), 293–312.

[10] Devore, J. 1999. *Applied statistics for engineers and scientists*. Duxbury Press.

[11] Fatahalian, K. et al. 2006. Sequoia: programming the memory hierarchy. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006).

[12] Frigo, M. 1999. A fast Fourier transform compiler. *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (New York, NY, USA, 1999), 169–180.

[13] Halstead,Jr., R.H. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538.

[14] IEEE Computer Society 2009. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*. (2009), c1 –626.

[15] Klöckner, A. et al. 2011. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*. (Sep. 2011).

[16] Kurzak, J. et al. 2011. *Autotuning GEMMs for Fermi*. Technical Report #245. LAPACK Working Note.

[17] Mcgraw, J. et al. 1985. *SISAL: Streams and iteration in a single assignment language, language reference manual version 1.2*. Lawrence-Livermore-National-Laboratory.

[18] Merrill, D. 2011. *Allocation-oriented Algorithm Design with Application to GPU Computing*. University of Virginia.

[19] Merrill, D. and Grimshaw, A. 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.

[20] Merrill, D. and Grimshaw, A. 2009. *Parallel Scan for Stream Architectures*. Technical Report #CS2009-14. Department of Computer Science, University of Virginia.

[21] Optimizing parallel reduction in CUDA: 2007. *http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf*. Accessed: 2009-12-12.

[22] Owens, J.D. et al. 2008. GPU Computing. *Proceedings of the IEEE*. 96, 5 (May. 2008), 879–899.

[23] Puschel, M. et al. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*. 93, 2 (Feb. 2005), 232–275.

[24] Rice, H.G. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*. 74, 2 (1953), pp. 358–366.

[25] Rogers, H. 1987. *Theory of recursive functions and effective computability*. MIT Press.

[26] Rudy, G. et al. 2011. A programming language interface to describe transformations and code generation. *Proceedings of the 23rd international conference on Languages and compilers for parallel computing* (Berlin, Heidelberg, 2011), 136–150.

[27] Thrust - Code at the speed of light - Google Project Hosting: *http://code.google.com/p/thrust/*. Accessed: 2011-08-25.

[28] Tiwari, A. et al. 2009. A scalable auto-tuning framework for compiler optimization. *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2009), 1–12.

[29] Tzeng, S. et al. 2010. Task management for irregular-parallel workloads on the GPU. *Proceedings of the Conference on High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), 29–37.

[30] Valiant, L.G. 1990. A bridging model for parallel computation. *Commun. ACM*. 33, 8 (Aug. 1990), 103–111.

[31] Volkov, V. and Demmel, J.W. 2008. Benchmarking GPUs to tune dense linear algebra. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), 31:1–31:11.

[32] Vuduc, R. et al. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*. 16, (Jan. 2005), 521–530.