Convergence and Scalarization for Data-Parallel Architectures

Yunsup Lee[1], Ronny Krashinsky[2], Vinod Grover[2], Stephen W. Keckler[2,3], Krste Asanović[1]
[1]University of California at Berkeley, [2]NVIDIA, [3]The University of Texas at Austin
{yunsup,krste}@eecs.berkeley.edu, {rkrashinsky,vgrover,skeckler}@nvidia.com

Abstract

Modern throughput processors such as GPUs achieve high performance and efficiency by exploiting data parallelism in application kernels expressed as threaded code. One drawback of this approach compared to conventional vector architectures is redundant execution of instructions that are common across multiple threads, resulting in energy inefficiency due to excess instruction dispatch, register file accesses, and memory operations. This paper proposes to alleviate these overheads while retaining the threaded programming model by automatically detecting the scalar operations and factoring them out of the parallel code. We have developed a scalarizing compiler that employs convergence and variance analyses to statically identify values and instructions that are invariant across multiple threads. Our compiler algorithms are effective at identifying convergent execution even in programs with arbitrary control flow, identifying two-thirds of the opportunity captured by a dynamic oracle. The compile-time analysis leads to a reduction in instructions dispatched by 29%, register file reads and writes by 31%, memory address counts by 47%, and data access counts by 38%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

General Terms Algorithms, Performance

Keywords CUDA, GPU, Scalarization

1. Introduction

Programming parallel systems is inherently challenging, and over decades of research and development only a few models have attained broad success. *Single-program multiple-data* (SPMD) accelerator languages like CUDA [19] and OpenCL [21] have proven to be accessible and productive. These languages allow the programmer to write code for a single thread and then use explicit data-parallel kernel invocations to attain high performance. However, although the SPMD accelerator model is simple for the programmer, it can introduce many hidden overheads. For example, with a conventional CUDA compiler, approximately 30% of thread registers replicate data that is uniform across all threads executing a kernel. Similarly, approximately 30% of all instructions are entirely redundant across the threads.

Redundancy across threads is the key inefficiency that this paper addresses. We do so in the context of GPU *single-instruction multiple-thread* (SIMT) architectures [17], which fetch an instruction once but then execute it on many threads simultaneously using parallel datapaths. For example, NVIDIA's GPUs have a 32-wide *warp*, and AMD's GPUs have a 64-wide *wavefront*. Emerging GPU architectures have also added scalar execution resources alongside the parallel datapaths [1].

Our primary contribution is a compiler algorithm to *scalarize* both thread registers and instructions, such that there is only one per warp (or wavefront) instead of one per thread. Our compiler uses two interlinked analyses to enable scalarization. The first, *convergence analysis* statically determines program points where the threads in a warp are guaranteed to be converged (i.e. no thread is following a divergent control-flow path). Convergence analysis is critical for scalarization, since the compiler can only scalarize regions that it can prove to be convergent. The second is *variance analysis*, which statically determines which program variables are guaranteed to have the same (or thread-invariant) value across the threads in a warp. In Section 3, we construct an intuitive argument for something that was not immediately apparent when we began our work: convergence and variance information can be usefully analyzed together in the same pass. In fact the two are inseparable in our implementation. We present an algorithm that iteratively analyzes and propagates convergence and variance information over a kernel's program dependence graph (PDG) [10]. Scalarization then uses this analysis to convert private thread registers into scalar registers shared across the threads in a warp, and also converts thread instructions into scalar instructions that execute one operation per warp instead of one operation per thread. Using *affine analysis*, our compiler also

This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

CGO'13 February 23–27, 2013, Shenzhen, China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE…$15.00

generates *warp-sequential* loads and stores when the threads in a warp access sequential (unit-stride) data in memory. For example, a single warp-sequential load instruction can fetch a word from memory on behalf of each of the threads in the warp, placing the result in the same-named private register belonging to each thread.

Section 4 characterizes 23 benchmarks and finds that the compiler is able to keep warps converged for 66% of the total thread execution time on average. When augmented with simple *dynamic convergence preservation*, convergent execution can improve up to 97% of total execution time. Scalarization reduces thread register usage by 20–33% on average depending on warp size, making it possible to either support more threads or reduce the register file size. Furthermore, 24–31% of dynamic instruction operands are scalars. On average our compiler scalarizes 23–29% of dynamically dispatched instructions, reduces memory address generation counts by 37–47%, and eliminates data access counts by 30–38%. These savings can provide proportional energy and performance gains.

Section 5 describes how compiler convergence analysis and scalarization are generally applicable to a variety of accelerator architectures. We also discuss how our compiler analysis enables *stackless temporal-SIMT*, an architecture with the potential to bridge the performance and efficiency characteristics of MIMD (multiple-instruction, multiple-data) and SIMT processors. With stackless SIMT, each thread has a dedicated program counter (PC), allowing diverged threads to execute independently at full throughput as they would on a multithreaded MIMD processor. This model eliminates the need for divergence stacks, instead relying on *compiler-managed reconvergence*. With temporal-SIMT the threads in a warp execute temporally on a single lane, allowing the architecture to amortize instruction overheads. By simply configuring a range of the warp's registers to be shared scalars, scalar instructions can execute on the same datapaths as regular thread instructions.

2. Background

This section provides an overview of SPMD accelerator programming models, SIMT accelerator architectures, and the basic principles of scalarization.

2.1 Accelerator Programming and Architectures

SPMD accelerator languages express data-parallel computation in the form of multithreaded *kernels*. Inside a kernel the programmer writes code for a single thread, and a thread typically processes a small amount of data. For example, a thread might compute the color of a single pixel in a graphics application. The programmer expresses parallel computation with explicit data-parallel kernel invocations that direct a group of threads to execute the kernel code. In CUDA these thread groups are termed *cooperative thread arrays* (CTAs), and a CTA may have up to 1024 threads.

Explicitly data-parallel languages map naturally to highly multithreaded architectures, such as GPUs and other multicore accelerators. These throughput architectures leverage parallelism spatially to execute computations at a high rate across many datapaths and cores. They also leverage parallelism temporally to saturate high-bandwidth memory systems. The interleaved execution of multiple threads essentially hides hardware latencies from each individual thread. This approach simplifies the programming model since the code written for an individual thread can simply access data and operate on it, without great concern for the access latency.

SPMD programming models also implicitly expose locality that architectures leverage for efficiency. GPUs use a SIMT architecture that executes an instruction on parallel datapaths for many threads at the same time, for example 32 threads in *warps*, using NVIDIA terminology. A warp may issue in a single cycle if the datapath width matches the warp width, or it may be sequenced over several cycles on a narrower datapath. Similar to single-instruction multiple-data (SIMD) architectures, SIMT architectures use this organization to amortize the instruction fetch and other control overheads associated with executing instructions. SIMT architectures also derive efficiency from data locality for the common case when the threads in a warp access neighboring data elements. To exploit this locality, SIMT architectures use *dynamic address coalescing* to turn individual element accesses into wide block accesses that the memory system can process more efficiently, for example with only a single cache tag check.

2.2 Overheads of SPMD

A SIMT architecture is able to substantially reduce the program counter and instruction fetch overheads of multithreading, but many hidden overheads of the SPMD programming model remain. Writing kernel code for a single thread at a time is simple for the programmer and improves productivity, but with a conventional compiler this model can create a substantial amount of redundant work across threads.

Consider the simple FIR filter example shown in Figure 1(a) in which each thread computes one output element by convolving a range of `flen` input elements with an array of `flen` coefficients. The compiled code is shown in Figure 1(b). Each thread maintains both a loop iteration count (`r7`) and a loop end count (`r3`) in registers and uses counter increment (`b20`) and conditional branch instructions (`b24`) to execute the loop. Thus, each thread executes a substantial amount of bookkeeping overhead in addition to the actual multiply-adds that perform useful work. Furthermore, most of the bookkeeping overhead is entirely redundant across threads. Each thread maintains identical loop counts, calculates the same branch conditions, replicates the same base addresses, and performs similar address math to retrieve data from structured arrays.

```
__global__ void fir(float* samples,
                    float* coeffs,
                    int flen,
                    float* results)
{
  int idx = threadIdx.x;
  float result = 0;
  for (int i=0; i<flen; i++)
    result += (coeffs[i] *
               samples[idx+i]);
  results[idx] = result;
}
```

(a)

```
b01 BB_1:
b02   mov r9, r1;         # threadIdx.x
b03   ld.u64 r1, [4096];  # samples
b04   ld.u64 r2, [4104];  # coeffs
b05   ld.u32 r3, [4112];  # flen
b06   ld.u64 r4, [4120];  # results
b07   iset.s32.gt r5, r3, 0;
b08   mov r6, 0;          # init result
b09   @r5 bra BB_3;
b10 BB_2:
b11   bra BB_5;
b12 BB_3:
b13   shl r5, r9, 2;      # tidx * 4
b14   iadd r1, r1, r5;    # sample addr gen
b15   mov r7, 0;
b16 BB_4:
b17   ld.f32 r5, [r2];    # load coeff
b18   ld.f32 r8, [r1];    # load sample
b19   fma.f32 r6, r5, r8, r6; # fp mul add
b20   iadd r7, r7, 1;     # loop bookkeeping
b21   iadd r1, r1, 4;     # samples bookkeeping
b22   iadd r2, r2, 4;     # coeffs bookkeeping
b23   iset.s32.lt r5, r7, r3; # test loop break
b24   @r5 bra BB_4;
b25 BB_5:
b26   shl r5, r9, 2;      # tidx * 4
b27   iadd r4, r4, r5;    # result addr gen
b28   st.f32 [r4], r6;    # store result
b29   exit;
```

(b)

```
c01 BB_1:
c02
c03   @s ld.u64 s1, [4096];   # samples
c04   @s ld.u64 s2, [4104];   # coeffs
c05   @s ld.u32 s3, [4112];   # flen
c06   @s ld.u64 s4, [4120];   # results
c07   @s iset.s32.gt s5, s3, 0;
c08      mov r6, 0;           # init result
c09   @s @s5 bra BB_3;
c10 BB_2:
c11   @s bra BB_5;
c12 BB_3:
c13
c14
c15   @s mov s7, 0;
c16 BB_4:
c17   @s ld.f32 s5, [s2];     # load coeff
c18      ldwseq.f32 r8, [s1]; # load sample
c19      fma.f32 r6, s5, r8, r6; # fp mul add
c20   @s iadd s7, s7, 1;      # loop bookkeeping
c21   @s iadd s1, s1, 4;      # samples bookkeeping
c22   @s iadd s2, s2, 4;      # coeffs bookkeeping
c23   @s iset.s32.lt s5, s7, s3; # test loop break
c24   @s @s5 bra BB_4;
c25 BB_5:
c26
c27
c28      stwseq.f32 [s4], r6; # store result
c29      exit;
```

(c)

Figure 1. Simplified FIR filter code example: (a) kernel code, (b) conventional compiler output, (c) scalarizing compiler output. In the scalarized code, register specifiers which begin with `s` are scalar registers and `@s` is used to annotate scalar instructions. Register numbers are preserved between the conventional code and the scalarized code for clarity.

In addition to bookkeeping overheads, a SPMD program often has redundancy in the actual data operands accessed and computation performed by individual threads. The kernel code executed by each thread can be viewed as one iteration of an inner loop. A single-threaded encoding of the kernel often has "outer loop" data that could be accessed or computed once and then used many times. However in the SPMD program encoding, factoring out this redundant work is not as straightforward for a programmer or compiler. For example, in Figure 1, each thread loads the same coefficients redundantly (`b17`, `b22`) and replicates their storage in private registers (`r5`). As another example, a straightforward SPMD coding of matrix-multiply has each thread compute the dot-product of a shared vector (a row of the first matrix) with a private vector (a column of the second matrix). In this formulation, the load operations of the shared vector are redundant across all threads.

2.3 Scalarization

Redundancy across threads is the key inefficiency that scalarization targets. Figure 1(c) shows the scalarized version of the program in Figure 1(a). The compiler analysis required to generate this code will be described in detail in Section 3.

The compiler statically maps replicated operands to shared scalar registers. If we consider a single 32-thread warp executing the example in Figure 1, the conventionally compiled code would use 9 registers per thread. The scalarized code in comparison uses 2 private registers per thread and 6 shared registers per warp, 76% fewer registers per warp (70 vs. 288). In terms of dynamic register operands accessed, the conventionally compiled code reads 11 operands and writes 7 operands per thread per loop iteration. The scalarized code in comparison reads 2 private and 9 scalar operands per iteration and writes 2 private and 5 scalar operands per iteration. Since the scalar reads and writes only need to be performed once per warp, a 32-thread warp would read 79% fewer source operands (73 vs. 352) and write 69% fewer destination operands in total (69 vs. 224).

The compiler also converts redundant instructions to scalar instructions. As described above, while conventional SIMT architectures factor out instruction fetch overheads across a warp, each thread still executes each operation. In Figure 1, the conventionally compiled code executes 8 operations per thread per loop iteration. The scalarized code executes only 7 scalar (including warp-sequential) operations and 1 regular thread operation per iteration. Since the scalar operations only execute once per warp, a 32-thread warp would execute 85% fewer operations with the scalarized code (39 vs. 256).

The compiler also generates warp-sequential loads and stores for the input and output data that is accessed with unit-stride addressing across threads, as further described in Section 3.5. These accesses are *coalesced statically* by the compiler, eliminating the need for dynamic coalescing. In the conventionally compiled code, a total of 64 unique addresses are generated per warp per loop iteration, compared to only 2 addresses per warp for the scalarized code.

2.4 Divergence Management

A SIMT architecture executes instructions at warp granularity for efficiency, but it must also implement the independent thread execution semantics of the SPMD programming model. GPUs achieve this by maintaining a *divergence stack* for each warp, and by using *active masks* to disable inactive threads as the warp executes instructions. When the threads in a warp execute branch instructions, their execution is said to *diverge* if they branch in different directions. When divergence occurs, a warp is split into two subsets of threads, one for branch taken and one for branch not taken. One subset remains active and the warp's current active mask is updated to reflect that subset. For the other subset, an active mask is pushed onto the divergence stack. The warp continues executing the first subset of threads until it reaches the *reconvergence* point, for example the join point after an if-then-else clause. Then, the warp switches to the deferred subset of threads that are pending on the divergence stack. Once the second subset also reaches the reconvergence point, the warp active mask is restored to the original set of threads and reconvergence is achieved. Divergence and reconvergence nest hierarchically through these stack push and pop operations.

The divergence stack may be a hardware or software mechanism, or a combination. NVIDIA GPUs implement the stack in hardware, but the compiler is responsible for manipulating it in order to correctly implement independent thread execution semantics [17]. AMD GPUs use a software approach with explicit management of thread active masks. The scalar unit introduced in AMD's recent Graphics Core Next architecture primarily executes instructions to manage control flow and divergence [1]. Intel's MIC accelerator similarly handles divergence with software-managed predication [11].

3. Compiler Foundation for Scalarization

To identify redundancy across multiple threads, the compiler must prove that a variable has a uniform value across all of the threads in a group. This process requires two key analyses. First, *convergence analysis* proves that the threads are in a converged state, meaning that all of the threads in the group are in the same point in the control-flow graph at the same time. This analysis builds on the CUDA kernel invocation model in which threads are launched in an initial convergent state. It also assumes convergence at `__syncthreads()` (i.e. barrier synchronization) calls, which are in effect programmer supplied assertions that threads are converged.

Second, *variance analysis* determines which variables in the converged threads have the same (uniform) value across all threads. This analysis builds on the semantics that kernel function call arguments are thread-invariant. Variance across threads originates with use of thread indices (e.g. `threadIdx.x` in CUDA) and with volatile and atomic memory accesses. Our compiler uses data-flow and control-dependence analysis to determine which variables are not dependent on thread-specific values. Such variables can be converted safely from per-thread variables to per-warp scalar variables.

We implement the algorithms in the context of a production CUDA compiler, based on the LLVM infrastructure [15]. Our compiler algorithms are agnostic to the divergent execution models described in Section 2.4, and are generally applicable to SIMT architectures with scalar execution resources.

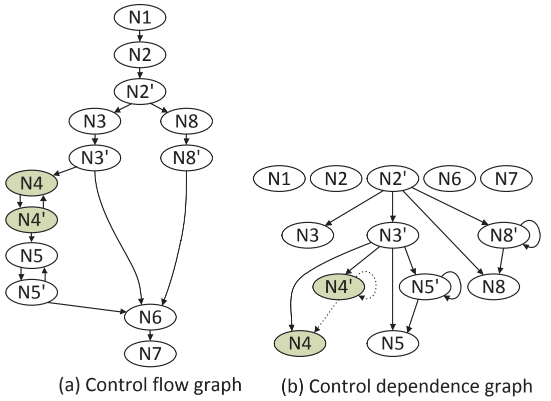
3.1 Convergence Analysis

A program point is considered convergent if and only if a thread-group barrier placed at that point can never fail. This property implies that either all threads in the group will arrive at the barrier, or none of the threads will. Note that reconvergence points found by an immediate post-dominator scheme may not be considered convergent, since our definition of convergent implies that all threads are fully converged rather than a subset being partially converged. Convergence may be defined with respect to a particular group size such as CTAs or warps.

To perform convergence analysis, we leverage two data structures common to compilers. First, the control flow graph (CFG) represents the program as a graph of basic blocks (BBs) connected via control flow (branch, jump) edges [24]. Instructions unrelated to control are encapsulated within the basic blocks. Figure 2(a) shows an example CFG containing conditional branch points, loops, and merges. Second, we leverage a standard global data-flow representation such as static single assignment form (SSA) [8] and the control dependence (CD) graph [10, 28] to identify basic blocks that are obviously convergent and determine a starting point for convergence analysis. Ferrante, et al. [10] define control dependence as follows:

> **Definition:** If X and Y are basic blocks in a CFG, Y is control dependent on X (written $X \prec Y$) *iff*
>
> 1. there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and
> 2. X is not post-dominated by Y.

Figure 2(b) shows the control-dependence relations in the CFG from Figure 2(a).



(a) Control flow graph (b) Control dependence graph

Figure 2. Example control flow and dependence graphs.

The simplest approach to convergence analysis is to use the control flow structure of the kernel. Entry and exit blocks of a single-entry-single-exit (SESE) region have the same convergence properties. If the entry of an SESE region R is convergent then so is its exit. We then use the notion of regions and its characterization as described in [3], where two blocks of a CFG are in the same region if both nodes have identical control-dependence predecessors. Such nodes are termed *control-equivalent*. Since all threads of a warp (and a thread block) are convergent at the entry block to the kernel, all blocks that are control-equivalent to the entry block must be convergent since they execute under the same control condition. Because the entry block where all threads in the kernel start has no control-dependence predecessor, all basic blocks with no control dependence predecessors are marked as convergent. Using this simple notion of convergence, it is easy to see from Figure 2(b) that blocks $N1$, $N2$, $N2'$, $N6$, and $N7$ have no predecessors and therefore must be convergent.

3.2 Combined Convergence and Variance Analysis

Leveraging variance analysis [26], we extend the simple convergence analysis above to identify when basic blocks across the threads are guaranteed to depend on the same condition. The key insight is that a basic block is convergent if and only if it is transitively control dependent only on convergent blocks whose branch condition is thread-invariant (written $Tinv(block)$ below) and that the entry block of the kernel is always convergent. Any result of a thread-invariant instruction is uniform and is a candidate for scalarization.

$$\forall b\ b \prec x : convergent(b) \wedge Tinv(b) \Rightarrow convergent(x)$$

Alternatively, a basic block is divergent if it is transitively control dependent on a divergent block or it is transitively control dependent on a block with a thread-variant branch condition (written $Tvariant(block)$).

$$\exists b\ b \prec x : divergent(b) \vee Tvariant(b) \Rightarrow divergent(x)$$

Our algorithm exploits the latter characterization to mark blocks as divergent after initially assuming, optimistically, that all blocks are convergent. This approach fits well with our combined variance and convergence analysis which starts with optimistic assumptions about thread-variance.

Figure 3 describes our optimistic algorithm for variance and convergence analysis. The first step performs initializations as follows (Figure 3(a)):

1. Optimistically mark every basic block of the kernel as convergent.
2. Optimistically mark every instruction as thread-invariant.
3. Initialize a worklist of instructions with those that read the thread id register, perform an atomic action on shared memory, or access volatile memory.

```
$worklist \leftarrow \emptyset$
for $bb \in blocks(kernel)$ do
    $Conv(bb) \leftarrow True$
    for $instr \in instructions(bb)$ do
        $Invariant(instr) \leftarrow True$
        if $instr\ reads\ thread\ id$ then
            $worklist \leftarrow worklist \cup \{instr\}$
        end if
        if $instr\ is\ an\ atomic\ instruction$ then
            $worklist \leftarrow worklist \cup \{instr\}$
        end if
        if $instr\ accesses\ volatile\ memory$ then
            $worklist \leftarrow worklist \cup \{instr\}$
        end if
    end for
end for
```

(a) Initialization.

```
while $worklist \neq \emptyset$ do
    $instr \leftarrow POP(worklist)$
    $Invariant(instr) \leftarrow False$
    for $s \in DataFlowSucc(instr)$ do
        if $Invariant(s) = True$ then
            $worklist \leftarrow worklist \cup \{s\}$
        end if
    end for
    if $instr\ is\ a\ conditional\ branch\ instruction$ then
        for $bb \in IteratedControlDependenceSucc(instr)$ do
            if $bb\ doesn't\ have\ a\ \_\_syncthreads()\ call$ then
                if $Conv(bb) = True$ then
                    $Conv(bb) \leftarrow False$
                    for $i \in instructions(bb)$ do
                        $worklist \leftarrow worklist \cup \{i\}$
                    end for
                end if
            end if
        end for
    end if
end while
```

(b) Analysis and propagation.

Figure 3. Combined convergence and variance analysis.

The worklist always consists of currently known thread-variant instructions and is seeded with those instructions that cannot be proven to be thread-invariant. The second step performs a fixed-point loop in which each step removes an instruction from the worklist and performs the following actions until the worklist is empty (Figure 3(b)):

1. Mark the chosen instruction, i, as thread-variant, and
2. Add every thread-invariant data-flow successor instruction of i in the SSA graph to the worklist.
3. If instruction i is a conditional branch instruction, propagate divergence to all convergent blocks that are iteratively control dependent on i but do not contain a barrier instruction. Add every instruction in blocks that are newly marked as divergent to the worklist.

When the algorithm terminates, any blocks that are marked convergent must be so; and any instructions not visited and marked as thread invariant must be so as well.

3.3 Analysis Example

To illustrate the algorithm, we use the flowgraph in Figure 2(a). This example assumes that the branch condition in the basic block $N4'$ is thread-variant. The corresponding control dependencies are reflected in Figure 2(b) with dotted lines. Note that the $IteratedControlDependenceSucc$ of instructions in $N4'$ is $\{N4', N4\}$. The algorithm in Figure 3 will propagate divergence to the targets of these control dependencies transitively, illustrated in Figure 2(b) by marking dark divergent blocks $N4'$ and $N4$.

After the algorithm terminates, all the blocks which are not marked as divergent (light colored in Figure 2) are convergent. Block $N5$ is inferred as convergent simply because N5 is control independent of $N4'$ (in the control dependence graph), which means that all diverged threads must pass through $N5$.

3.4 Convergence of Warps that Exit Early

In many CUDA applications, threads in a CTA may exit early based on tests that check for the thread id. In the following, all threads with `threadIdx.x` greater than 3 return and wait at the kernel exit for the rest of the warp to arrive.

```
__global__ void f() {
  if (threadIdx.x <= 3) {
    S1; } }
```

The compiler can safely assume that S1 is convergent since the remaining threads are at the exit, and any scalar registers which are otherwise holding thread-invariant values are safe to initialize in statement S1. We extended our convergence analysis algorithm to not propagate divergence information across control-dependent successors of conditionals if the exit block of the kernel is control dependent on the variant condition.

3.5 Affine Analysis

Affine analysis is used to determine if thread-variant address operands of load and store instructions can be converted to warp-sequential load and store instructions. Warp-sequential instructions access memory with a unit-stride address across successive threads. For example, given a load instruction:

```
ld.type Rx, [addr]
```

the compiler leverages both variance and scalar evolution analyses to determine if `addr` can be expressed as a simple linear expression of the following form:

`base` + bitwidth(`type`)*`threadIdx.x`

Such load instructions can be transformed into a warp-sequential instruction:

```
ldwseq.type Rx, [base]
```

in which register Rx in each thread of the warp is written with the respective value in the loaded vector corresponding to the thread's index.

4. Implementation and Evaluation

We evaluate our compiler using CUDA benchmarks from Rodinia [4] and Parboil [27]. Benchmarks in these two suites cover compute-intensive scientific domains including bioinformatics, image processing, medical imaging, graph algorithms, data mining, physical simulation, and pattern recognition. We reduced the input dataset sizes in some cases to make simulation time manageable. We also modified the source code to change texture references into global memory references, since our target abstract architecture lacks texture caches.

The modified CUDA LLVM compiler first generates PTX instructions annotated with convergence information for each basic block, and variance and affine information for all registers. The PTX source code is processed by our backend compiler to target the RISC-like machine ISA shown in Figure 1. The convergence, variance, and affine information is used by the backend compiler to map invariant values to scalar registers, and to mark redundant instructions as scalar instructions. Instruction scheduling and register allocation are also performed in the backend.

We run the compiled code on our in-house simulator to get a detailed breakdown of instructions issued, operations executed, register reads and writes, memory address counts and data access counts. Our simulator runs one kernel (i.e. one grid invocation) at a time. We execute PTX source code on Ocelot [9] to obtain reference memory dumps before and after each kernel launch. The initial memory dump is used to populate the initial memory state of the simulator, and the post-kernel launch memory dump is used to verify the kernel execution. Each benchmark's composite kernel runs are summed together for all results presented in this paper.

4.1 Convergence Analysis Results

The quality of convergence analysis is critical for scalarization, as the compiler can only scalarize regions that it can prove are convergent. Convergence analysis is also important for managing reconvergence in a stackless SIMT architecture later discussed in Section 5.2. Figure 4 shows the effectiveness of our compiler analyses. The benchmarks on the X-axis are sorted left-to-right in decreasing effectiveness of compiler convergence analysis. The Y-axis represents the total instructions dynamically dispatched for execution by the microarchitecture. For each benchmark, the left bar shows the breakdown of instructions proven convergent by different variants of the compiler. The right bar shows the fraction of instructions that could be proven convergent by a dynamic oracle. The fraction of the bars labeled *Diverged* cannot be proven convergent at compile time.

Simple convergence analysis, which only looks at the shape of the control flow graph, can only keep thread execution convergent 32% of the time on average. By coupling convergence analysis with variance analysis, the compiler is able to determine cases where branch conditions are invari-

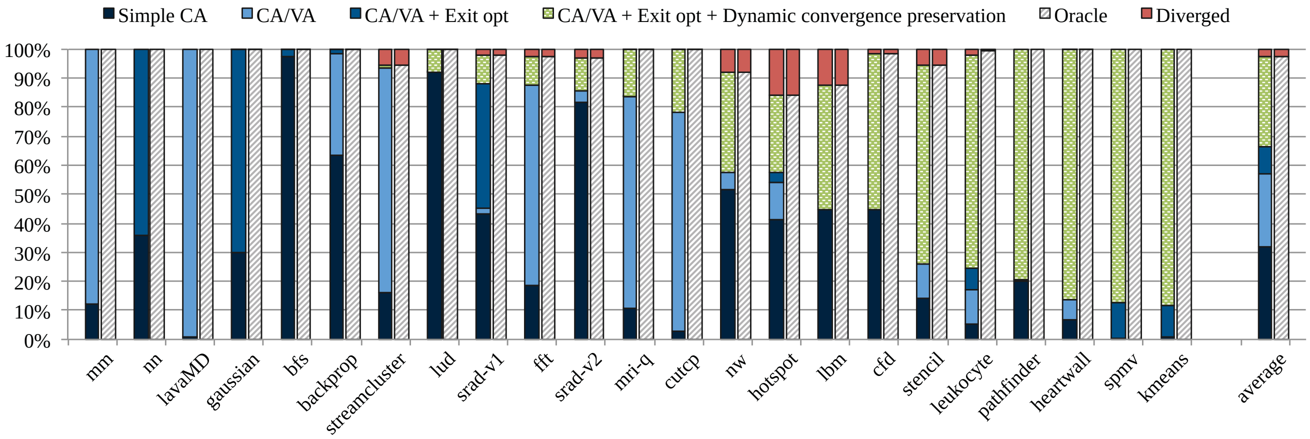


Figure 4. Effectiveness of convergence analysis with warp size of 4 threads. The X-axis is sorted by the effectiveness of convergence analysis done by the compiler. CA =Convergence Analysis, VA=Variance Analysis.

ant across threads, increasing convergent execution to 57%. The exit optimization further increases convergence to 66%. We also show results for dynamic convergence preservation, a simple hardware mechanism that prevents warps from diverging when threads dynamically branch in the same direction (note that this mechanism does not imply a hardware divergence stack). Figure 4 shows that this hardware and software approach improves convergent execution to 97% on average with warp size of 4 threads. With a wider warp size, convergence identified by the compiler will remain the same, while convergence imposed by dynamic convergence preservation will tend to decrease because the likelihood of a warp to diverge increases with more threads.

We also performed a limit study where we use oracle knowledge to maximize convergence. Optimal alignment of convergent blocks and instructions can be reduced to the Multiple Longest Common Sequence (MLCS) problem [18]. We reduced the complexity of our MLCS implementation by leveraging the compiler's convergence analysis and only analyzing divergent regions. Oracle convergence analysis based on the dynamic instruction trace shows that the best possible schedule can keep thread execution convergent 97% of the time on average. Overall, oracular analysis is no better than the combination of our convergence/variance analysis with dynamic convergence preservation.

4.2 Scalarization Results

Figure 5(a) shows the breakdown of static instructions into scalarized and unscalarized (labeled *thread*), normalized to a baseline without scalarization (left bar in each group). On average, the compiler scalarizes 29% of static instructions. The total static instruction count increases by 2%, primarily due to instructions generated to calculate the base address of warp-sequential memory operations. Figure 5(b) shows the breakdown of register accesses into the same categories, relative to the same baseline (left bar in each group), for warp sizes of 4, 8, 16, and 32. Scalarization reduces total register requirements by 20% with a warp size of 4, and up to 33% with a warp size of 32, as wider warp sizes amortize more redundancy from scalarized operations.

Figure 6 shows how scalarization affects dynamic instruction, register, and memory activity counts. In each graph, the left bar in each benchmark group is the baseline without scalarization, while the remaining bars show warp widths of 4, 8, 16, and 32 threads. We differentiate between the number of issued instructions (Figure 6(a)) and executed thread operations (Figure 6(b)), and each of these counts are broken down by source: scalars, statically converged warps, warps converged through dynamic convergence preservation, or diverged threads. Note that only one instruction is "issued" for all the threads in a warp when it is converged. "Operations executed" counts the total number of individual thread operations, regardless of convergence. Scalarization is subject to the effectiveness of convergence analysis (Figure 4), which is why benchmarks towards the left of Figure 6 have more scalar and statically converged warp instructions, and the benchmarks towards the right have more dynamically converged warp instructions.

In general, wider warp sizes decrease the instruction and operation counts because an instruction only issues once for all threads in a converged warp, and because scalar operations are only issued and executed once per warp. However, the number of diverged thread instructions increases with wider warps mainly because the likelihood of divergence increases with larger groupings of threads. This effect is apparent for `nw`, `hotspot`, `lbm`, and `stencil`. Still, since each converged warp instruction represents 4–32× more operations than a diverged thread instruction (following the warp size), the total operation count is always lower with scalarization than without. The savings range from 23–29% depending on warp size.

As expected, the other statistics of register read counts, register write counts, memory address lookups, and memory data accesses (Figures 6(c)-(f)) have roughly the same shape as the executed thread operations. With a warp size of 4, register reads and writes are reduced by 24%, memory address

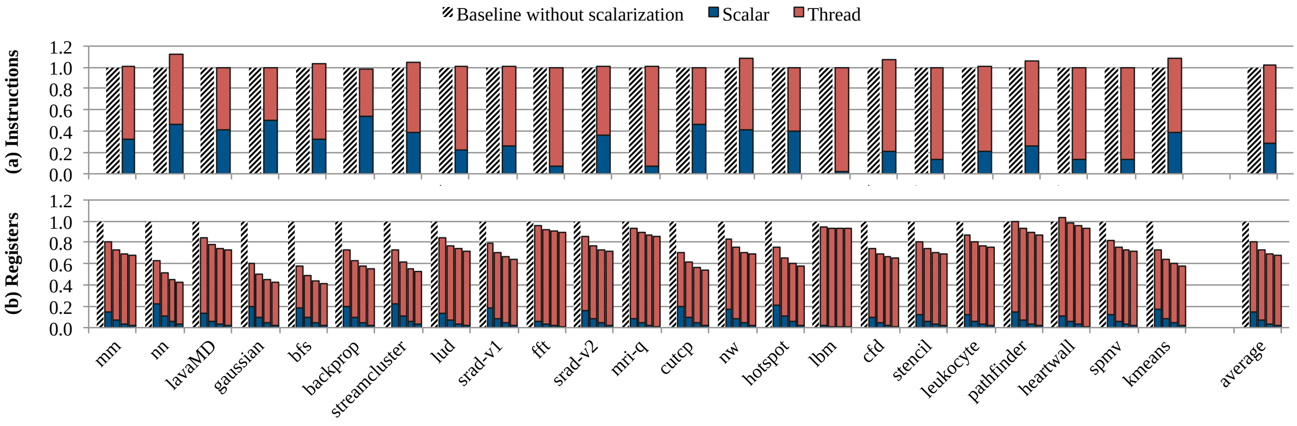


Figure 5. Static Scalarization Metrics – Each group shows results for warp sizes of 4, 8, 16, and 32.

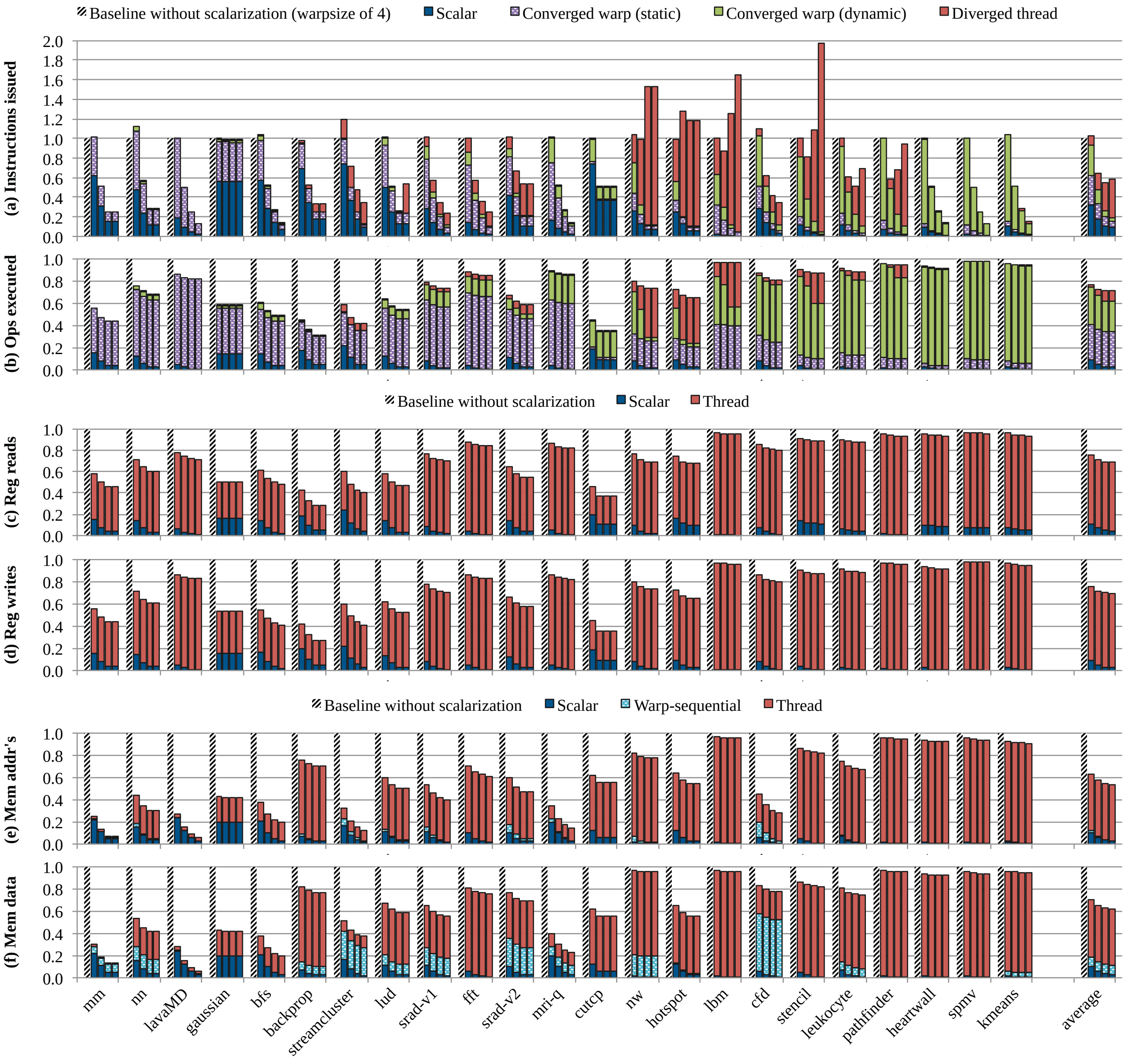


Figure 6. Dynamic Scalarization Metrics – Each group shows results for warp sizes of 4, 8, 16, and 32.

counts going to the innermost cache are reduced by 37%, and the number of memory data elements accessed is reduced by 30%. With wider warp sizes, scalar register accesses and scalar memory operations are more effectively amortized, and warp-sequential memory address activity similarly decreases (though the data counts stay constant). With a warp size of 32, register reads and writes are reduced by 31%, memory address counts are reduced by 47%, and data access counts go down by 38%.

5. Architecture Implications of Scalarization

Compile-time convergence analysis and scalarization can improve efficiency and performance in various contexts. The algorithms in our paper apply to existing processors, and they may also enable new hardware microarchitectures.

5.1 Scalarization in SIMT Microarchitectures

Figure 7 shows a range of SIMT microarchitectures extended with scalarization support. The same instruction-set architecture that we target in Section 3 applies to all of these microarchitectures, as do the microarchitecture-neutral results in Section 4. Figure 7(a) shows a traditional SIMT microarchitecture extended with a scalar unit on the left. A warp is mapped across the SIMT lanes with one thread per lane. In contrast to the wide SIMT unit, the scalar unit has a 1-wide datapath with a scalar register file and resources to execute scalar instructions. Scalarization reduces overall register file capacity by eliminating redundant operand storage, or alternatively allows a register file of a given size to map more threads. Scalar instructions improve performance as they allow regular SIMT instructions to execute in parallel, and they reduce energy by eliminating replicated work. Figure 7(b) shows an alternative microarchitecture which executes scalar instructions on a single lane instead of a separate unit, thus avoiding the area overhead when scalarization is not used. This architecture still reduces energy by only activating one lane when executing scalar instructions, but it would not reduce register pressure or improve performance.

Scalarization may also help enable new microarchitectures with better divergent-thread performance. In spatial-SIMT GPUs, divergence can be a performance bottleneck since throughput and efficiency are halved each time the threads in a warp diverge [2]. With complete divergence, only one of the warp's threads executes instructions at a time. A potential solution is a *temporal-SIMT* microarchitecture, as shown in Figure 7(c). In temporal-SIMT lanes fetch and execute instructions independently. A warp is mapped to a single lane, and the threads in a converged warp dispatch an instruction one after the other over successive cycles. In this way the temporal-SIMT lane amortizes instruction overheads similar to a 1-lane vector machine [25]. When threads are diverged, on the other hand, instructions simply dispatch for a single cycle and the independent lanes essentially operate as a traditional multithreaded MIMD processor.

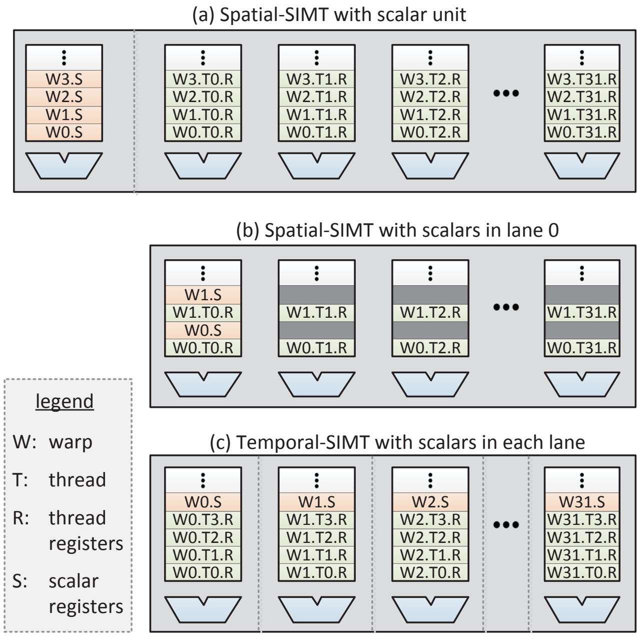


Figure 7. SIMT microarchitectures with scalarization support

Temporal-SIMT is a particularly good match for efficient scalarization. As shown in Figure 7(c), the register file mappings are configured to allocate a set of scalar registers for each warp. No physical partitioning of the register file is necessary (as in Figure 7(a)), and no register file slots are wasted (as in Figure 7(b)). Regular thread instructions can directly source operands from scalar registers instead of their usual private registers. The register file is only read once as the first thread dispatches, and the scalar operand is then held in a pipeline register as other threads dispatch. When a scalar instruction executes on a temporal-SIMT architecture, it simply dispatches once for the warp instead once per thread. Note that there will be no additional ports added to the register file or the instruction cache if the microarchitecture continues to issue one instruction per cycle. Scalarization on a temporal-SIMT architecture will improve both performance and energy, while requiring only minimal hardware modification and no separate scalar execution resources.

5.2 Stackless SIMT

The divergence stacks used in current GPUs (Section 2.4) have several drawbacks. First, as mentioned above, execution efficiency drops each time the threads in a warp diverge. The warp itself executes all of its composite thread code paths serially, so no parallelism is possible between diverged threads in the same warp. Secondly, in current SIMT compilers, achieving correct execution in the presence of unstructured control flow is a major challenge [29]. Finally, the divergence stack creates a pitfall where the SPMD model can break down: since the "threads" in a warp do not truly execute asynchronously. Threads can only synchronize at warp (or CTA) granularity, and threads in the same warp are only able to communicate when their execution is convergent [20].

These drawbacks of SIMT could all be addressed by mapping SPMD kernels to a future *stackless temporal-SIMT* architecture. Similar to MIMD, such an architecture would provision a hardware PC per thread and allow diverged threads to truly execute independently. Compiler convergence analysis is an important enabler for this form of stackless SIMT since, without a stack, hardware has no way to reconverge diverged warps. To enable *compiler-managed reconvergence*, the architecture can provide a `syncwarp` instruction that acts as a barrier for currently executing threads. The compiler simply places a `syncwarp` in blocks it identifies as convergent, and as long as all threads make progress, the warp will eventually reconverge at this instruction. However, placing a `syncwarp` in every convergent block incurs an overhead that is often not necessary. We can place these synchronization operations only at those convergent blocks that have at least one divergent predecessor, thus ensuring that scalar registers are only written in convergent blocks guarded by `syncwarp` instructions. In the example shown in Figure 2(a), node $N5$ must be guarded by a `syncwarp`.

6. Related Work

Our convergence analysis is based on the variance analysis described by Stratton et al. [26]. Their work identifies data accesses that are thread-invariant or will give the same value across the threads of a CTA. Their basic variance analysis was used to optimize CUDA programs when compiled to multicore CPUs. We extend their basic variance analysis algorithm to track not just thread-variant data, but also control divergence. We make optimistic assumptions about convergence and thread-invariant data and then track thread-variant information and divergent information together.

Coutinho et al. [7] describe what they call "divergence analysis," which is also an extension of the approach of [26]. Their analysis finds divergent values, by first converting SSA information into gated single assignment [22], and then replacing control-flow merges with a predicate select operator. Their end algorithm is relatively simple because it can use data-flow analysis to propagate divergence information, but it requires a change of representation. However their divergent values are similar to thread-variant values as described in [26].

Collange [5] presents work with goals similar to ours, but uses an approach like that described in [7]. Collange does not use a gated representation but instead performs a symbolic analysis on a lattice of tags, which encodes and tracks alignment of various instruction operands. Coutinho et al. [7] and Collange [5] do not perform convergence analysis, which is important for exploiting scalar code generation.

Karrenberg and Hack [13] describe an analysis based on a data-flow lattice approach which is similar to our affine analysis. However, their analysis is geared towards vectorization, rather than scalarization. Also, their analysis does not use control dependence information, which is useful in our case to perform convergence analysis.

The ISPC language includes explicit uniform data types that allow a program to indicate scalar values in source code [23]. While this approach may be well matched to tightly coupled SIMD architectures, our approach relieves the programmer from this burden and uses the compiler to discover uniform values that a programmer may not be able to specify. Furthermore ISPC does not have any explicit notion of convergence.

Kerr et al. [14] implement a thread-invariant expression elimination pass, also based on [26]. The focus of their optimization pass is different than ours; they use common subexpression elimination on invariants after vectorization, whereas we allocate invariants to scalar register.

Lee et al. [16] explore a range of vector, vector-thread, and SIMT architectures, comparing area and energy efficiency on regular and irregular codes. While the architectures they explore overlap with some of the scalar-SIMT architectures we describe in Section 5, their work does not examine efficiency optimizations enabled by convergence analysis.

Collange proposes a stackless SIMT architecture which implements reconvergence in hardware by comparing thread PCs every cycle [6]. This approach is different and less efficient than ours in which the compiler uses a `syncwarp` instruction to reconverge diverged warps. Intel's Sandy Bridge GPU [12] maintains a PC per thread, but threads do not truly execute independently. Instead, the compiler must sequence through all code paths, and PC comparators are used to mask inactive threads.

7. Conclusions

This paper presented new compiler algorithms for thread convergence and variable variance analysis that elides redundant instructions and register accesses in threaded code through a technique called scalarization. Our compiler algorithms are extremely effective at identifying convergent execution even in programs with arbitrary control flow, identifying two thirds of the instructions captured by a dynamic oracle. Simple hardware mechanisms can boost this to 100%. The compile-time analysis leads to a reduction in operations executed and register accesses of 23–31% depending on warp size.

Compiler convergence analysis and scalarization may enable alternative hardware architectures such as stackless temporal-SIMT. We anticipate additional optimizations, such as scalarizing across subsets of warps, will provide even greater benefits. We plan to further quantify the benefits of scalarization on various microarchitectures, including ones that dynamically scalarize instructions and operands without compiler guidance.

Acknowledgments

We thank members of the NVIDIA compiler team including Xiangyun Kong and Gautam Chakrabarti for various improvements to the variance analysis algorithm, and Manjunath Kudlur and Jaydeep Marathe for valuable discussions on the convergence analysis algorithm. We also thank members of the NVIDIA research group for their feedback on this work, and in particular Mojtaba Mehrara and Greg Diamos for their contributions to concepts and infrastructure. This research was funded in part by DARPA contracts HR0011-10-9-0008 and HR0011-11-C-0100, and an NVIDIA graduate fellowship.

References

[1] AMD. AMD Graphics Core Next (GCN). AMD Fusion Developer Summit, June 2011. URL `http://developer.amd.com/afds/assets/presentations/2620_final.pdf`.

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[3] T. Ball. What's in a Region?: or Computing Control Dependence Regions in Near-linear Time for Reducible Control Flow. *ACM Letters on Programming Languages and Systems*, 2(1-4), March 1993.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Characterization*, pages 44 –54, October 2009.

[5] S. Collange. Identifying Scalar Behavior in CUDA Kernels. Technical Report hal-00555134, Université de Lyon, January 2011.

[6] S. Collange. Stack-less SIMT Reconvergence at Low Cost. Technical Report hal-00622654, Université de Lyon, September 2011.

[7] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira. Divergence Analysis and Optimizations. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 320–329, October 2011.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, October 1991.

[9] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, September 2010.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[11] Intel. A First Look at the Larrabee New Instructions (LRBni). Intel White Paper, 2009.

[12] Intel. Intel HD Graphics OpenSource PRM Volume 4 Part 2: Subsystem and Cores. Intel Programmer's Reference Manual, February 2010.

[13] R. Karrenberg and S. Hack. Whole-function Vectorization. In *International Symposium on Code Generation and Optimization*, pages 141–150, April 2011.

[14] A. Kerr, G. Diamos, and S. Yalamanchili. Dynamic Compilation of Data-parallel Kernels for Vector Processors. In *International Symposium on Code Generation and Optimization*, pages 23–32, April 2012.

[15] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *International Symposium on Code Generation and Optimization*, pages 75–88, March 2004.

[16] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators. In *International Symposium on Computer Architecture*, pages 129–140, June 2011.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March/April 2008.

[18] D. Maier. The Complexity of Some Problems on Subsequences and Supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, March/April 2008.

[20] NVIDIA. NVIDIA CUDA C Programming Guide 4.2, April 2012.

[21] OpenCL. The OpenCL Specification Version 1.2. Khronos OpenCL Working Group, 2011.

[22] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The Program Dependence Web: a Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.

[23] M. Pharr and W. R. Mark. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing (InPar)*, May 2012.

[24] J. H. Reif and H. R. Lewis. Efficient Symbolic Analysis of Programs. *Journal of Computer and System Sciences*, 32(3):280–314, June 1986.

[25] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[26] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. mei W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *International Symposium on Code Generation and Optimization*, pages 111–119, April 2010.

[27] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois, Urbana-Champaign, March 2012.

[28] M. Weiss. The Transitive Closure of Control Dependence: the Iterated Join. *ACM Letters on Programming Languages and Systems*, 1:178–190, June 1992.

[29] H. Wu, G. Diamos, S. Li, and S. Yalamanchili. Characterization and Transformation of Unstructured Control Flow in Bulk Synchronous GPU Applications. *International Journal of High Performance Computing Applications*, 26(2):170–185, May 2012.