Dynamic Compilation of Data-Parallel Kernels for Vector Processors

Andrew Kerr[1], Gregory Diamos[2], S. Yalamanchili[3]
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
{arkerr[1], gregory.diamos[2], sudha[3]}@gatech.edu

ABSTRACT

Modern processors enjoy augmented throughput and power efficiency through specialized functional units leveraged via instruction set extensions. These functional units accelerate performance for specific types of operations but must be programmed explicitly. Moreover, applications targeting these specialized units will not take advantage of future ISA extensions and tend not to be portable across multiple ISAs. As architecture designers increasingly rely on heterogeneity for performance improvements, the challenges of leveraging specialized functional units will only become more critical. In particular, exploiting software parallelism without sacrificing portability across the spectrum of commodity and multi-core SIMD processors remains elusive.

This work applies dynamic compilation to explicitly data-parallel kernels and describes a set of program transformations that efficiently compile bulk-synchronous scalar kernels for SIMD functional units while tolerating control-flow divergence. It is agnostic to specific features of ISAs, and performance scalability is expected from 2-wide to arbitrary-width vector units. This technique is evaluated with existing workloads originally targeting GPU computing. A microbenchmark written in CUDA achieving near peak throughput on a GPU achieves over 90% peak throughput on an Intel Sandybridge. Speedups for real-world applications running on on CPUs featuring SSE4 achieve up to 3.9x over current state of the art heterogeneous compilers for data-parallel workloads.

1. INTRODUCTION

As processor designers reach the limits of frequency and ILP scaling, they are increasingly turning toward heterogeneity of functional units within systems and indeed within processors as the means for performance and power-efficiency scaling. Modern CPUs are equipped with SIMD instruction sets such as Streaming SIMD Extensions (SSE) [1] and Advanced Vector Extensions (AVX) [2] for x86, PowerPC's AltiVec, and Cell's SPU. These provide higher throughput than scalar pipelines but present several programming and engineering challenges to using them effectively. Specifically, programmers must re-write critical procedures using platform-specific parallel instructions, and the resulting programs are not portable across architectures.

We have concurrently seen the emergence of massively data parallel execution models and languages such as OpenCL [3] and NVIDIA's CUDA [4] that exploit thread-level parallelism in which programs are composed of *kernels*, whose execution spawns hundreds or thousands of scalar threads each executing the same kernel code. Such explicitly parallel models of computation targeting programmable graphics units (GPUs) from NVIDIA and AMD have been shown to be an effective approach for high performance computing. Recent work [5–7] has shown that GPU execution models are productive vehicles for targeting both GPUs and multicore CPUs, offering tremendous gains in portability and productivity without sacrificing performance. A dynamic compilation environment using this programming model is therefore capable of leveraging GPU and CPU architectures simultaneously. Such portability presents a compelling argument for rewriting compute-intensive software in a data-parallel manner without coupling the implementation to a particular class of processor. However, utilizing SIMD functional units remainds a challenge.

In this paper, we propose and evaluate techniques for the compilation of explicitly data-parallel kernels to SIMD vector units such as SSE or AVX while tolerating control-flow divergence. In particular, we propose a dynamic compilation approach for the optimized use of vector units integrated within the datapaths of a multicore CPU. The capability extends the portability of data parallel applications written in CUDA and OpenCL to high performance vector units. The principle challenge in mapping data parallel kernels to vector processors is in handling control flow. Static compilation approaches to thread serialization fail when threads execute divergent paths, and some technique is needed to tolerate this divergence. Methods relying on predication require hardware support and suffer from low utilization in heavily divergent kernel regions or regions with unstructured control flow [8].

This work focuses on a dynamic compilation approach to utilizing SIMD functional units when serializing light-weight thread executions. Specifically, a dynamic compiler and translation cache partitions data-parallel kernels into schedulable regions and dynamically specializes these regions based on vector widths determined as the kernel is executing. This work proposes *yield on diverge* in which control-flow divergence is detected at runtime and handled as a context switch storing live state to thread-local memory, exiting the vectorized kernel and rentering a dynamic execution manager. The execution manager forms an warp from ready threads waiting to resume execution at the same location, and continues execution. This work demonstrates a proposed dynamic compilation model that scales to thousands of light-weight threads executing on modern commodity multi-core CPUs featuring vector functional units.

This work proposes the following contributions:

- Vectorization of data parallel kernels via a program transformation for the static interleaving of a set of data-parallel scalar threads targeting vector functional units

- Yield on diverge, a software-only technique for suspending thread execution and reforming warps to minimize the effect of branch divergence
- A dynamic compilation model from a low-level data parallel intermediate representation (IR) that makes use of the preceding techniques that scales to thousands of light-weight threads
- An evaluation of the implementation on a modern multi-core processor with multiple vector units and over 40 existing data-parallel workloads
- Implementation and evaluation of optimizations to remove redundant thread-invariant instructions exposed by vectorization

The rest of this paper is organized as follows. In Section 2, we describe the PTX execution model targeted by CUDA and OpenCL. In Section 4, we describe *vectorization*, a novel program transformation that interleaves static instructions and promotes them to vector types where possible. We also describe implications for control-flow divergence and present *yield on diverge*, a novel method for handling it. In Section 3, we present an execution manager for coalescing logical PTX threads for executing vectorized kernel regions. We present results of a thorough evaluation in Section 6 and conclude by describing lessons learned in transforming data-parallel execution models targeting dynamic compilation frameworks.

2. DATA-PARALLEL EXECUTION MODELS

Parallel Thread eXecution, or PTX [9], is the RISC-like virtual instruction set targeted by NVIDIA's CUDA and OpenCL compilers and used as an intermediate representation for GPU kernels. PTX's Single-Instruction, Multiple-Thread (SIMT) execution model is defined for a hierarchical collection of scalar threads executing concurrently. Rather than launch one thread per processor, data-parallel kernels are written for thousands of light-weight threads that collectively perform a computation, each typically producing a small number of elements in output data sets.

Collections of scalar threads or work items are partitioned into cooperative thread arrays (CTAs), analogous to work groups from OpenCL, that are allowed to synchronize via barriers and exchange data. Sets of CTAs are nondeterministically dispatched to available processor cores which execute them to completion. Global barriers across all cores and CTAs are only permissible at kernel boundaries enabling the platform to implement a weakly consistent global memory model as well as avoid overheads of synchronization and cache coherence. Consequently, the programming model enables scalability across large ranges of concurrency in the underlying processor(s). The relationship between kernel grids and CTAs is illustrated in Figure 1. This work targets the PTX execution model in particular but applies to any bulk synchronous execution model in which multiple logical threads are executed in lock step on a SIMD processor.

The PTX instruction set consists of standard arithmetic instructions for integer and floating-point arithmetic, load and store instructions to explicitly denoted address spaces, texture sampling and graphics related instructions, and branch instructions. Additionally, special instructions for interacting with other threads within the CTA are provided such as CTA-wide barrier instructions, warp-wide vote instructions, and reduction operations, to name several. PTX is decoupled from actual hardware instantiations and relies on a just-in-time compilation toolchain to target native instruction set architectures.

In the context of the PTX execution model, this work is concerned with mapping a collection of scalar *threads* from one or more *cooperative thread arrays* onto one or more vector functional units. A vector register file serves as the source and destination of vector operations. Vector load and store instructions operate between memory and this register file. Vector functional units are organized into multiple lanes where each lane implements a single arithmetic or logical operation. A single vector instruction controls all lanes. The width of vector unit will be referred to as the *warp size* for compatiblity with GPU computing terminology. Unlike SIMD processors from NVIDIA and AMD, vector units/lanes cannot implement arbitrary control flow which both simplifies the implementation and limits their applicability. In particular vector units are distinguished by the following.

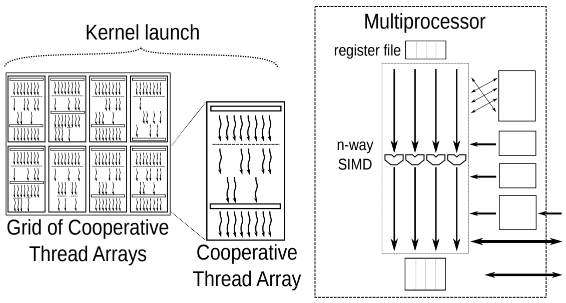


Figure 1: PTX [9] execution model. Kernels are scalar functions launched over a hierarchical collection of threads exhibiting coarse-grain and fine-grain synchronization semantics. The processor model consists of a replicated and unsynchronzied set of multiprocessors, each with a wide SIMD functional unit, multiple on-chip address spaces, and access to off-chip weakly consistent global memory.

- vector loads and stores fetch continguous sequences (vectors) of scalar data
- vector operators are applied within a lane; lanes may not be arbitrarily masked
- conditional select operators may choose between two values in each lane

This machine model is implemented in numerous commodity processors presently available including Advanced Vector Extensions (AVX), Streaming SIMD Extensions (SSE), AltiVec, and ARM Neon. Moreover, the current trend is increasing vector widths from 4 in the case of SSE to 8 in the case of the recently available AVX. Proposed processor architectures such as Intel's Knights Ferry [10] suggest this trend will continue with increasing vector widths.

3. DYNAMIC COMPILATION MODEL

The proposed compilation model is wrapped by an API front-end for heterogeneous computing. This implementation supports the CUDA Runtime API. PTX modules are explicitly registered with the runtime which immediately parses and analyzes kernels within the modules. These are added to a global translation cache which lazily translates PTX kernels to LLVM and then vectorizes these translations for several warp sizes presented by the target machine model. Kernel launches, illustrated in Figure 2, spawn a set of hardware threads, each running a dynamic execution manager. The kernel's grid of CTAs is statically partitioned across the set of execution managers which concurrently serialize the execution of light-weight threads within the CTAs while respecting the semantics of the execution model. Execution managers form warps, or collections of PTX threads, waiting to execute the same block within the thread. The number of

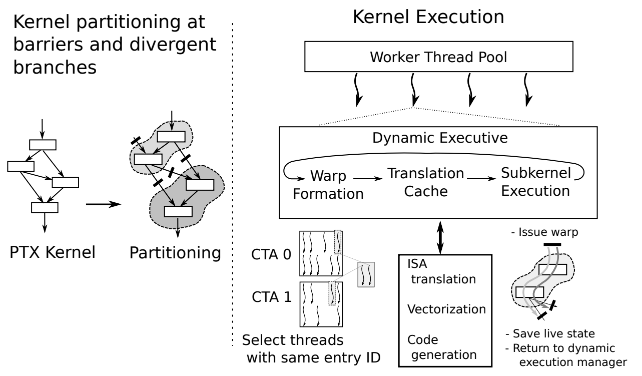
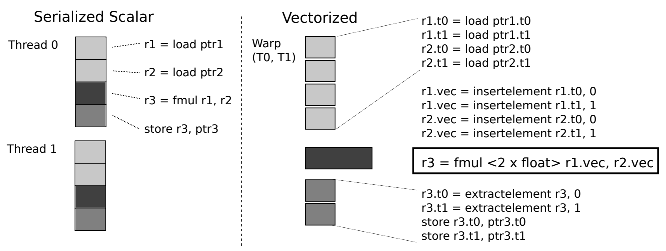


Figure 2: Dynamic compiler and execution manager framework for data-parallel kernels supporting vectorization.



```
r1 = load ptr1
r2 = load ptr2
r3 = fmul r1, r2
store r3, ptr3
```

```
r1.t0 = load ptr1.t0
r1.t1 = load ptr1.t1
r2.t0 = load ptr2.t0
r2.t1 = load ptr2.t1

r1.vec = insertelement r1.t0, 0
r1.vec = insertelement r1.t1, 1
r2.vec = insertelement r2.t0, 0
r2.vec = insertelement r2.t1, 1

r3 = fmul <2 x float> r1.vec, r2.vec

r3.t0 = extractelement r3, 0
r3.t1 = extractelement r3, 1
store r3.t0, ptr3.t0
store r3.t1, ptr3.t1
```

Figure 3: Serializing scalar threads executing the same basic block by interleaving static instructions and promoting arithmetic instructions to vector operators.

threads within the warp is used to query the global translation cache and obtain a native ISA binary. When threads reach a CTA-wide barrier or diverge, they yield via statically defined kernel exit points and control returns to the execution manager. This process iterates until all threads have terminated, and all worker threads reach a kernel-wide barrier at which point the kernel is finished.

4. VECTORIZING SCALAR KERNELS

This work proposes *vectorization*, a program transformation mapping a kernel of data-parallel scalar threads onto a vector processor. This transformation produces a specialized form of a kernel by *replicating* scalar instructions and, where supported by the target ISA, *promoting* replicated instruction sets to vector operators. Execution of a single vectorized kernel is computationally equivalent to the serial execution of a scalar version of the kernel over a collection of threads where each thread is mapped to a lane within the vector functional unit, and the width of each vector operator is equivalent to the number of threads covered by this kernel's execution. Figure 3 illustrates the transformation of a scalar instruction sequence into a vectorized form. Scalar load and store instructions are replicated, and the binary operator (floating-point multiply, in this case) is promoted to an element-wise vector operation. In the scalar kernel, two iterations would be required to execute this kernel over two threads. The vectorized kernel requires a single iteration for an equivalent execution and exhibits higher instruction-level parallelism.

Vectorization may be implemented with Algorithm 1 whose input is a scalar kernel. Thread-local and CTA-local data members are accessed via a context object identifying the executing thread. This context object includes grid dimensions, block dimensions, block ID, thread ID, and base pointers to the following address spaces: parameter, shared, and thread-local. The output is a vectorized kernel in which a single

```
Input: Instruction $i$
Input: warp size $ws$
Output: Vectorized instruction
replicate $i$ for each of $ws$ threads
foreach replicated instruction do
    update thread ID operands
if $i$ is vectorizable then
    replace $ws$ instructions with single vector-typed instruction
memoize resulting instruction or bundle
```

Algorithm 1: `Vectorize`(i, w) replaces a scalar instruction a replicates set of instructions, one for each thread in the warp. This set may be promoted to a single vector instruction.

execution of each basic block is equivalent to executing that block by all of the threads in a warp. The input to the resultant vectorized kernel is an array of context objects, each describing a unique thread. This basic transformation does not consider divergence which is addressed by a subsequent transformation described in Section 4.1.

The vectorization pass is implemented as a transformation that replicates instructions while maintaining a mapping from scalar source instructions to the replicated set. Thread-local values such as pointers to local memory and thread indices are loaded from a thread context object. Vectorized kernels receive an array of context objects constituting the warp, and accesses to context objects in vectorized kernels are modified to index the correct thread's context. Following replication, vectorizable instruction bundles are replaced by a single vector instruction with vectorized operands. Either the replicated instruction bundle or the vectorized instruction are memoized into the mapping. To vectorize the operands, they are either selected from the mapping, or they are recursively vectorized. The order in which instructions are vectorized affects recursion depth as well as performance of the compilation pass due to locality of the instruction objects and tabel lookups. The method adopted here is a breadth-first traversal of basic blocks composed with a linear scan of instructions within each basic block. This work vectorizes binary floating-point and integer operators as well as calls to transcendental functions for which both LLVM and the compilation target, the x86-64 ISA with AVX, have built-in support

Non-vectorizable Instructions. CPU instruction set architectures do not typically support vector forms of all instructions. Loads and stores, for instance, do not support scatter and gather with vectors of pointers. Rather, many ISAs such as SSE and AVX enforce loading of contiguous data from a single base address. Significant performance may be lost if this value is not aligned to super-word boundaries. This approach groups loads and stores in a class of instructions which may be not vectorized and are instead replicated for each thread. The values produced are explicitly packed into vectors when a non-vectorizable instruction produces a value used by a vectorized instruction, and explicitly unpacked when a non-vectorizable instruction uses a vectorized operand. A subsequent dead-code elimination pass removes unused instructions. Conservative handling of load and store instructions as non-vectorizable enables this technique to accommodate mis-aligned accesses and accesses to random locations, two cases that would not perform well or are not supported by SSE.

Explicitly repacking scalar values into vectors presents some overhead, though the extent of additional data movement instructions emitted depends on the actual kernel being compiled and on the quality of the backend code generator. In the particular case of memory instructions, we envision divergence analysis [11] and affine analysis [12] to identify opportunities in which multiple threads are guaranteed to access contiguous data. In these instances, arbitrary loads may be replaced with vector loads. An evaluation with this optimization remains for future work.

Sequences of interleaved replicated instructions exhibit instruction level parallelism that is at least as high as warp size. This comes at the cost of increasing the live ranges of values which places pressure on register usage. Moreover, transformations within LLVM's code generator attempt to subvert explicit instruction interleavings in order to reduce live ranges while discarding ILP. This required modifying LLVM to select an existing code generator that maintains the instruction schedules of source LLVM modules.

Implicit Synchronization. Guo, et al. [13] identiy idioms related to implicit synchronization among the threads in a warp when executing on SIMD processors. As an optimization, programmers rely on the hardware executing the threads of a warp in lock-step and omit barriers when threads in the same warp exchange data through shared memory. Omitting barrier instructions saves several cycles by not issuing the instruction, however such programs are not portable across processors with different warp sizes (from AMD to NVIDIA GPUs, for example). Moreover, it is not always possible for a compiler to address implicit synchronization, as not all threads in the warp may reach the implicitly synchronized code. The compiler cannot not insert a warp-wide barrier without risking incorrect behavior in the case when some but not all threads reach the implied barrier. To the best of our knowledge the technique proposed in [13] is not capable of handling such a case. The work described in this paper assumes the programmer does not require warp synchronous execution and yields undefined behavior for such kernels.

4.1 Divergent Control Flow

The set of threads mapped to a vectorized kernel must necessarily take the same control paths. An execution of a kernel is *convergent* if all threads follow the same path; execution is *divergent* if threads evaluate conditional control-flow instructions differently. Some kernels may be statically proven to be entirely convergent, and presumably some kernels contain potentially divergent paths that are never taken by common datasets. Characterization studies [14] indicate most real-world CUDA programs experience some form of divergence which must be efficiently tolerated. Figure 4 (a) shows a sample control flow graph with two threads executing B0 and B1 then diverging at the the branch terminating B1. Thread 0 branches to B3 while thread 1 falls through to B2. A single execution of a vectorized basic block is equivalent to both threads executing the scalar form, therefore some mechanism must be present to avoid executing B2 for thread 0.

This work proposes *yield on diverge*, a software-only approach which checks branch conditions at runtime. Figure 4 (b) illustrates the execution of a kernel with divergence control flow. An execution manager collects a set of ready threads waiting to execute the same basic block. The execution manager then selects a vectorized kernel whose warp size is equal to the size of the collection of threads, and control enters the vectorized kernel. A scheduler block performs an indirect branch based on the identity of the actual entry point, and control resumes execution within vectorized basic blocks.

Conditional branches are modified with additional instructions to detect divergent branches. On divergence, threads yield to an execution manager which insert threads into a *ready* queue and reform a new warp. Execution of a vectorized block is logically equivalent to each thread within the warp executing that block, and consequently warps may only be formed of threads waiting to enter the same block. Yields to the execution manager are analogous to a context switch. Barrier synchronizations are handled like divergent branches except threads are inserted into a *waiting* queue within the execution manager.

Algorithm 2 describes how vectorized kernels are transformed to accommodate control-flow divergence. This applies the `Vectorize`(i, ws) function described in Algorithm 1 to vectorized instructions. Conditional branches terminating basic blocks are transformed by summing the branch predicates from each thread. If the sum is zero, all threads jump to the fall-through target (the branch was uniformly not taken). If the sum is equal to warp size, all threads jump to the branch target (uniformly taken). Otherwise, control enters an exit handler which performs the divergent yield. Successors to divergent branches are inserted into a list of possible entry points which are then used to construct a scheduler block at the beginning of the kernel.

Transitions from the execution manager to the kernel are accomplished via a compiler-inserted scheduler block which acts like a trampoline. A basic block inserted into the kernel contains a large switch statement conditioned on the warp's entry ID. These integer-valued IDs select basic blocks that are the successors to divergent branches (or barrier synchronizations) identified in Algorithm 2. For each entry point, an entry handler block is inserted to restore the warp's live state from local memory. Its terminator instruction jumps to the vectorized entry block. Algorithm 3 describes how a scheduler block is constructed.

```
Input: Warp size $ws$
Input: Scalar kernel to be vectorized
Output: Vectorized function supporting control-flow
begin
    $entrySet$ := {} $exitSet$ := {}
    foreach basic block b in kernel do
        foreach non-control instruction i in b do
            Vectorize($i$, $ws$)
        if b ends in conditional branch then
            insert empty basic block exit$_b$ to function
            insert instruction: sum( predicates )
            replace the conditional branch with:
            switch sum( predicates ) do
                case 0
                    jump to fall-through successor
                case $ws$
                    jump to branch successor
                otherwise
                    jump to exit$_b$
            add exit$_b$ to {$exitSet$}
            add $successors(b)$ to {$entrySet$}
    CreateScheduler({$entrySet$})
    CreateExits({$exitSet$})
end
```

Algorithm 2: Inserts detection and handling code into kernel.

```
Input: {$entrySet$}
create empty basic block scheduler
insert switch statement into scheduler with default target of
entry block to function
foreach b in entrySet do
    create empty basic block entry$_b$
    insert load instructions into entry$_b$ for all live-in values at
    block b
    insert jump to block b
    add to switch statement in scheduler:
    case (b)
        jump to entry$_b$
```

Algorithm 3: `CreateScheduler`({$entrySet$}) creates a scheduler block and inserts code to restore live state.

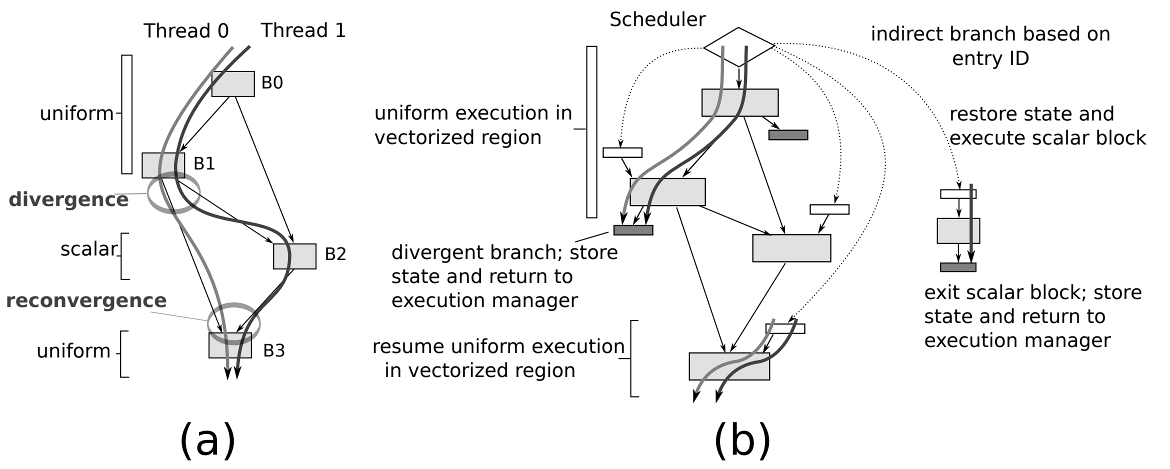

Thread 0
Thread 1
B0
uniform
B1
divergence
scalar
B2
reconvergence
uniform
B3
(a)
Scheduler
indirect branch based on entry ID
uniform execution in vectorized region
restore state and execute scalar block
divergent branch; store state and return to execution manager
exit scalar block; store state and return to execution manager
resume uniform execution in vectorized region
(b)

Figure 4: (a) Control-flow graph executed by two threads diverge at B1 and reconverge at B3. (b) Executing a kernel with divergent control flow through a vectorized and a scalar specialization of the kernel.

```
Input: {$exitSet$}
create local variable $resumeEntryId$
create local variable $resumeStatus$
foreach exit$_b$ in $exitSet$ do
    insert store instructions into exit$_b$ for all live-out values at
    block $b$
    insert $resumeEntryId \leftarrow$ select(predicate, { branchTarget,
    fallThrough })
    insert $resumeStatus \leftarrow$ { Thread_branch,
    Thread_barrier, Thread_exit }
    insert return
```

Algorithm 4: `CreateExits`({$exitSet$}) stores live-out state at divergence sites, inserts a conditional select operator to specify the target entry point, specifies a status indicating why the warp has returned to the execution manager, and exits.

Exit handling code inserted by Algorithm 4 into exit blocks performs yields to the execution manager. At yield points such as divergent branches and barriers, control passes from a vectorized block to an exit block. The exit block first spills all live values to thread-local memory for each thread. Then, a conditional select operator stores a constant-valued integer identifying the branch target block for each thread which is then written to that thread's resume point field. Divergent threads will evaluate this select instruction differently and write different entry IDs to their resume point fields. Finally, a value indicating the disposition of the kernel exit is written to the warp's resume status field. The execution manager, described in Section 5.2, updates its pool of ready thread contexts according to the resume status type and chooses a new warp by collecting threads with the same resume point.

This work considers three classes of kernel yields: divergent branches, CTA-wide barriers, and thread termination. When yielding on barriers, the execution manager places context objects in a wait queue to avoid rescheduling them until all threads in the CTA have reached the barrier. On termination, the context object is discarded. This work does not implement function calls, mainly due to their relatively new introduction to programming model on which this work was evaluated. These may be potential sources of divergence also, either during conditional call instructions or indirect calls when the target is non-uniform. The approach described here may be extended to function calls via the introduction of a thread-local call stack, replacing call targets with kernel entry IDs, and by always yielding on function calls. This remains for future work.

Figure 5 illustrates entry and exit handlers in greater detail. Block B1 has been vectorized for warp of size 2 and exists within the shaded region shown in the figure. Block B1_entry has been added to the kernel and provides a control path from an external scheduler into the vectorized region that loads live values from thread-local memory. A conditional branch instruction terminating B2 has been replaced with a switch statement whose conditional is the sum of all branch conditions within the warp. Its default successor is the exit handler, and two other successors are vectorized blocks within the kernel.

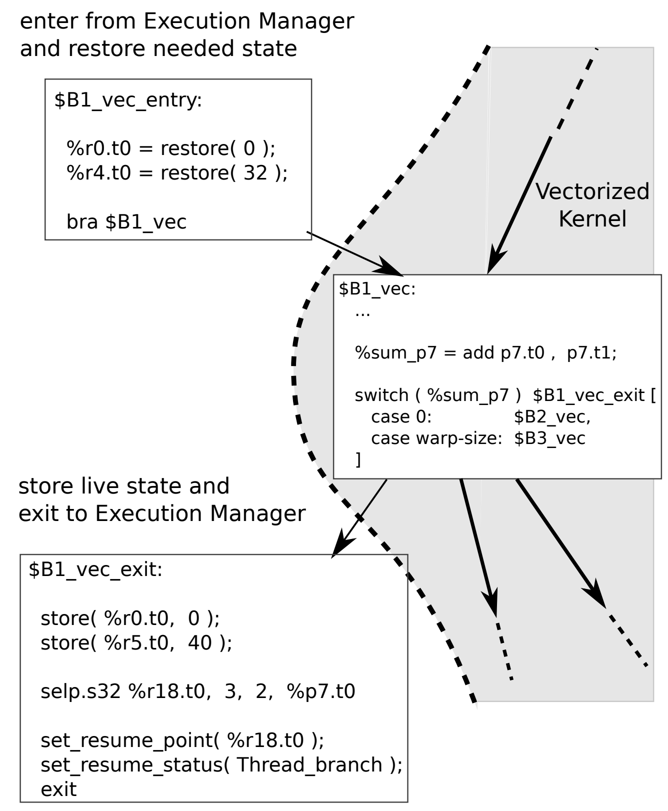
This technique requires a scalar specialization and a specialization for some maximum vector width. Additionally, implementations may produce specializations for narrower vector widths. The implementation for this work assumes each kernel has been specialized for warp sizes of 1 thread, 2 threads, and 4 threads corresponding to available vector processing hardware in the target processor. Entry and exit points have been added to divergence and reconvergence sites to restore live variables from thread local memory and enter the kernel. A scheduler block performs an indirect jump to the entry point selected by the warp's entry ID.

5. IMPLEMENTATION

The vectorization transformation described in the previous section was implemented as a device backend to GPU Ocelot [15], a dynamic compilation framework for GPU computing. GPU Ocelot translates PTX code to LLVM's IR and utilizes its extensive analysis, optimization and code generation facilities [16]. Our implementation described in this section extends GPU Ocelot's multicore CPU backend with the addition of a dynamic execution manager, dynamic translation cache, and the vectorization program transformation.

5.1 Dynamic Translation Cache

The translation cache is the module responsible for producing native ISA binaries of each kernel by translating from PTX to LLVM, applying program transformations, and JIT compiling to the native ISA of the target CPU. Exhibiting the external semantics of a code cache, it may be queried by execution managers running in the worker threads by specifying an entry point ID and warp size. Before translation to LLVM, a PTX to PTX transformation replaces non-branch predicated



```
$B1_vec_entry:

  %r0.t0 = restore( 0 );
  %r4.t0 = restore( 32 );

  bra $B1_vec
```

```
$B1_vec:
  ...

  %sum_p7 = add p7.t0 ,  p7.t1;

  switch ( %sum_p7 )  $B1_vec_exit [
    case 0:              $B2_vec,
    case warp-size:  $B3_vec
  ]
```

```
$B1_vec_exit:

  store( %r0.t0,  0 );
  store( %r5.t0,  40 );

  selp.s32 %r18.t0,  3,  2,  %p7.t0

  set_resume_point( %r18.t0 );
  set_resume_status( Thread_branch );
  exit
```

Figure 5: Divergent branch entry and exit handlers for a vectorized kernel. The conditional branch in the vectorized block B1_vec has been replaced by explicit checks. On divergence, threads yield by exiting via B1_vec_exit.

instructions with select and splits basic blocks at barriers. Entry and exit handlers are inserted with procedures to store and restore live values as well as update thread status and next entry points on kernel exits. The process of translating PTX to LLVM has been described in detail in [16]. This work leverages many of these techniques to translate scalar PTX kernels into LLVM representations but applies the unique approach to execution model transformations described in Section 4.

When kernels are launched, execution managers query the translation cache for particular warp sizes. These initiate translation from PTX to a scalar LLVM representation and subsequent vectorizing for the requested warp size. Potentially, the translation cache could be modified to support querying for additional specialization parameters beyond warp size such as optimization level or particular kernel argument values. This remains for future work. Following translation and vectorization, the translation cache applies existing LLVM transformation passes including traditional compiler optimizations such as basic block fusion and common subexpression elimination. Finally, LLVM's code generator performs JIT compilation to yield a native ISA form of the vectorized kernel which is inserted into the cache to future requests from execution managers.

5.2 Dynamic Execution Manager

Each worker thread instantiates an execution manager which orchestrates the execution of all PTX threads from this set of CTAs while respecting CTA-wide barrier semantics (Figure 2). The execution manager contains a data structure of thread context objects, manages per-CTA memory structures such as shared memory and a block of contiguous memory partitioned into per-thread local memory. It implements warp formation and a thread scheduler. Prior to each kernel entry, the execution manager may select any thread not waiting at a barrier for execution. The current algorithm selects a ready thread via a round-robbin scheduler then attempts to construct the largest warp possible from other ready threads with the same entry point. The execution manager then calls the kernel and passes the warp of thread contexts. When threads yield to the execution manager, the warp's resume status indicates whether thread context objects should be terminated, returned to the ready pool, or added to their parent CTA's barrier pool.

Execution managers block while contending for lock on the dynamic translation cache. Compilation which is performed in the parent worker thread of the querying execution manager, so multiple worker threads querying for the same unavailable translation would be stalled. A possible optimization to the execution manager might give scheduling priority to warps for which translations exist to avoid stalling while the dynamic translation cache is actively compiling a previously requested translation. At this time, we only perform translations on kernel granularities so the benefits of concurrent execution and translation are less apparent. This is the subject of ongoing work but is orthogonal to vectorization.

6. EXPERIMENTAL RESULTS

This section presents the results from an evaluation of the described extensions to Ocelot-2.0.1464 compiled with LLVM 3.0. Evaluations were conducted on a desktop workstation running Ubuntu 11.04 x86-64 and using over 40 benchmark applications chosen from the CUDA Software Development Kit and the Parboil Benchmark Suite [17]. The evaluation system contains an Intel Sandybridge (i7-2600) CPU. Sandybridge supports SSE 4.2 and AVX. The proposed techniques for targeting vector functional units are expected to utilize AVX, but current lack of support for AVX in LLVM's code generator made such an evaluation infeasible as of this time. Moreover, this work is expected to apply to future architectures such as Intel's Knight's Ferry [10] equipped with 16-lane vector units. However, lack of simulation tools and a backend code generator for this ISA prevent an evaluation on this platform at this time.

The first set of experiments investigates speedups for idealized cases showcasing the benefits of vectorization and thread fusion. The second set of experiments measure performance improvements for real-world applications and provide statistics about application behaviors recorded by the execution manager. These statistical behaviors justify some design decisions and provide insights into sources of speedup and future optimizations. The third set of experiments evaluates the effectiveness of several proposed optmziations enabled by this dynamic compilation framework. This set of evaluations is intended to capture the performance gains possible using a portable data-parallel kernel representation that runs on GPUs and CPUs and is not necessarily intended to be competative with hand-tuned kernel implementations.

Throughput. This microbenchmark attempts to achieve peak theoretical throughput of floating-point units by replicating a sequence of interleaved, independent instructions. As described by Volkov [18], pipeline latency may be hidden given a sufficiently large number of threads. Increasing threads results in increased pressure on the register file, but the benchmark's relatively small number of live values and non-overlapping ranges is easily to accommodated. Multicore CPUs are more heavily pipelined with issue latency of four cycles in the case of SandyBridge's SSE floating-point simple arithmetic unit [1].

Table 1 illustrates sustained floating-point throughput for increasing vector widths for a compute-bound kernel running on the test plat-

Table 1: Peak floating-point throughput.

Warp size	1	2	4	8
GFLOPs/s	25.0	47.9	97.1	37.0

form. Floating-point throughput is expressed in single-precision GFLOP/second on a machine whose peak floating-point throughput is estimated to be 108 GFLOP/s. The benchmark itself consists of back-to-back floating point multiply and adds within a heavily unrolled loop launched over 576 threads. Warps of 4 threads achieve 97.1 GFLOPs/s on the target machine, or 90% of peak. Scalar threads saturate the scalar FPU issue ports and achieve 25.0 GFLOPs/s. Exceeding the vector width of the target processor requires the code generator to emit multiple vector operators in series which increases register pressure and extends the live ranges of values. Consequently, executing the above benchmark with a warp size of 8 threads while targeting SSE results in degraded performance.

6.1 Performance Gains

Speedup. The principle benefit of vectorization is the efficient utilization of vector functional units for applications that exhibit divergent control flow. This set of evaluations captures runtimes of CUDA kernels from the CUDA 2.2 SDK and Parboil application suites for a maximum warp size equal to the machine vector width of 4 threads. Speedups relative to a baseline of scalar execution are presented in Figure 6. The baseline translator and thread scheduler is identical to what was presented in [16].

Average speedup is 1.45x. Speedup varies from approximately 1.0x in the case of applications such as *BoxFilter*, *ScalarProd*, and *SobolQRNG*. These applications have memory-bound kernels but perform frequent synchronizations such that threads maintain high locality even without vectorization. Other applications that are more compute bound with fewer synchronizations achieve higher speedups. *BinomialOptions* achieves 2.25x speedup over the baseline, and the Parboil application *cp* achieves 3.9x speedup. Both applications have very uniform control flow properties and unrolled loops. Other applications such as *MersenneTwister*, *mri-fhd*, and *mri-q* run slower with dynamic warp formation. We believe this is due to control-flow irregularity. Threads with uncorrelated control-flow properties may diverge at every branch unless maximum warp size is limited. This observation motivates future work to detect cases when diverging branches are so frequent that scalar execution is optimal.

Average Warp Size. Figure 7 illustrates the average warp size of each kernel for the applications executed in Figure 6. This metric expresses the fraction of kernels of warp sizes 1, 2, and 4, where 4 is the maximum warp size. The results indicate that most kernel entries from the execution manager have warp size of 4 for every application except *SimpleVoteIntrinsics* which is only ever to form warps of 2 theads at most. These results also show that many applications are not entirely convergent which justfies the design decision to tolerate divergence and use dynamic warp formation to maximize available warp size. Finally, convergence does not entirely capture performance properties. *BinomialOptions*, for instance, achieves among the highest speedups.. This indicates that a dynamic warp formation strategy is very effective in improving utilization, but an implementation may be penalized by frequent kernel exits to the execution manager.

Liveness at Entry Points. This metric counts the average number of values restored per thread at entry points from the execution manager taken during the execution of each program. Runtime overhead at transitions between the execution manager and the kernel is proportional to the number of values restored. On average, 4.54 values are live per thread at each entry. Figure 8 illustrates this for each of the benchmark applications. Most applications with barriers have live state at yield points and require some context to be reloaded. On average, fewer values than architectural registers registers need to be restored indicating compiler-inserted context save and restore points may be at least as efficient as other types of cooperative threading libraries.

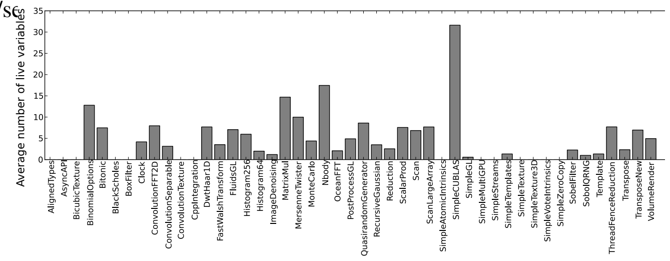


Figure 8: Average number of values loaded per thread on entry from the execution manager.

Yield Overheads. Figure 9 illustrates the fraction of CPU clock cycles spent in different phases of the execution of kernels in an application. For many applications, time spent in the Execution Manager (EM) is extensive. This includes testing barriers, inserting thread contexts into warps, and updating thread status after the execution of a kernel. Yield points that store and restore live state on transitions to the execution manager present a small overhead relative to cycles spent actually executing the subkernel. Applications such as *MersenneTwister*, *Nbody*, and *CP* achieve both high speedup, and nearly all execution time is spent within the vectorized subkernel. Synchronization-intensive applications such as *BinomialOptions* and *MatrixMul* spend more time within the execution manager and have limited speedup, even with little divergence. As vectorization reduces the execution time of the kernel, the relative percentage of time spent in the execution manager increases. These results suggest improving efficiency of the execution manager is key to further increases in performance even for applications with highly efficient compute-bound kernels.

6.2 Thread Invariant Expression Elimination

Scalar threads following the same paths through a kernel may compute identical results in some expressions. These expressions have data-dependencies to CTA-wide invariants such as kernel arguments, block and grid dimensions, and shared constants. For example, many CUDA kernels compute the expression *blockDim.x * gridIdx.x* such as when determining a thread's global index in a kernel grid. *Thread-invariant* expressions are redundant across a warp, and their elimination is expected to improve performance when threads are serialized. The approach to vectorization described in this paper enables classical compiler optimizations - common subexpression elimination - to identify thread-invariant expressions and eliminate them.

This experiment constrains warps to consist of consecutively indexed threads, a mapping defined *a priori* and termed *static warp formation*. Following vectorization, thread ID values are replaced with constant expressions relative to the warp's base thread. For example, *thread 0* loads its thread ID from a context object, but *thread 2* computes it from *thread 0*'s ID. Expressions in each thread, which are not true dependencies of thread ID, are subsequently marked as thread-invariant. Standard common subexpression elimination optimizations downstream of vectorization eliminates redundant thread-invariant expressions via a conservative analysis.

Collange *et al.* [12] show an average of 15 % of result PTX operands are reported as thread-invariant averaged over CUDA SDK applica-

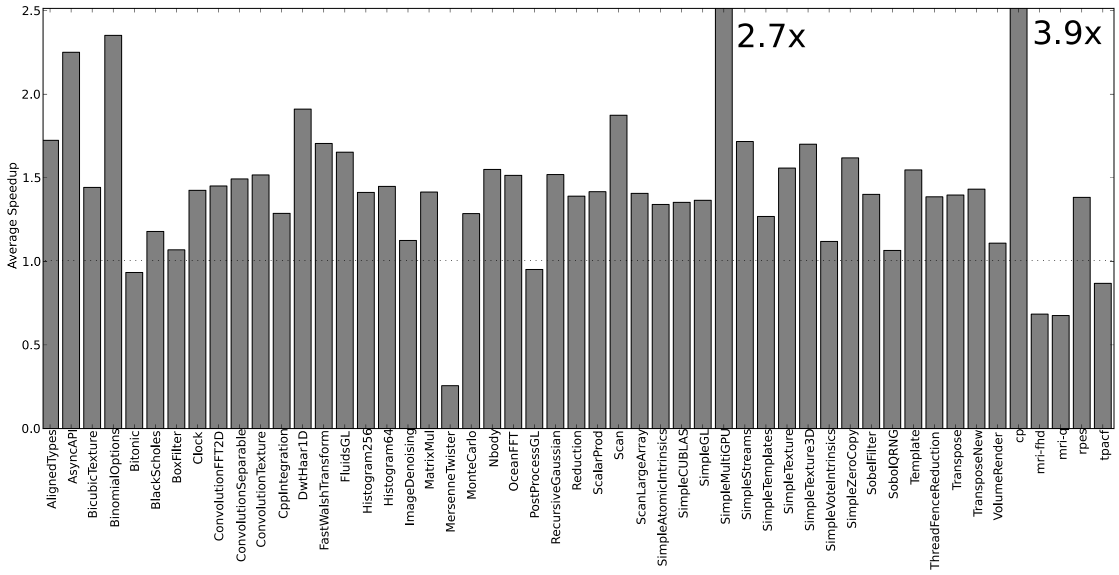


Figure 6: Speedup of benchmark applications.

tions. This work's approach to vectorization with static warp formation was able to reduce LLVM instruction counts by 9.5 % on average for a warp size of 2 threads. LLVM's optimization pass also reduces vector instructions to scalar instructions when lanes other than the base lane are redundant. For a warp size of 4 threads, 11.5% of instructions were eliminated. Larger warps imply a large fraction of thread-invariant instructions.

Speedup with Thread-Invariant Elimination. This experiment constrains warps to consist of consecutively indexed threads from the same CTA and applies thread-invariant elimination. Performance normalized to vectorization with dynamic warp formation is plotted in Figure 10. Average speedup is 11.3%, yet some applications achieve considerably higher performance with static warp formation than with dynamic warp formation. *MersenneTwister* experienced a 4.9x slowdown with dynamic warp formation, but static warp formation and thread invariance achieved a 1.30x speedup over completely scalar execution. The boost in performance is likely due to constrained warp formation in the presence of irregular control flow behavior.

7. RELATED WORK

Karrenberg [19] and Shin [20] present approaches to vectorization that focus on conditional select operators. These works replace conditional control-flow with conditional data-flow and rely on predication in combination with control-flow graph restructuring transformations to accommodate divergence. Predication is a light-weight technique for disabling divergent or terminated threads along some control paths but reduces SIMD utilization. Instructions predicated off which cannot be vectorized incur additional penalties, as they occupy pipeline stages yet their results are discarded. Their evaluation includes several optimizations that were not implemented for this work such as coalescing of affine vector loads and stores.

Stratton *et al.* [5] propose several approaches to translate the PTX execution model for efficient execution on multicore CPUs. Stratton describes a source-to-source translator that inserts nested thread loops into the control structures of a CUDA kernel's abstract syntax tree. Live values spanning multiple thread loops are expanded into arrays indexed by thread ID. Scalar threads are entirely serialized, and memory accesses are dramatically reordered across threads. Diamos *et al.* [16] describe the translator from PTX to LLVM on which this work was based. The multicore CPU backend to GPU Ocelot serializes scalar threads similarly to [5]. This work revealed the memory reordering problem and opportunities to exploit control-flow uniformity.

In [21], Steffen *et al.* present a hardware mechanism for terminating kernels that have executed divergent branches and spawning continuations that execute after a grouping phase chooses threads waiting to execute the same branch target. This technique incurrs overheads for all branches regardless of uniformity and does not immediately support CTA-wide synchronization barriers. Launching continuations on control-flow divergence through specialized hardware support is similar to what the dynamic execution manager of this work performs using software.

G-Streamline [22] controls the incidence of thread divergence by re-mapping tasks to threads. While such a re-mapping is trivial in implementations of the proposed technique, this work does not investigate scheduling nor task distribution heuristics to maximize control-flow utilization. Rather, it assumes that in any mapping, divergence is possible and requires a context switch betweeen specializations for different warp sizes.

Other approaches to portable vectorization such as Liquid SIMD [23] proposed by Clark *et al.* encode vectorizable operations as sequences of annotated scalar operators that are promoted to vector types by a dynamic compilation environment at runtime. This provides portability in terms of vector widths without incurring significant translation overhead. However, Liquid SIMD is applicable to program representations that have already been vectorized, perhaps with a technique such as proposed in this work. It does not approach the problem of vectorizing collections of scalar threads with correlated control flow. Barik [24] *et al.* present an algorithm for efficiently and automatically vectorizing scalar code by forming short vectors from independent instructions, using horizontal vector operators, and by algebraic simplification. Like other classical approaches to vectorization, the initial representation is not a collection of data-parallel threads but instead focuses on vectorizing scalar threads. The set of optimizations proposed is complimentary to dynamic vectorization presented here and

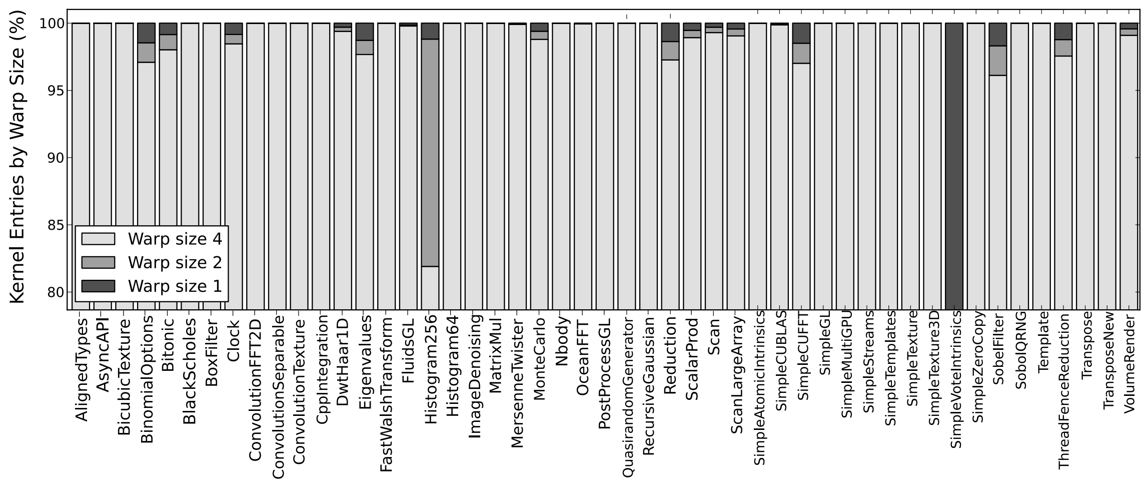


Figure 7: Average warp size of executed kernels with maximum warp size of 4 threads.

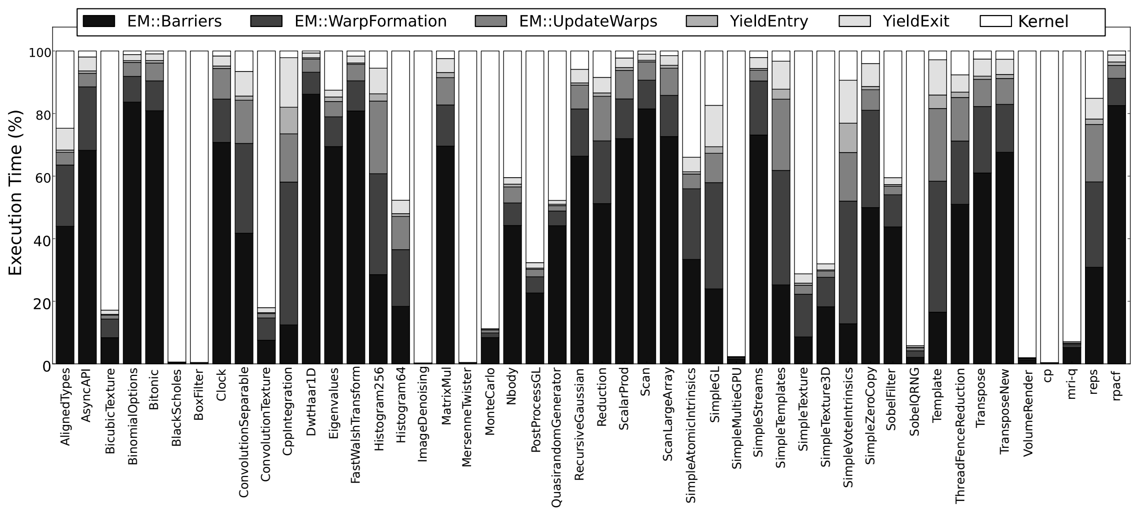


Figure 9: Fraction of cycles in execution manager (EM), yields to and from the EM, and executing subkernel.

might be used after vectorization to improve the quality of generated code.

8. CONCLUSIONS

This research shows explicitly data-parallel kernels can be compiled for efficient execution on modern multicore CPUs leveraging vector and SIMD functional units while tolerating control-flow divergence. We present a program transformation for specializing a kernel representation for various vector widths and propose a method for accommodating divergent control flow instructions via a light-weight virtual context switch implemented by compiler-inserted handling blocks. A dynamic execution manager orchestrates the execution of collections of threads by forming warps from a pool of ready threads with identical entry locations. An implementation of this technique is evaluated within GPU Ocelot, a research compilation framework for heterogeneous platforms. We apply dynamic vectorization to real-world workloads from existing GPU compute applications.

Microbenchmarks demonstrate near-peak computational throughput on GPUs and, with dynamic vectorization, peak throughput on an Intel Sandybridge CPU with vector ISA extensions. This technique is expected to scale across multiple vector widths and is not coupled to features of particular instruction set extensions. Consequently, it is applicable to other processor architectures with vector accelerator units such as PowerPC and ARM. Moreover, this technique does not require hardware support for divergence and provides dynamic compilation support for deploying data-parallel kernels on systems composed of both GPUs and multicore CPUs. Future work focuses more on the interaction within the complex memory hierachies of future systems to further improve the quality of vectorization.

9. ACKNOWLEDGEMENTS

The authors would like to thank Vinod Grover, Nathan Clark, and Scott Mahlke for discussions relating to dynamic compilation for vector architectures and to the anonymous reviewers for their recommendations and feedback. This research was supported in part by NSF under grants IIP-1032032, CCF-0905459, by LogicBlox Corporation, and equipment grants from NVIDIA Corporation.

10. REFERENCES

[1] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018 in Intel

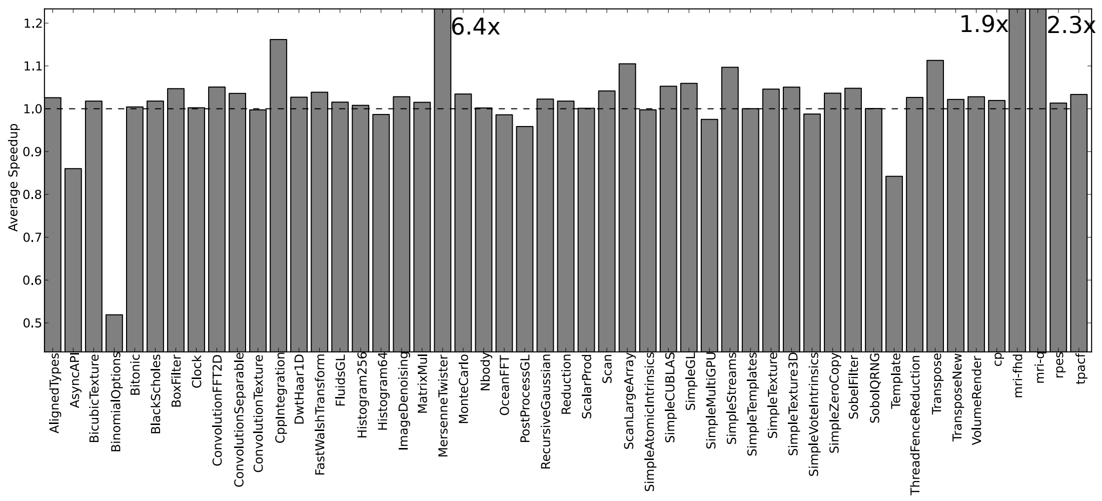


Figure 10: Speedup of static warp formation with thread-invariant elimination over dynamic warp formation.

64 and IA-32 Optimization Manaul. Intel Corporation, March 2009.

[2] Intel Corp. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*, March 2008.

[3] KHRONOS OpenCL Working Group. *The OpenCL Specification*, December 2008.

[4] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA Corporation, Santa Clara, California, 2.1 edition, October 2008.

[5] John Stratton and Vinod Grover et al. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO 2010*, Toronto, Canada, April 2010.

[6] Jayanth Gummaraju and Laurent Morichetti et al. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[7] Jaejin Lee and Jungwon Kim et al. An opencl framework for heterogeneous multicores with local memory. PACT '10, pages 193–204, New York, NY, USA, 2010. ACM.

[8] Haicheng Wu, G. Diamos, Si Li, and S. Yalamanchili. Characterization and transformation of unstructured control flow in gpu applications. In *First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, June 2011.

[9] NVIDIA. *NVIDIA Compute PTX: Parallel Thread Execution*. NVIDIA Corporation, Santa Clara, California, 1.3 edition, October 2008.

[10] Larry Seiler and Doug Carmean et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[11] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. Divergence analysis and optimizations. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 320 –329, oct. 2011.

[12] Sylvain Collange and David Defour et al. Dynamic detection of uniform and affine vectors in gpgpu computations. Technical report, Universite de Perpignan, University of California Davis, June 2009.

[13] Ziyu Guo, Eddy Zheng Zhang, and Xipeng Shen. Correctly treating synchronizations in compiling fine-grained spmd-threaded programs for cpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 310 –319, oct. 2011.

[14] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *IISWC'09*, Austin, TX, USA, October 2009.

[15] Gregory Diamos, Andrew Kerr, and Sudhakar Yalamanchili. Gpuocelot: A binary translation framework for ptx., June 2009. `http://code.google.com/p/gpuocelot/`.

[16] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[17] IMPACT. The parboil benchmark suite, 2007.

[18] Volkov Vasily and Demmel James W. Benchmarking gpus to tune dense linear algebra. In *Supercomputing'08*, Piscataway, NJ, USA, 2008.

[19] Ralf Karrenberg and Sebastian Hack. Whole-function vectorization. CGO, 2011.

[20] Jaewook Shin. Introducing control flow into vectorized code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 280–291, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Michael Steffen and Joseph Zambreno. Improving simt efficiency of global rendering algorithms with architectural support for dynamic micro-kernels. MICRO '43, Washington, DC, USA, 2010.

[22] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 369–380, New York, NY, USA, 2011. ACM.

[23] Nathan Clark and Amir Hormati et al. Liquid simd: Abstracting simd hardware using lightweight dynamic mapping. In *HPCA '07*, pages 216–227, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Rajkishore Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. MICRO '43, pages 201–212, Washington, DC, USA, 2010. IEEE Computer Society.