SIMD Divergence Optimization through Intra-Warp Compaction

Aniruddha S. Vaidya*
Intel Corporation
Santa Clara, CA, USA
asvaidya@gmail.com

Anahita Shayesteh
Intel Corporation
Santa Clara, CA, USA
anahita.shayesteh@intel.com

Dong Hyuk Woo
Intel Corporation
Santa Clara, CA, USA
dong.hyuk.woo@intel.com

Roy Saharoy
Intel Corporation
Santa Clara, CA, USA
roy.saharoy@intel.com

Mani Azimi
Intel Corporation
Santa Clara, CA, USA
mani.azimi@gmail.com

ABSTRACT

SIMD execution units in GPUs are increasingly used for high performance and energy efficient acceleration of general purpose applications. However, SIMD control flow divergence effects can result in reduced execution efficiency in a class of GPGPU applications, classified as divergent applications. Improving SIMD efficiency, therefore, has the potential to bring significant performance and energy benefits to a wide range of such data parallel applications.

Recently, the SIMD divergence problem has received increased attention, and several micro-architectural techniques have been proposed to address various aspects of this problem. However, these techniques are often quite complex and, therefore, unlikely candidates for practical implementation. In this paper, we propose two micro-architectural optimizations for GPGPU architectures, which utilize relatively simple execution cycle compression techniques when certain groups of turned-off lanes exist in the instruction stream. We refer to these optimizations as basic cycle compression (BCC) and swizzled-cycle compression (SCC), respectively. In this paper, we will outline the additional requirements for implementing these optimizations in the context of the studied GPGPU architecture. Our evaluations with divergent SIMD workloads from OpenCL (GPGPU) and OpenGL (graphics) applications show that BCC and SCC reduce execution cycles in divergent applications by as much as 42% (20% on average). For a subset of divergent workloads, the execution time is reduced by an average of 7% for today's GPUs or by 18% for future GPUs with a better provisioned memory subsystem. The key contribution of our work is in simplifying the micro-architecture for delivering divergence optimizations while providing the bulk of the benefits of more complex approaches.

*Now with NVIDIA Corporation

ISCA '13 Tel-Aviv, Israel

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Single-Instruction-Stream, Multiple-Data-Stream Processors (SIMD)*; I.3.1 [**Computer Graphics**]: Hardware Architecture—*Graphic Processors*

General Terms

Performance, Design, Experimentation

Keywords

GPU, SIMD, branch divergence

1. INTRODUCTION

General-purpose computation on graphics processing units or GPGPU computing [29] has rapidly become an important segment of parallel architecture computing wherein certain highly data parallel chunks or "kernels" of computation from an application are offloaded to GPUs for high performance and energy efficient acceleration [15]. As a result, the architecture of various high-end, commodity discrete GPUs and integrated GPUs on general purpose processors have been enhanced in significant ways to support high-performance GPGPU computation. Some recent examples of the discrete GPU include NVIDIA® Kepler™ [28] and AMD Radeon™HD 7970 [1] while 3rd Generation Intel® Core™procssor (formerly known as Ivy Bridge) [13] and AMD A10-4600M APU (formerly codenamed as Trinity) [34] belong to the integrated GPU class. These GPUs can efficiently execute data parallel programs developed in one or more GPGPU programming languages such as NVIDIA® CUDA™ [27], OpenCL [20], and DirectCompute [24].

The primary compute elements in most of these GPGPU architectures are multiple, highly-threaded SIMD compute cores (sometimes called SIMT cores). SIMD ALUs by definition have a single control flow for a wide number of data paths called channels that require lock-step execution. They also typically have wide datapaths to access several data elements at a time to match the compute capacity of multiple channels on ALUs. Such an architecture can deliver high performance and execution efficiency when SIMD control flow remains synchronized across all channels and memory requests are aligned resulting in good utilization of bandwidth offered by the wide memory interface. Unfortunately,

in some instances during execution, SIMD control flow cannot remain synchronized. Such SIMD control flow divergence may occur during the execution of an "if /else" conditional block. Some lanes may execute the "if" portion while the other lanes may execute the "else" portion depending on the branch result of each lane. A common solution to address such "branch divergence" hazard transforms the control flow problem into a data flow problem by sequentially executing all the control flow paths for all channels — in the example above, both the "if" portion of the block and the "else" portion are executed in turn by all channels — but by predicating (turning off) appropriate channels in each path [22, 12]. With this approach, each level of nesting of control flow divergence can result in significant performance (compute throughput) loss. The loss of compute throughput due to such diminished SIMD efficiency, *i.e.,* the ratio of enabled to available lanes, is called the SIMD divergence problem or simply compute divergence. We also classify applications that exhibit a significant level of such behavior as divergent applications. As stated earlier, well aligned memory requests across SIMD lanes that result in just a single or a small number of contiguous cache line accesses are also important for good SIMD efficiency. Otherwise, unaligned memory requests or memory divergence results in memory stalls and poor utilization of memory system bandwidth.

To address these problems, especially the former, recent studies have proposed several different solutions including micro-architectural techniques [12, 23, 11, 25, 30], compiler-based approaches [9, 14], hybrid micro-architectural and compiler-based approaches [5], and application restructuring [33]. The leading approaches, thread block compaction (TBC) [11], and related approaches, such as large warps micro-architecture (LWM) [25] and CAPRI [30], work in essentially similar ways, where recombining is restricted to threads that belong to the same thread block or workgroups (a unit of data parallelism defined by certain GPGPU programming models). These techniques use a thread-block-wide reconvergence stack shared by all threads in a thread block so as to exploit their control flow locality while reducing memory divergence and addressing semantic correctness issues generated by a previous technique [12].

While TBC and LWM provide good performance improvement for divergent applications, they introduce complexity in the micro-architecture of SIMT ALUs in GPUs. First, instruction scheduler needs to implement implicit warp barriers for synchronizing PC-values across warps at divergence points to enable compaction opportunities. Second, each of the combined source warps should have execution state that needs to be tracked and restored. In a practical implementation, such execution state may include dependence scoreboard, lane exception state, and performance counters. Third, these approaches also require highly-banked, per-lane addressable register files, which have an additional area and energy cost. Furthermore, combining warps can increase memory divergence (*i.e.,* the number of distinct memory or cache line requests per SIMD instruction), which can lead to performance loss. Overall, the combination of micro-architectural changes required for managing warps makes these techniques less desirable for practical implementations.

This paper approaches the problem from the opposite view of practical, low-complexity micro-architectural techniques that can address compute divergence in SIMD GPU applications while not adversely impacting their memory divergence. It is worthwhile to note that our proposal exploits multi-cycle execution micro-architectural feature that already exists in many GPUs. In Intel's Ivy Bridge GPUs [16], the number of lanes in a SIMD instruction (sometimes referred to as the warp-width) varies from 8, 16 to 32. This number is 32 for NVIDIA GPUs [26] and 64 for AMD's GPUs [1]. However, the corresponding number of hardware execution lanes is typically a fraction of the SIMD instruction width: 4-wide SIMD ALU in the case of Intel's Ivy Bridge GPU, 8-wide for NVIDIA® GeForce® 8880 [26], and 16-wide for AMD Radeon™7970 [1]. This implies that each wide SIMD instruction typically executes over multiple execution cycles due to narrower hardware width. There is a good reason for such multi-cycle execution; GPUs typically have large register files. An area and power efficient implementation of these large register files limit the number of concurrent read/write ports (typically single ported). Operand reads/writes for multi-operand instructions (such as 3r-1w FMA instruction) would thus be done over multiple cycles (four in the FMA example). As a result, executing one SIMD instruction over four cycles in a GPU's SIMD ALU results in a well-balanced system. In our approach, we utilize this feature to obtain higher SIMD efficiencies in the presence of divergence with relatively simple micro-architectural techniques. In particular, we present two techniques that compress cycles during the execution of instructions with subset of channels turned off.

- Basic Cycle Compression (BCC): BCC looks at squeezing out cycles in the execution pipeline in cases where any aligned set of four channels are turned off.
- Swizzled Cycle Compression (SCC): SCC is a generalized form of BCC where channel positions are swizzled prior to execution so that that they form groups of 4-aligned enabled and disabled channels to squeeze out idle execution cycles for those 4-aligned disabled channels.

Our proposed techniques are similar to density-time optimization for divergent control flow in vector architectures [32, 2, 4]. In particular, BCC is similar to "element group skipping" [2] when considering a set of vector execution pipes. It appears that this optimization was considered to have limited potential in the context of vector architectures and its performance was not sufficiently characterized previously. However, our work finds BCC to provide good performance benefits for various divergent GPGPU workloads. Further, SCC is a novel optimization.

In this paper, we present a detailed performance evaluation of our proposal using a cycle-level simulator of a hypothetical GPGPU architecture loosely based on Intel's Ivy Bridge GPUs (a GPU in 3rd Generation Intel® Core™) processor with several OpenCL-based GPGPU workloads and a handful of graphics workloads. Our evaluations are based on two approaches — execution-driven simulation in some instances and trace-based profiling in others. Our novel contributions that distinguish ours from prior work include the followings.

1. We present a novel approach to SIMD divergence optimization in GPGPU architectures called intra-warp compaction and introduce two practical techniques with relatively simple implementations. The architecture details for these techniques further demonstrate this claim. Prior work on inter-warp compaction requires significantly higher implementation complexity.

2. Inter-warp divergence techniques can have the significant drawback of causing increased memory divergence. Both coherent and divergent applications can be adversely impacted by this. Our techniques intrinsically do not create additional memory divergence beyond what may already exist in an application.
3. Most prior results have evaluated their techniques on NVIDIA style GPUs and with NVIDIA® CUDA™ benchmarks. Our work presents insights for a distinct GPU architecture, Intel's GPU architecture. Our evaluations use a large number of OpenCL benchmarks as well as traces from several additional OpenCL workloads and 3D graphics workloads.

The rest of the paper is organized as follows: Section 2 gives an overview of the candidate architecture and its control flow divergence behavior. In Section 3, we describe our proposed optimization techniques followed by a description of the micro-architectural changes required in Section 4. We describe our performance analysis tools, methodologies, and workloads and discuss the performance results in Section 5. Related work is discussed in Section 6. Finally, Section 7 summarizes the work presented here.

2. BASELINE GPU ARCHITECTURE

2.1 Intel Ivy Bridge GPU Architecture

In this work, we use the Intel's recent Ivy Bridge-like GPU as our baseline architecture. The high level block diagram of Intel's recent Ivy Bridge GPU is shown in Figure 1. For GPGPU computation, the GPU device driver issues commands to the device through a command streamer interface. This results in the dispatch of work to the compute cluster via the thread dispatcher. We focus our description on this GPU compute cluster. The compute cluster consists of several shader cores, called execution units (EUs). Ivy Bridge GPU configurations have six EUs for the HD2500 version and 16 EUs for HD4000. In addition to the EUs, there are various other blocks, which will be described after we present the EU architecture and pipeline.

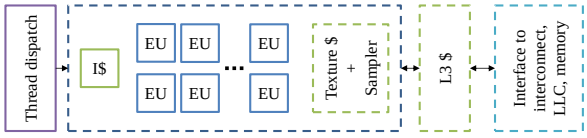


Figure 1: High-level architectural block diagram.

2.2 EU Architecture

Each EU in Ivy Bridge GPU is a multi-threaded SIMD core that supports between six and eight hardware threads (called EU threads) depending on the configuration. Each EU executes instructions from a variable width SIMD ISA [17]. Apart from some special purpose architectural registers, each EU thread has a general-pupose register file (GRF) with 128 256b-wide registers.

Note that our baseline EU pipeline is similar to a hypothetical pipeline loosely based on that of the Ivy Bridge GPU's EU and other GPU simulation models [3]. The basic pipeline operations are based on the available public sources [19, 17]. These pipestages are as follows:

1. Per thread **instruction prefetch stage** prefetches instructions to an instruction queue from L1 instruction cache when the queue is running low.
2. Per thread **instruction decode stage** decodes instructions (compressed and compact format), which can be of variable SIMD widths of 1, 4, 8, 16, or 32 for different instruction types. The final value of SIMD execution masks are computed concurrently with the instruction decode stage. These execution masks are used by the BCC/SCC control logic for SIMD divergence optimizations that are described later.
3. Per thread **scoreboard/dependence-check stage** checks/sets dependencies and queues instructions that pass dependence check for dispatch.
4. **Thread arbitration stage** arbitrates among hardware threads with issueable instructions (rotating/age-based priority arbiter assumed) to issue instructions to the SIMD ALUs. Up to two instructions from distinct threads can be issued per arbitration pass [19, 13, 17]. Currently, EUs can issue two instructions every two cycles. BCC/ SCC control logic would have time from the end of the execution mask computation above, until the end of this stage (*i.e.,* prior to operand fetch and execute stages) to determine appropriate control settings for cycle compression.
5. **Address computation/operand fetch/swizzling stage** computes an address and an appropriate index of GRF operands that need to be fetched. Most EU instructions can optionally encode 4-lane input operands swizzles. If selected, swizzling hardware delivers swizzled operands from the register file to the ALU.
6. **Execute stage** executes a given instruction. Typical 32b operand instructions are executed in two 4 lane-wide ALUs, i.e. FPU (Most int and floating point instructions including FMAs) and EM (most extended math instructions such as divide, sqrt, sin, cos, log, exp instructions as well as most floating point ops and FMAs). Since the SIMD width of instructions is variable, instruction execution happens in multiple waves of 4 lanes per cycle in each ALU. For example, a SIMD8 floating point ADD takes 2 execution cycles while a SIMD16 floating point add takes 4 execution cycles. Memory ops and barriers/fences are handled by special "SEND" instructions through a separate pipe.
7. **Write-back stage** writes back an output value and clears dependency for the appropriate EU thread.

With these pipeline stages, EU ISA supports complex register indexing usage utilized for 3D graphics and media applications where an instruction's operand may span subfields in multiple registers. Operand fetch in such cases is more complex and may require several cycles. Furthermore, instructions support predication of individual lanes both explicitly and implicitly (when driven by execution of control flow instructions).

2.3 Memory Subsystem and Execution Model

A group of EUs share an L1 instruction cache, texture caches, and samplers in the second level of the hierarchy. All EUs share a data cache referred to as the L3 cache. A group of EUs access the L3 data cache and a highly banked and fast shared local memory through an interface called a data-cluster. L3 misses are looked up in the last-level cache that is shared with the main processor cores and eventually in main memory via the DDR3 memory controllers. We have also assumed the standard OpenCL execution model and terminology [20] in this work.

3. DIVERGENCE OPTIMIZATION

As mentioned earlier, SIMD control flow divergence increases the dynamic count of SIMD instructions even though only a subset of channels are enabled and doing useful work. Consequently, performance improvement can be achieved through compression of some or all of the wasted, idle execution cycles.

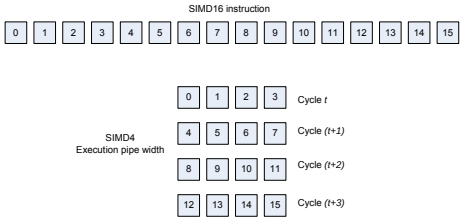


Figure 2: Execution of a SIMD16 instruction.

To explore such an opportunity, here we detail Intel's Ivy Bridge graphics architecture. It derives high efficiency at small hardware cost by using a large number of SIMD channels (called the SIMD execution width), which is a multiple of the actual hardware datapath width. In this architecture, for example, GPGPU SIMD instructions typically have a width of 8 or 16 (even up to 32 for DirectCompute kernels) [16, 18] while the FPU in an EU pipeline is 4-wide (executing four parallel FP operations per cycle). Maximum effective throughput for a set of independent back to back SIMD16-wide instructions for this FPU would be one instruction completed every four cycles due to the difference between the instruction width and FPU width. A set of four contiguous channels out of the 16 in the instruction are sequenced through the 4-wide FPU over four cycles in a pipelined manner as shown in Figure 2. Similarly, a SIMD8 instruction would flow through the pipeline in two cycles.

To access the potential for divergence optimizations through execution cycle reduction, we define a metric called "SIMD efficiency" as the ratio of the average number of enabled channels per executed instruction and the average SIMD width of the executed instructions over the entire kernel execution. The raw SIMD efficiency of more than 65 OpenCL and 3D graphics applications studied is shown in Figure 3. As shown in the figure, applications with high SIMD efficiency (more than 95%) exhibit very few divergent instruction execution in the code and are referred to as "coherent" applications. On the other hand, the lower SIMD efficiency is from 1.0, the higher the performance potential from divergence optimizations is. These so-called "divergent" applications are the focus of our micro-architectural optimization techniques for compression of execution cycles.

3.1 Basic Cycle Compression (BCC)

BCC harvests execution cycles, *i.e.,* compress instruction execution, in those cases where contiguous sets of channels are disabled in such a manner that one or more execution cycles would have been dead cycles, *i.e.,* where no useful computation would have otherwise happened in the execution pipeline. These opportunities can be identified after the instruction decode stage by examining the final computed value of the SIMD execution mask. For each such set of channels whose execution can be eliminated, the corresponding operand fetch, micro-op issue to the execution pipe, and write-back of the results are suppressed. Instead, corresponding operations for the subsequent set of channels are issued. If there is no (additional) work to be issued for the current instruction, then the available slot is used for subsequent instructions.

This is sketched out in Figure 4 (a), which captures an example of an "if/else" segment of code. The "if" block is executed by only four out of the 16 channels (conversely the "else" block by 12 out of the 16 channels). With BCC, three out of four cycles of execution of an instruction in the "if" block can be harvested. The savings in the "else" block execution is limited to one. In total, a half of the execution cycles (four out of eight) can be harvested by BCC for the complete execution of the block. In this particular example, the compressed execution time is equivalent to the time needed for execution of the instruction without the divergence caused by the "if/else" clause. Even though divergence optimizations discussed so far were limited in context to control flow divergence, BCC can harvest execution cycles in all cases where dispatch, control flow, or predication results in the disabling of channels. EUs support various types of masks for disabling specific channels in various contexts, such as during the dispatch of an EU thread, predication of individual instructions, and predication in handling control flow instructions. Micro-architecture details and performance of BCC optimization scheme are discussed in later sections.

3.2 Swizzled Cycle Compression (SCC)

Unfortunately, some divergence patterns do not favor BCC. In particular, BCC cannot take advantage of cycle compression opportunities when turned off channels in an instruction are not contiguous or contiguous but not favorably aligned to the hardware SIMD pipeline width. One such example is illustrated in Figure 4 (b). In order to address this case, we have proposed the second optimization referred to as "swizzled cycle compression (SCC)".

In the SCC approach, each SIMD instruction is examined with an eye on the number and positions of disabled channels. The positions of disabled and enabled channels are then arranged into groups such that the groups of disabled channels can create harvestable dead execution cycles like in the BCC above. Such an example is showed in Figure 4 (c).

Rearranging channel positions is done through operand swizzling (permutation) hardware prior to dispatch to the execution pipeline. The destination operand positions have to be correspondingly unswizzled prior to writeback to the register file. Although SCC can optimize the divergence pattern of Figure 4 (b) within a single or repeated warps, it is worth noting that TBC-like approaches cannot when it is repeated across warps because those optimizations preserve lane/channel positions. Micro-architecture details and performance potential of SCC is discussed later.

4. MICRO-ARCHITECTURE

4.1 BCC Micro-Architecture

Hardware support required for enabling BCC optimization is relatively straightforward. The required functionality is specific to identification of cases where parts of operand fetch and instruction execution can be disabled based on the execution mask and the operand datatype in use. In EUs of Ivy Bridge GPUs, information from the instruction encoding (*e.g.,* instruction predicate mask) and channel mask registers (dispatch mask, mask for conditional block, loop,

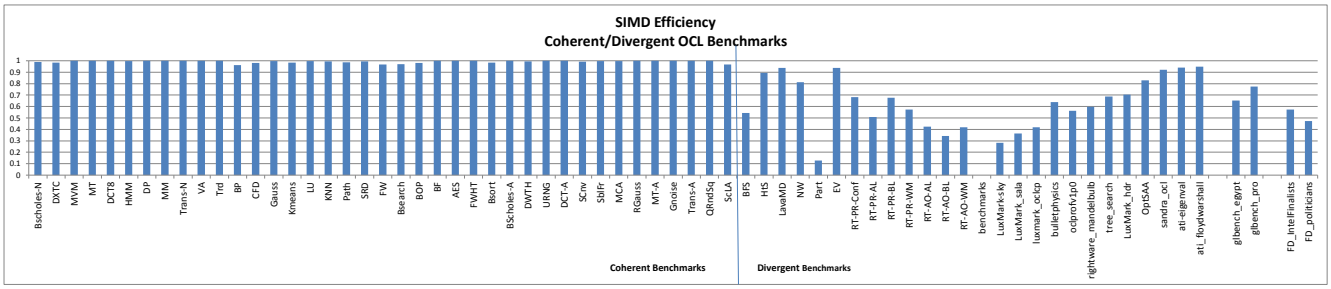


Figure 3: SIMD efficiency of various OpenCL applications on Ivy Bridge architecture.

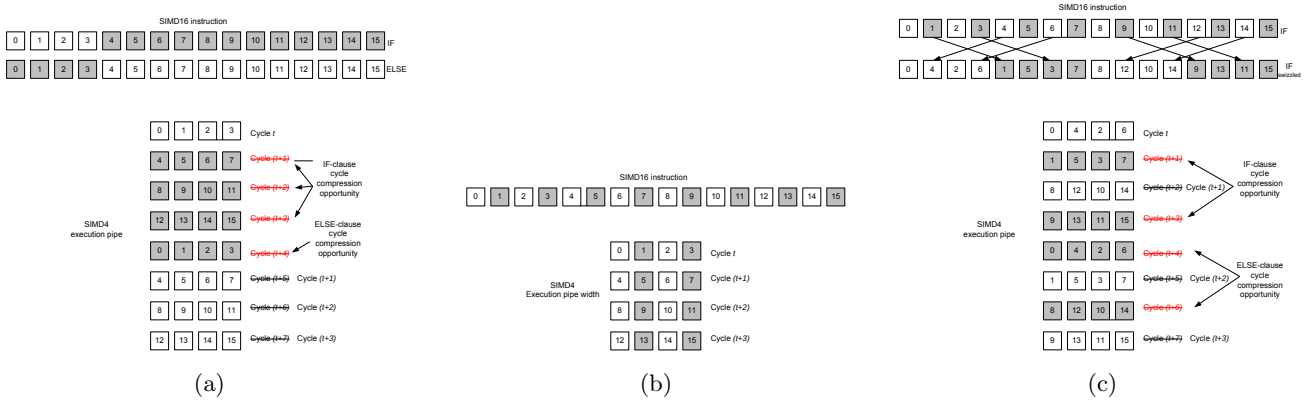


Figure 4: (a) BCC in IF/ELSE blocks, (b) An example of an unfruitful case for BCC, (c) Swizzling of channel positions to coalesce enabled and disable channels using SCC.

etc.) are used to dynamically compute execution mask of an instruction. Subsequently, the logic associated with BCC determines which parts of the operand fetch/execute/writeback need to be actually issued to the execution pipe. For example consider a SIMD16 32b add instruction:

ADD(16) R12, R8, R10 [Exec - mask: 0xF0F0]

Each register is 256b wide, and SIMD16 operation accesses for the 32b datatypes implicitly use pairs of registers, *i.e.,* R12-13, R8-9, and R10-11. We assume that micro-architecture support for BCC treats a SIMD16 macro instruction as four quartile operations or "micro-ops", *i.e.,* ADD.Q0 through ADD.Q3. Each micro-op would access a 128b portion of the register. Internal to the EU pipeline, this would represent the micro-op sequence as follows:

ADD.Q0 (4) R12.H0, R8.H0, R10.H0
ADD.Q1 (4) R12.H1, R8.H1, R10.H1
ADD.Q2 (4) R13.H0, R9.H0, R11.H0
ADD.Q3 (4) R13.H1, R9.H1, R11.H1

In the above sequence R8.H0 and R8.H1 imply the lower and upper half (128b portion) of the register R8. Given the execution mask 0xF0F0 for the 16 channels, BCC suppresses the issue of ADD.Q0 and ADD.Q2 micro-ops. Note that the corresponding operand fetches/write-backs for the unissued micro-ops are also not required which in turn offers register file access energy savings. A relatively small change to the register file data path to fetch only half width (128b) registers is required. The original (baseline) register file and modified register file for BCC are shown in Figures 5 (a) and (b).

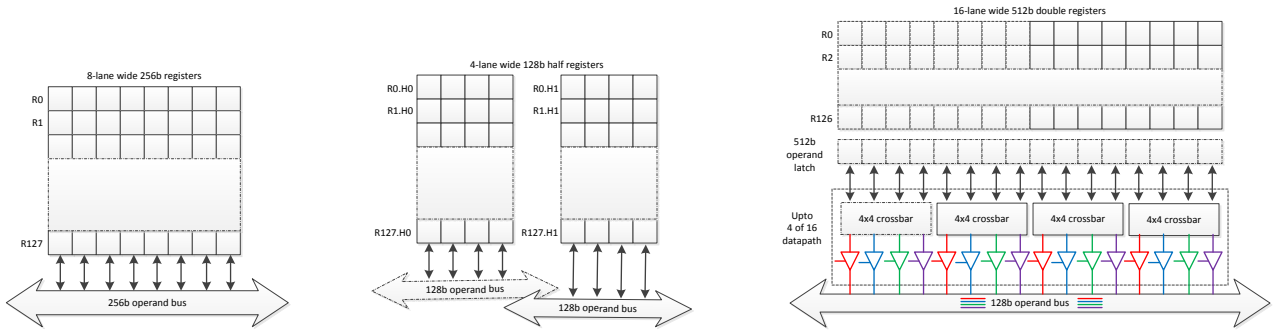
While we have described the above example assuming 32-bit (floating point) operands, the actual number of execution cycles and the number of operand registers accessed would depend on datatypes and execution width of the instruction. For example, the number of execution cycles can be lower or higher based on whether operands are short integer, float, or double precision.

Based on the assessment of the micro-architecture changes to the decode and issue logic as well as the register file datapath, there is no performance penalty for implementing the changes to support the BCC optimization. The overall benefit of BCC depends on the relative frequency of low SIMD efficiency while executing a kernel with favorable patterns of turned off channels. The benefit from execution cycles saved also depends on datatypes used by instructions executed. Benefits may be higher for wider datatypes (doubles and long integers) that take more cycles through the execution pipe, and conversely, benefit may be lower for narrow datatypes (half float/short)

4.2 SCC Micro-Architecture

SCC requires control logic to identify an appropriate swizzle setting for the operands. This is based on the computed execution mask so as to compact lanes. On the other hand, unswizzle settings, required for the write-back stage, are simply the inverse permutation of the operand swizzle settings.

The simplified operand datapath required for accessing a single operand for our SCC micro-architecture is illustrated in Figure 5 (c). A full-width operand fetch (16 lanes, 512b) is done in 1-cycle. This operand is held in a 512b latch (as 4 x 128b quad). Each quad passes through a four lane swizzler with individual lane enables. The swizzled outputs for each of the four quads is a load on a 128b wired-OR bus so data path width matches the ALU width. The ALU



(a) (b) (c)

Figure 5: Register file organizations and 1-operand datapaths for (a) Baseline, (b) BCC, and (c) SCC.

consumes swizzled lanes over one to four cycles depending on the degree of execution cycle compression. Swizzle crossbar and lane select settings can be done in parallel with the register file access so operand access times are unchanged. The apparent overhead is the cost of four 4-lane crossbars and a wide 512b latch. Intel GPU includes ISA support for 4-lane operand swizzling so additional overhead is minimal. However, this may or may not be the case for other GPUs.

The remaining part of the problem is to design the control logic for determining which execution masks lead to compressed cycles and then to derive appropriate swizzle and "lane enable" settings. The C-language pesudo-code for the algorithm to compute key parts of the SCC control logic settings is shown in Figure 6. While there are various possible ways to permute lanes for deriving an optimal cycle count for SCC, the algorithm that we illustrate represents one such method. In particular, this algorithm attempts to minimize the number of intra-quad lane swizzles. Each contiguous set of four execution mask bits is called a quad. In the first phase, a population count of the enabled lanes in the execution mask is determined. This is in turn used to determine the optimal number of cycles required to execute the instruction. For each of the four lane positions corresponding to the execution width (4) of the SIMD ALU, the number of cycles that lane is active is determined. Compared to the optimal number of execution cycles, lanes may be under, over, or appropriately subscribed, which determines whether they are swizzle from, swizzle to, or unswizzled lanes. While the algorithm shows some of the steps as for loops, this can be computed by hardware in parallel for each lane and then used in each cycle for computing swizzle and enable settings.

Figure 7 shows an example, in which the optimal number of execution cycles are achieved using the algorithm from Figure 6. The lanes positions that appear in each cycle and the corresponding swizzle settings and lane enables derived for each execution cycle are also shown.

With these control signals, appropriate micro-ops with swizzled operands are issued to the SIMD ALU in a manner similar to that described for BCC in Section 4.1. Assuming a register file organization as shown in Figure 5 (c), there is no operand fetch bandwidth savings for SCC.

4.3 Additional Considerations

In the lack of a detailed power model for precise quantification of the energy implications of our proposal, we discuss it in qualitative terms where appropriate. BCC and SCC

```
a_q_cnt = #active quads in 16b exec mask
a_ln_cnt = total active lanes
a_ln_q[n] = queue of quads w/ lane n active
o_cyc_cnt = optimal cycles to execute
/* = ceil(a_ln_cnt/4), as hw exec width 4 */

if(a_q_cnt == o_cyc_cnt)
  /* skip empty quads, BCC-like. Done */
else /* initial setup work */
  for(n=0; n<4; n++){ /* exec width is 4 */
    a_ln[n] = qlength(a_ln_q[n])
    if(a_ln[n] > o_cyc_cnt)
      // lane n will swizzle when needed
      surplus[n] = a_ln[n] - o_cyc_cnt
    else
      surplus[n] = 0
    tot_surplus += surplus[n]
  }

/* in each cycle w/ work to schedule ... */
/* compute lanes to swizzle ... */
/* ... and which to enable in each quad */

for(c=0; c<o_cyc_cnt; c++) {
  for(n=0; n<4; n++) {
    if( !is_empty(a_ln_q[n]) )
      out_ln[n] = Qx /* lane not swizzled */
  /* Qx: quad# from dequeue(a_ln_q[n]) */
    else if( tot_surplus != 0 )
      out_ln[n] = Qy.swizzle(m,n)
      /* Qy: dequeue(a_ln_q[m]), surplus[m] > 0 */
      surplus[m]--; tot_surplus--
    else
       out_ln[n] is disabled
       /* No surplus, lane not filled */
  }
}
```

Figure 6: Algorithm to compute SCC settings.

optimizations offer dynamic energy reductions through opportunistic execution cycle reductions. Note that, with a BCC optimized register file, one can expect to save operand fetch energy in cases where BCC is effective. So, BCC is expected to provide both a performance advantage and energy savings given its simple control logic.

For an Intel GPU with existing support for operand swizzling, SCC does not require significant additional hardware in the datapath. However, in absence of any such pre-existing support, the datapath would incur the cost of additional crossbars and wire-lengths required for swizzling. Further, SCC control logic is more complex than that of

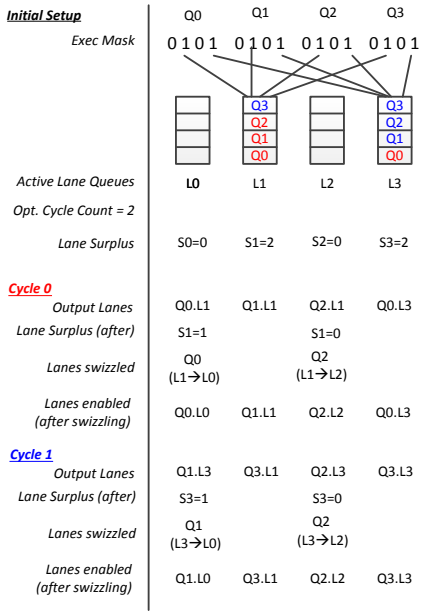


Figure 7: SCC settings computation example.

BCC, thus, for applicable instructions, there is likely to be a modest increase in control logic power. However, we are unable to quantify these effects more precisely.

We also have compared the area of a BCC optimized register file against a 8-banked, per-lane addressable register file required by inter-warp techniques [12, 11]. Using the 32nm model of CACTI 5.x models, we found that the register file area overhead of BCC is 10% compared to a baseline Ivy Bridge style register files (256b). On the other hand, the register file area overhead of inter-warp techniques turns out to be higher than 40%. On the other hand, the register file for SCC is wider but shorter than the baseline register file (due to reduced addressing overhead). Note that this comparison does not include the additional area for swizzling crossbars and lane enable signals that may be required for SCC.

Figure 5 showed the register file organizations for baseline, BCC, and SCC configurations for a single operand only. There are multiple practical ways to support multiple read and write operand access to the register file without requiring more ports; (1) Register file can be accessed over multiple cycles, *e.g.,* four cycles to provide three read and one write access with a single ported, single pumped register file. (2) Multiple parallel register file banks can be used for non-conflicting operand accesses. (3) Multi-pumping the register file is another option (register file clock is a multiple of ALU clock). For BCC and SCC which cause execution cycle reduction, multi-pumping and multi-banking are the preferred options.

Finally, as BCC and SCC both increase the overall throughput of the EUs, adequate instruction fetch bandwidth and front-end processing bandwidth (decode, dependency check, and scheduling bandwidth) may be needed to balance the higher rate of execution, fetch, and issue due to cycle compression. The level of increases in these units required by the architecture may be based on the distribution of cycle compression opportunities in the typical workloads (smaller if typical bursts are short-lived; higher if long sustained periods of cycle compression are common).

5. PERFORMANCE EVALUATION

5.1 Evaluation Methodology

In this paper, we have used both execution-driven and trace-based simulation methodologies for performance evaluation of the BCC and SCC proposals. GPGenSim, an in-house GPGPU performance simulator, was used for execution-driven approach. It is a cycle-level simulator that captures the abstract micro-architecture of an Ivy Bridge style GPU as described in Section 2. In the stand-alone (GPU only) version, GPGenSim enables execution of full OpenCL applications (CPU host + GPU) but only the GPU performance behavior, *i.e.,* that of OpenCL kernel executions on the GPU, is modeled; CPU host execution is functional only. GPGenSim includes detailed performance models for multi-threaded, dual-issue, SIMD ALU pipeline, highly banked memory (shared local memory), GPU data cache, and the shared last-level cache (LLC) banks with the CPU cores. The simulation model can be configured with a range of machine parameters such as number of EUs per compute cluster, number of clusters, and GPU cache-memory hierarchy latencies and bandwidths. GPGenSim interprets OpenCL kernel binaries and works in tandem with a full Ivy Bridge GPU functional model (independently developed by a different team) that drives the performance model. The accuracy of the EU model of GPGenSim for all major compute operations has been correlated with hardware measurements to within 2% using a range of OpenCL micro-benchmarks and kernels. The functional-level model executes the full EU instruction set [17]. In addition to the stand-alone mode used for the performance analysis presented in this paper, GPGenSim is also integrated in a parallel x86-64 CMP and processor "uncore" model to provide a full heterogeneous multicore + GPU execution-driven model. A detailed description of the entire modeling infrastructure is beyond the scope of this paper.

While the execution driven methodology of GPGenSim worked well for various OpenCL compute workloads, there were many other challenges for other OpenCL and non-OpenCL workloads. Some OpenCL workloads such as the OpenCLoovision (Face-Detection) application use the texture sampler and image buffer APIs, which are currently not supported by our performance model. For commercial benchmarks such as Sandra or RightWare, source code availability is an issue since one has to scale-down applications for a reasonable execution time on a simulator. In some other cases such as LuxMark or BulletPhysics, these applications have large code bases, which makes porting and tuning to our execution environment difficult. 3D Graphics benchmarks use a different binary format and ISA modes unsupported by our OpenCL/GPGPU focused performance model. In such instances, we have relied on a trace-driven simulation approach using the functional model only. We have instrumented the functional model to obtain SIMD execution mask for every executed instruction to enable computation of BCC and SCC benefit for all workload traces.

Execution driven simulation using GPGenSim infrastructure has been used to evaluate cycle compression techniques using more than 50 OpenCL workload collection from a combination of Rodinia2.0 [7, 8], other well known OpenCL HPC benchmarks, and two different in-house ray tracing workloads. With the exception of a handful of cases where scaling of the problem size was not possible, almost all workloads

Table 1: OpenCL workloads used.

Name	Comment	Name	Comment
Bscholes-N	Black Scholes (finance)	DXTC	DirectX texture compressor (3D Graphics)
MVM	Matrix vector multiplication (linear algebra)	MT	Mersenne twister (random number generator)
DCT8	8x8 discrete cosine transform	HMM	Hidden Markov model (speech processing)
DP	Dot Product (linear algebra)	MM	Matrix multiplication (linear algebra)
Trans-N	Matrix transpose (linear algebra)	VA	Vector addition (linear algebra)
Trd	Tridiagonal (linear algebra)	BP	Back propagation (pattern recognition)
BFS	Breadth-first search (graph algorithm)	CFD	CFD solver (unstructured grid)
Gauss	Gaussian elimination (structured grid)	HtS	Hot spot (thermal modeling, linear algebra)
Kmeans	Kmeans (clustering, molecular dynamics)	LavaMD	LavaMD (linear algebra, molecular dynamics)
LU	LU decomposition (linear algebra)	KNN	k-Nearest neighbors (linear algebra)
NW	Needleman-Wunsch (bioinformatics)	Part	Particle filter (medical imaging)
Path	Path finder (grid traversal)	SRD	Speckle reducing anisotropic diffusion
FW	Floyd Warshall (graph analysis)	Bsearch	Binary search
BOP	Binomial option pricing model (finance)	BF	Box filtering (image filtering)
AES	Advanced encryption/decryption standard	FWHT	Fast Walsh Hadamard Transfrom
Bsort	Bitonic sort	BScholes-A	Black Scholes-A (finance)
DWTH	Haar discrete wavelet transform	URNG	Uniform random number generator
EV	Eigen Value (linear algebra)	DCT-A	Discrete cosine transform
SCnv	Simple convolution (functional analysis)	SblFr	Sobel filter (image processing)
MCA	Monte Carlo Asian pricing (option pricing)	RGauss	Recursive Gaussian
MT-A	Mersenne Twister (random number generator)	Gnoise	Gaussian noise
Trans-A	Matrix transpose (linear algebra)	QRndSq	Quasi random sequence
ScLA	Scan large array	RT-PR-Conf	Ray tracing, primary rays, conference scene
RT-PR-AL	Ray tracing, primary rays, alien scene	RT-PR-BL	Ray tracing, primary rays, bulldozer scene
RT-PR-WM	Ray tracing, primary rays, windmill scene	RT-AO-AL	Ray tracing, ambient occlusion, alien scene
RT-AO-BL	Ray tracing, ambient occlusion, bulldozer scene	RT-AO-WM	Ray tracing, ambient occlusion, windmill scene
FD-IntelFinalists	Face detection IntelFinalists scene	FD-Politicians	Face Detection politicians scene

were run with large enough problem size (10M to 1B or more instructions). The key attributes of the workload collection used are shown in Table 1. Kernel execution on the simulator provides us the instruction stream; execution masks of instructions are then evaluated for BCC and SCC optimization opportunities independently, and the execution cycle savings are recorded. Note that the SCC optimization subsumes BCC and, therefore, its benefits are at least as much as that of BCC. For this approach, we provide detailed execution cycle compaction benefit results in Section 5.3 and execution time analysis results in Section 5.4.

On the other hand, we have used the trace-based simulation to evaluate nearly 600 workload traces. More than 200 of these were OpenCL workload traces, 45 of which showed over 5% benefit for intra-warp compaction. From the nearly 380 OpenGL (3D graphics) workload traces, 80 showed a benefit of 10% or more for intra-warp compaction. Divergent workloads in these trace sets include workloads like LuxMark (raytracing), Sandra and Rightware benchmarks, Face-Detection (OpenCLoovision), BulletPhysics, and the GLBench 3D graphics benchmarks. Detailed results on cycle reduction in EU using our trace-based approach is also included in Section 5.3. However, these benchmarks are not included in the execution time analysis discussion.

Despite the few limitations outlined above, this study represents one of the most comprehensive performance studies of SIMD divergence across a range of important classes of compute, media, and 3D-Graphics workloads.

5.2 Ivy Bridge Micro-Benchmark Study

First, divergence micro-benchmarking on Ivy Bridge GPUs demonstrates the presence of a limited-version of a BCC-like optimization. In this subsection, we discuss performance results from our micro-benchmarking study that can be used to infer the type of divergence optimization.

Our experiments show that, in Ivy Bridge EUs, SIMD16 instructions whose upper or lower eight lanes are inactive are executed in only two cycles instead of the four cycles

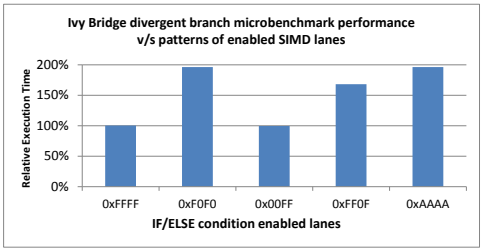


Figure 8: Ivy Bridge optimization observed.

expected. Figure 8 shows the execution times for different divergence patterns, which reveals that patterns such as 0x00FF and 0xFF0F are optimized. For example, one would have expected the execution time for the divergence pattern 0x00FF to be double that for the pattern 0xFFFF (no-divergence case, baseline) for balanced "if/else" blocks. However, Figure 8 shows that the time is exactly the same as that of the no divergence case (first and third bar from the left). Note that, a full BCC optimization would have made the relative execution time of the 0xF0F0 case to be identical to that of the no-divergence case, and an SCC-like optimization would have helped 0xAAAA case.

In our performance studies, we appropriately model the Ivy Bridge optimization by treating SIMD16 instructions with execution masks of upper or lower eight bits inactive as SIMD8 instructions and report intra-warp compaction benefit only after applying this optimization. In other words, we subtract-out the benefit from Ivy Bridge optimization in all our reported results on BCC and SCC optimizations. On other GPU architectures where such an optimization does not exist, the observed BCC and SCC benefits would be much larger than what is reported in this paper. To validate our performance evaluation methodology, we used divergence micro-benchmarks with various divergence patterns to correlate the calculated benefits from BCC and SCC against the GPGenSim simulation results. Table 2 summarizes the GPGenSim results for the performance benefits of

Table 2: Ivy Bridge Optimization, BCC, and SCC Performance Observed on GPGenSim

Loop Nesting Level	Execution Mask Example of a single branch path	Execution Masks All branch paths In Hex	BCC Benefit	Additional SCC Benefit	Ivy Bridge Optimization Benefit
L1	0101 0101 0101 0101	5555,AAAA		50%	
L2	0001 0001 0001 0001	1111,4444,8888,2222		75%	
L3	0000 0001 0000 0001	0101,1010,0404,4040,0808,8080,02020,2020	50%	25%	
L4	0000 0000 0000 0001	16 patterns with 1-bit set	25%		50%

Ivy Bridge optimization, BCC, and SCC, each applied in turn for multiple levels of nested branches.

5.3 Cycle Compaction Benefit

Earlier in Figure 3, we presented the aggregated results from all applications classified into *coherent* and *divergent* application based on *SIMD efficiency* [11] of above or below 95%. The results presented in this section are focused on divergent applications, which are the main beneficiary of our proposed techniques.

Figure 9 shows the SIMD utilization breakdown for all SIMD16 and SIMD8 instructions in divergent applications. No execution cycle compaction is possible in the following cases: SIMD8 instructions with 5–8 active lanes and SIMD16 with 13–16 active lanes. Other cases where intra-warp compaction with either BCC or SCC is possible are as follows: 1-cycle saving for 9–12 active lanes, 2-cycle saving for 5–8 active lanes, and 3-cycle saving for 1–4 active lanes in SIMD16 instructions; 1-cycle saving can also be achieved for 1–4 active lanes in SIMD8 instructions. 5-8 active lanes in a SIMD16 instruction may also represent an opportunity for the existing Ivy Bridge optimization depending on the position of the active lanes. In the figure, several benchmarks (Ray Tracing Ambient Occlusion and LuxMark runs) only show SIMD8 instructions. Here, the GPU complier generates a SIMD8 kernel (no SIMD16 operations), most likely due to higher register pressure in these kernels. The rationale is that, in the modeled GPUs, SIMD8 kernels have access to all 128 registers while SIMD16 kernels have only 64 registers as SIMD16 instructions end up using a pair of registers for one operand.

The performance gains of BCC and SCC techniques for divergent applications are shown in Figure 10. Note that the reduction in EU cycles presented is above and beyond the existing Ivy Bridge optimizations. In 23 out of 29 applications (BFS, Hotspot, EigenValue, Ray Tracing, and most of the trace-based workloads) in our divergent sample set, SCC offers considerable gains beyond BCC alone and minimal gain for the remaining (LavaMD, Needleman-Wunsch and Particle Filter).

For several trace-based OpenCL workloads such as LuxMark, BulletPhysics, and RightWare, BCC and SCC provide benefits as high as 25%–42%. In those workloads, one quarter to one third of the additional benefit is attributed to SCC. Several other OpenCL kernels see benefits of 5%–25%. For the OpenGL benchmarks from the GLBench suite, we see gains of 15%-22% with the major portion associated with SCC. Face-detection workloads see benefits around 30% with the larger share of the benefits coming from SCC.

5.4 Execution Time Analysis

In this subsection, we discuss the performance benefit of our intra-warp divergence optimization techniques on kernel execution times considering the pipeline and memory system resource constraints, as well as any workload imbalance. Table 3 lists the model parameters used in the performance study. GPGenSim accurately models BCC and SCC cycle compaction time effects. We compare EU cycle reduction benefit with total reduction in kernel execution time and identify micro-architectural model parameters that may limit performance gains from the divergence optimizations.

Table 3: Microarchitecture Parameters

EU	6 EUs, 6 Threads per EU
SLM	64KB, 5 cycles
L3	128KB, 64-way, 4 banks, 7 cycles
LLC	2MB, 16-way, 8 banks, 10 cycles
L3 BW	1-2 accesses per cycle from data cluster to L3
Issue BW	2 instructions every 2 cycles

Due to reasons outlined in Section 5.1, our timing analysis is limited to the 14 divergent benchmarks running on GPGenSim. We study these benchmarks in two distinct sets here in order to focus on their specific behaviors. Note that our optimizations have no adverse impact on coherent applications because intra-warp compaction does not create any additional memory divergence. Our results show that some kernels see nearly all of the EU execution cycle compaction benefit reflected in their execution time savings with a data cluster to L3 cache bandwidth of two cache lines per cycle for six EUs ("DC2" case). A few other kernels see only a fraction of the EU cycle savings reflected in the execution time as they are memory latency constrained. One workload in particular does not see any benefit in execution time as its execution has significant workload imbalance across different EUs. The specific details are discussed below.

Reduction in total cycles (performance gains) provided by BCC and SCC techniques for Ray Tracing benchmarks is shown in Figure 11. In each set of the three stacked bars in the figure, the first two bars represent percentage reductions in execution time with peak data cluster bandwidth of one and two cache lines per cycle to/from L3 cache, respectively. The third bar in the set represents the percentage reduction in EU cycles for comparison purposes. The percentage reductions are shown on the primary Y-axis. For most benchmarks, the percentage reduction in execution time, with a data-cluster peak throughput of one cache line per cycle, is much smaller than the percentage reduction in EU cycles. Reduction in execution time is almost 90% of the EU cycle reduction benefit when the peak data cluster bandwidth is doubled to two cache lines per cycle. The significant performance difference in two cases can be explained by considering the secondary axis data, which captures that data cluster throughput demand. For most ray-tracing workloads, this demand is significantly over one cache line per cycle but never exceeds two cache lines per cycle.

The second group of benchmarks studied is divergent application set from Rodinia suite. We found that we can reduce EU cycles for these applications by an average of 18% and 21% with BCC and SCC, respectively. However, these cycle reductions do not lead to similar benefits in total execution time. Unlike Ray Tracing kernels, these benchmarks

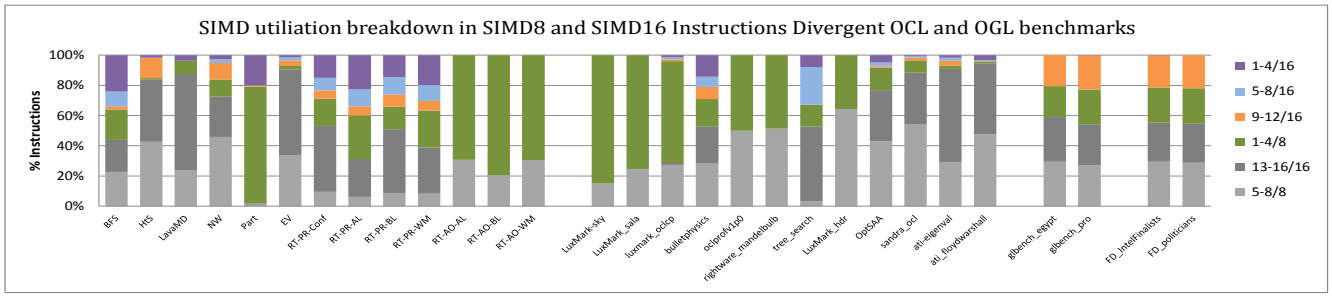


Figure 9: SIMD utilization breakdown in SIMD8 and SIMD16 instructions for divergent OpenCL applications.

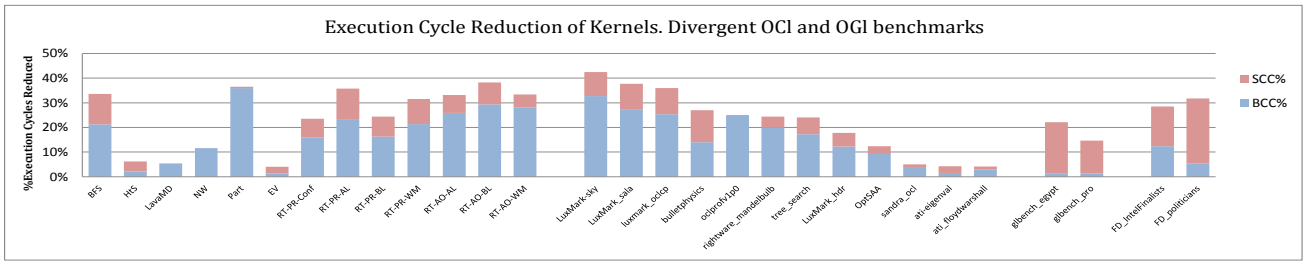


Figure 10: Execution cycle time reduction using BCC & SCC over and above existing Ivy Bridge optimization.

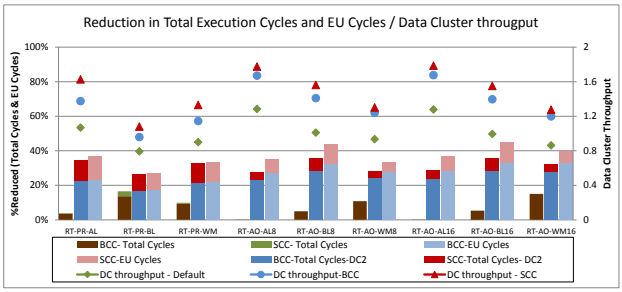


Figure 11: Reduction in total execution cycles and EU cycles in Ray Tracing kernels (primary axis) and data cluster throughput (secondary axis).

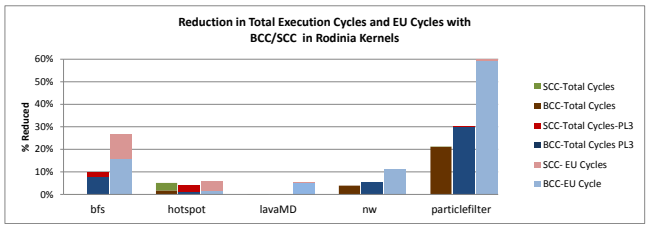


Figure 12: Reduction in total execution cycles and EU cycles with BCC/SCC in Rodinia kernels.

are generally not limited by data cluster bandwidth. Some of these benchmarks suffer from the long latency memory access times that cannot be hidden due to the low number of active threads and inherent memory divergence in some of the workloads. If memory stalls dominate the execution time as is the case for BFS, any optimization in EU cycles will not make a noticeable impact on the overall performance.

Figure 12 shows the percentage reduction in total cycles for a 128KB L3 cache and a perfect L3 model (infinite capacity) compared with percentage reduction in EU cycles. Our results show, while EU optimization provides performance gains for some applications (hotspot, NW, particle-filter), overall execution time saving is not as high as that of the EU cycle saving. On the extreme side, BFS and LavaMD do not see any benefit in execution time reduction. The perfect L3 model results show some performance improvement for BFS, which demonstrates its memory divergence challenges. On the other hand, LavaMD shows no benefit even from a perfect L3 cache.

In general, Rodinia benchmarks show less reduction in total execution time with BCC and SCC compared to Ray Tracing, despite noticeable saving in EU cycles. For these benchmarks, further performance improvements would require addressing memory system issues, better load-balancing and workload tuning, and removal of other sources of stalls.

Overall, the execution time analysis section shows limited additional benefit for SCC over BCC optimization, which is consistent with EU cycle reduction benefits observed earlier for this set of benchmarks. However, we wish to note that the trace-based divergent workloads indicate more promising benefits from SCC over BCC, and we expect to reduce execution time much better for these workloads.

In this paper, we have conducted a detailed study of SIMD divergence that shows significant performance benefit from intra-warp compaction optimizations for an Intel Ivy Bridge-like GPU model with SIMD width of 16. Other architectures such as NVIDIA and AMD GPUs have a SIMD width of 32 and 64 respectively. Our intuition and studies (not shown here) using the GPGPUSim [3] suggest a trend that SIMD efficiency of GPGPU applications reduces with wider SIMD widths. We are also not aware of any intra-warp compaction optimizations for these other GPU architectures (similar to what we have observed for the Ivy Bridge GPU). One can therefore expect a larger optimization opportunity and potential benefit from applying intra-warp compaction techniques to these other architectures.

6. RELATED WORK

Techniques for handling SIMD divergence were first proposed in the Illiac IV supercomputer using a single predicate register [6]. Chap GPU architecture introduced the concept of stack of predicate registers [22]. Modern approaches for addressing this problem are detailed below.

Inter-warp compaction techniques: Most prior techniques belong to the class of inter-warp compaction. Fung *et al.* first proposed dynamic warp formation (DWF) where enabled threads executing the same instruction (same program counter) are regrouped into new warps to improve SIMD efficiency [12]. DWF had an architectural option for "thread swizzling", *i.e.,* moving a thread from from its home lane into into a different one when forming newly compacted warps. However, this approach would have register access conflicts from multiple threads that have the same home lane in a combined warp, making it nonviable. SCC permutes operand lanes within a single warp in a conflict free manner. Other limitations of their approach including semantic correctness as well as increased memory divergence led to the subsequent thread block compaction (TBC) proposal [11] and the related large warp micro-architecture [25]. Rhu *et al.* improve on TBC by avoiding unnecessary synchronizations and unproductive compactions [30]. Overall, inter-warp compaction techniques are micro-architecturally complex, require lane-addressable registers, and can increase memory divergence.

Intra-warp compaction techniques: Our BCC and SCC proposals fall into this category of techniques. To our knowledge, the only other approach that uses intra-warp compaction is simultaneous branch interweaving (SBI) [5]. SBI attempts to execute "if" and "else"block instructions simultaneously and improve SIMD efficiency. It requires a higher micro-architectural complexity than our proposal as well as relies on thread-frontiers [9].

Software and hybrid techniques: Diamos *et al.* address SIMD inefficiency when threads execute unstructured control flow [9]. They propose "thread frontiers", which is a bounded region of the program containing all threads that have branched away from the current warp. A compiler can identify thread frontiers and different hardware schemes can be used to re-converge as threads enter the frontier. Han *et al.* proposed a compiler-based technique that reduces branch divergence through iteration delaying (reducing divergence in loop iterations) and branch distribution (reducing the amount of divergent code in branches by moving common code outside of the branching blocks) [14].

Memory divergence: Overall SIMD efficiency depends on both well-aligned control flow and memory accesses. Dynamic warp subdivision addresses branch and memory divergence from a different perspective [23]. It hides latency better by subdividing a convergent warp into warp splits, which can be individually regarded as an additional scheduling entity. Our work focuses on making compute divergence more efficient and is orthogonal to a memory divergence solution.

Vector Architectures: Vector architectures [32, 10, 21, 31, 2, 4] have a distinct programming model, execution model, and workload characteristics compared to GPGPU architectures. However, the intra-warp compaction techniques proposed in this paper are similar to density time optimizations for addressing vector control flow divergence.

Smith *et al.* study conditional operation support in a vector ISA and propose an implementation of masked operations that skips over blocks of false values [32]. However, for multiple parallel vector pipes, a BCC-like approach is proposed but not considered particularly advantageous. Instead, a more complex approach, in which each vector pipe is not in lockstep, is preferred and density time optimizations are applied for vector element masks that map to the same pipe. BCC closely resembles "element group skipping" density-time optimization across multiple vector pipes [2], where it is regarded as useful in a more limited context and its performance is not characterized. The performance analysis in this work shows that BCC is a useful optimization for several divergent GPGPU workloads. Vector fragment compression is the density time optimization implemented in the Maven Vector Thread architecture [4] and is applied within a vector pipe rather than across multiple vector pipes. The lane swizzling approach in SCC is a completely novel optimization approach for execution cycle compression.

7. CONCLUDING REMARKS

Existing SIMD architectures of GPUs benefit from wide number of threads executed in lockstep. However, GPU applications that exhibit control flow divergence suffer from reduced SIMD efficiency due to serialized execution of divergent paths. On the other hand, SIMD cores in GPUs from Intel, AMD, and NVIDIA all execute an instruction over multiple consecutive cycles using narrower ALUs than the instruction's SIMD width and we believe this gap will remain in future GPUs for the following reasons:

- A Narrower ALU width provides opportunities for improving divergent workload. A wide instruction width in GPUs (32 for NVIDIA and 64 for AMD GPUs) makes divergent code more inefficient without optimizations.
- 64-bit datatype support (double, int64) has been introduced for HPC and financial workload segments. This support is coming through wider datatypes taking longer latencies for an instruction rather than doubling ALU widths and datapath widths.
- Mobile GPUs need to support the same ISA, software APIs and features as GPUs in the client and high-performance segments. This is, however, an area, cost, and energy sensitive segment which would favor narrower hardware widths.

In this paper, we found that this gap between ALU width and instruction SIMD width provides us an interesting opportunity to squeeze out idle execution cycles for a subset of disabled SIMD lanes when no useful work is done. In this work, we have demonstrated how this opportunity can be exploited to provide increased SIMD efficiency using two relatively simple techniques for cycle compression, without significantly increasing micro-architectural complexity.

We have studied the performance benefit of these optimizations in the context of Intel's recent GPU architecture for a large number of divergent GPGPU and 3D Graphics applications. For the studied Intel architecture GPU model, we see up to 42% reduction in number of EU cycles using both BCC and SCC, with an average benefit of 20% benefit for divergent applications after accounting for an existing Ivy Bridge optimization inferred through micro-benchmarks. We believe that a higher cycle compression opportunity exists for divergent applications in other GPU architectures with larger SIMD widths as we expect more adverse impact of control flow divergence.

Maximum and average benefit for divergent workloads are summarized in Table 4. Here, the row labeled DC1 represents performance benefits with the more constrained memory bandwidths and smaller L3 and LLC cache of today, while the row labeled DC2 represents the expected performance for future GPUs with ample memory bandwidth.

Table 4: Summary of BCC and SCC Benefits

Divergent Workloads	BCC		SCC	
	max	avg	max	avg
GPGenSim (EU cycles)	36%	18%	38%	24%
Traces (EU cycles)	31%	12%	42%	18%
GPGenSim Execution time (DC1)	21%	5%	21%	7%
GPGenSim Execution time (DC2)	28%	12%	36%	18%

We believe that micro-architecture support for SIMD divergence optimization will enable a wider class of applications to be efficiently accelerated on future GPUs and remains an important area for future investigations.

8. ACKNOWLEDGMENTS

Help and support from Murali Sundaresan, Subramaniam Maiyuran, Jonathan Pearce, Ben Ashbaugh, Kipp Owens, Berna Adalier, Sven Woop, Warren Hunt, Ingo Wald, and Aaron Kunze is gratefully acknowledged. The authors also thank their shepherd, Brucek Khailany, and other anonymous reviewers for their help with improving this paper.

9. REFERENCES

[1] *AMD Radeon HD 7970 Graphics*, AMD. [Online]. Available: amd.com

[2] K. Asanovic, "Vector microprocessors," Ph.D. dissertation, UC Berkeley, 1998.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of International Symposium on Performance Analsys of Systems and Software*, 2009.

[4] C. F. Batten, "Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators," Ph.D. dissertation, MIT, 2010.

[5] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," in *Proceedings of International Symposium on Computer Architecture*, 2012, pp. 49–60.

[6] *ILLIAC IV – System Description*, Burroughs Corp, 1974, Computer History Museum resource.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of International Symposium on Workload Characterization*, 2009, pp. 44–54.

[8] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Proceedings of International Symposium on Workload Characterization*, 2010.

[9] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "SIMD re-convergence at thread frontiers," in *Proceedings of International Symposium on Microarchitecture*, 2011, pp. 477–488.

[10] R. Espasa and M. Valero, "Multithreaded vector architectures," in *International Symposium on High Performance Computer Architecture*, 1997, pp. 237 –248.

[11] W. Fung and T. Aamodt, "Thread block compaction for efficient simt control flow," in *International Symposium on High Performance Computer Architecture*, 2011, pp. 25–36.

[12] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of International Symposium on Microarchitecture*, 2007, pp. 407–420.

[13] V. George and H. Jiang, "Intel next generation microarchitecture code name IvyBridge," in *Intel Developer Forum*, 2012, Technology Insight Video.

[14] T. Han and T. Abdelrahman, "Reducing branch divergence in GPU programs," in *Workshop on General Purpose Processing on GPU*, 2011, p. 3.

[15] W. Hwu, Ed., *GPU Computing Gems — Jade and Emerald Eds.* Morgan Kaufmann, 2011.

[16] *DirectX Developer's Guide for Intel Processor Graphics: Maximizing Performance on the New Intel Microarchitecture Codenamed IvyBridge*, Intel Corp, April 2012. [Online]. Available: software.intel.com

[17] *Intel Open Source HD Graphics Programmer's Reference Manual (PRM) for 2012 Intel Core Processor Family (codenamed IvyBridge)*, Intel Corp, 2012. [Online]. Available: intellinuxgraphics.org

[18] *Intel SDK for OpenCL Applications 2012: OpenCL Optimization Guide*, Intel Corp, 2012. [Online]. Available: software.intel.com

[19] D. Kanter, "Intel's IvyBridge graphics architecture." [Online]. Available: realworldtech.com/ivy-bridge-gpu/

[20] *OpenCL - The open standard for parallel programming of heterogeneous systems*, The Khronos Group. [Online]. Available: khronos.org/opencl/

[21] 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 *Proceedings of International Symposium on Computer Architecture*, 2011, pp. 129–140.

[22] A. Levinthal and T. Porter, "Chap-a simd graphics processor," in *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, 1984, pp. 77–82.

[23] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of International Symposium on Computer Architecture*, 2010, pp. 235–246.

[24] *Compute Shader Overview*, Microsoft Corp. [Online]. Available: msdn.microsoft.com/en-us/library/ff476331.aspx

[25] V. Narasiman, M. Shebanow, C. Lee, R. Miftakhutdinov, O. Mutlu, and Y. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of International Symposium on Microarchitecture*, 2011, pp. 308–317.

[26] *Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview*, Nvidia Corp, November 2006. [Online]. Available: nvidia.com

[27] *NVIDIA CUDA C Programming Guide: Version 4.2*, Nvidia Corp, April 2012. [Online]. Available: nvidia.com

[28] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, Nvidia Corp, 2012. [Online]. Available: nvidia.com

[29] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of of IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[30] M. Rhu and M. Erez, "CAPRI: prediction of compaction-adequacy for handling control-divergence in GPGPU architectures," in *Proceedings of International Symposium on Computer Architecture*, 2012, pp. 61–71.

[31] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis, "Vector lane threading," in *Proceedings of International Conference on Parallel Processing*, 2006, pp. 55–64.

[32] J. E. Smith, S. G. Faanes, and R. Sugumar, "Vector instruction set support for conditional operations," in *Proceedings of International Symposium on Computer Architecture*, 2000, pp. 260–269.

[33] I. Wald, "Active thread compaction for GPU path tracing," in *Proceedings of ACM SIGGRAPH Symposium on High Performance Graphics*, 2011, pp. 51–58.

[34] D. Woligroski, "AMD A10-4600M review: Mobile trinity gets tested," Tom's Hardware, May 2012. [Online]. Available: tomshardware.com