Performance Debugging of GPGPU Applications with the Divergence Map *

Bruno Coutinho Diogo Sampaio Fernando M. Q. Pereira Wagner Meira Jr.
Departamento de Ciência da Computação Universidade Federal de Minas Gerais, Brazil
{coutinho,fpereira,meira}@dcc.ufmg.br, dnsampaio@gmail.com

June 7, 2010

Abstract

The increasing programability and the high computational power of Graphical Processing Units (GPU) make them attractive to general purpose programming. However, taking full benefit of this execution environment is a challenging task. One of these challenges stem from divergences, a phenomenon that occurs when threads that execute in lock-step are forced to take different program paths due to branches in the code. In face of divergences, some threads will have to wait, idly, while their diverging siblings execute. Optimizing the code to avoid divergences is difficult, because this task demands a deep understanding of programs that might be large and convoluted. In order to facilitate the detection of divergences, this paper introduces the divergence map, a data structure that indicates the location and the volume of divergences in a program. We build this map via dynamic profiling techniques, which we have implemented on top of an open source CUDA compiler. To illustrate the importance of the divergence map, we have used it to pin-point the core regions that must be optimized in well known public applications. By hand optimizing some applications, we have added 9-11% speedups onto kernels that have already gone through the sieve of many programmers.

1 Introduction

Increasing programmability and low hardware cost are boosting the use of graphical processing units (GPU) to run general purpose applications. Illustrative examples of this new trend are the rising popularity of CUDA[1], Stream SDK [2] and OpenCL[3]. Running general purpose programs in GPUs is attractive because these processors are massively parallel. For instance, the Nvidia's GeForce GTX 285 series has 240 processing units and 30,720 hardware threads. Such a hardware has allowed GPU-based applications to run over 400x faster than equivalent CPU based programs [1]. This trend is likely to continue, as upcoming hardware more closely integrates GPUs and CPUs [2], and new models of heterogeneous hardware are introduced [3].

GPUs are highly parallel; however, due to its restrictive programming model, not every application can benefit from all their processing power. In these processors, threads are organized in groups that execute in lock-step. Such groups are called *warps* in the Nvidia jargon [1], or *wavefronts* in ATI's [2]. To better understand the rules that govern threads in the same warp, we can imagine that each warp might use a number of processing units, but has only one instruction fetcher. As an example, the GeForce GTX 285 GPU is able to run 30 warps at the same time; each warp consists of 32 threads, and uses 8 processing units. Thus, each warp might perform 32 instances of the same instruction in four cycles of the hardware pipeline. Regular applications, such as scalar vector multiplication, fare very well in GPUs, as we have the same operation being independently performed on different chunks of data. However, not every application is so regular, and divergences may happen.

Divergences happen when threads inside the same warp follow different paths after processing the same branch. The branching condition might be true to some threads, and false to others. Given that each

*This work is partially supported by CNPq, CAPES, Finep and Fapemig.

[1]See *The CUDA Programming Guide, 1.1.1*
[2]See *ATI CTM Guide*
[3]See *The OpenCL Specification, 1.0*

warp has access to only one instruction at each time, in face of a divergence, some threads will have to wait, idly, while others execute. Hence, divergences may be a major source of performance degradation. As an example, Baghsorkhi *et al.* [4] have analytically found that approximately one third of the execution time of the prefix scan benchmark [5], included in the CUDA software development kit (SDK), is lost due to divergences. Optimizing an application to avoid divergences is problematic for a number of reasons. First, some parallel algorithms are inherently divergent; thus, threads will naturally disagree on the outcome of branches. Second, finding highly divergent branches burdens the application developer with a tedious task, which requires a deep understanding of code that might be large and convoluted.

In this paper we introduce the *Divergence Map*, a data structure that indicates the location and the volume of divergences. We build the divergence map via dynamic profiling of the GPU program, or kernel, as it is normally called. The divergence map indicates how many warps have visited each kernel branch, and how many divergences have happened per branch. This data structure has two main purposes. First, it shows to the application developer the program points that cause performance degradation due to diverging execution paths. Second, the divergence map provides useful feedback to self-adjusting compiler optimizations, which rely on profiling information to decide the best ways to improve a program.

We have implemented a dynamic profiler as a patch to the Ocelot open source [6] CUDA compiler. We have produced divergence maps to many benchmarks freely available in the Internet, and we have used this information to manually optimize two of these applications. As an example, by changing only 2.3% of the code of a well-known implementation of parallel quicksort [7] we got speedups of up to 9.2%. This number may seem small, but this gain applies onto a highly optimized application, that has gone through three years of public scrutiny. All the code and an extensive description of these experiments are available in our webpage [4]. Although we have implemented our tool on a CUDA compiler, the techniques introduced in this paper fit any architecture that follows the *single-instruction multiple-data* (SIMD) execution model that characterizes GPU's warps. Our profiler also works with predicated instructions, which, like ordinary branches, also suffer from divergencies.

[4] `http://www2.dcc.ufmg.br/laboratorios/llp/wiki/doku.php?id=coutinho`

2 Related Work

Profilers are old allies of compiler designers. Since Graham *et al.*'s highly influential `gprof` [8], many profiling tools and techniques have been described. Profiling has been initially used to measure the dynamic behavior of sequential applications [8, 9, 10], but subsequent works have extended profiling into the realm of parallel computers with great success [11, 12, 13, 14, 15]. Parallel profilers, such as the recent works of Tallent *et al.* [14] or Eyerman and Eeckhout [11] generally focus on systems where threads are able to execute independently of each other. Thus, profiling applications that fit the SIMD model, used in GPU's warps, is still a challenge.

The development of general purpose applications on GPUs has received a substantial amount of attention recently; however, there are few works dealing directly with divergences in SIMD architectures. Fung *et al.* [16] determined the best program regions to re-converge divergent threads, and the hardware necessary to perform this task. Kerr *et al.* [6] have evaluated, via an emulator, the impact of divergences on the performance of CUDA applications. Our objectives are similar to Kerr *et al.*'s, but we do profiling instead of simulation. Simulation has the advantage of using a controlled environment to obtain data; however, it has problems of its own, i.e simulation accuracy. Baghsorkhi *et al.* [4] have described an analytical model that estimates the cost of divergences; however, they must assume that a given branch is divergent, a fact that we check via profiling.

We know two GPU profilers. On the academic side, Boyer *et al.* [17] have described a profiler that supports debugging; however, it does not handle divergences in any special way. On the industrial side, Nvidia has released the CUDA Visual Profiler, or CVP [5]. CVP lets the CUDA developer to probe several aspects of the kernel behavior, including the existence of divergent threads in the parallel program. Regarding the measurement of divergencies, our tool is more precise than CVP. CVP tells the developer how many divergences have occurred during the kernel execution, probably, as we speculate, by reading some counter implemented in hardware. However, it does not point the place or the volume in which the divergences happened – exactly the information that we provide, and that we believe is the most useful to guide programmers in optimizing their applications.

[5] `http://forums.nvidia.com/index.php?showtopic=57443`

```
__global__ static void bitonicSort(int * values) {
  extern __shared__ int shared[];
  const unsigned int tid = threadIdx.x;
  shared[tid] = values[tid];
  __syncthreads();
  for (unsigned int k = 2; k <= NUM; k *= 2) {
    for (unsigned int j = k / 2; j>0; j /= 2) {
      unsigned int ixj = tid ^ j;
      if (ixj > tid) {
        if ((tid & k) == 0) {          7,329,816 / 28,574,321
          if (shared[tid] > shared[ixj]) {
15,403,445 / 20,490,780   swap(shared[tid], shared[ixj]);
          }
        } else {
          if (shared[tid] < shared[ixj]) {
4,651,153 / 8,083,541     swap(shared[tid], shared[ixj]);
          }
        }
      }
      __syncthreads();
    }
  }
  values[tid] = shared[tid];
}
```

Figure 1: The implementation of bitonic sort. This code is freely available in the CUDA SDK.

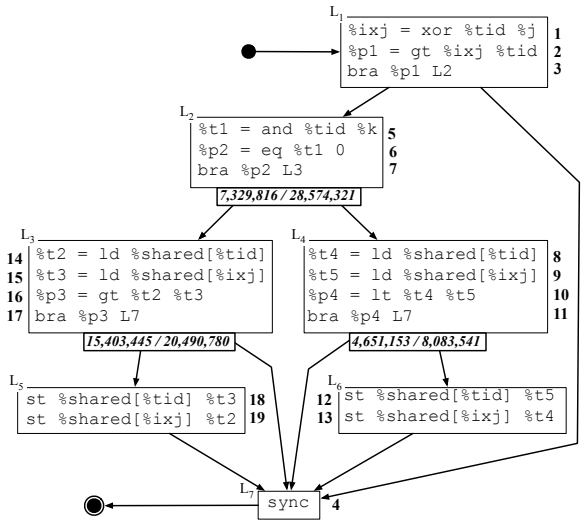

Figure 2: The (simplified) PTX representation of part of the bitonic kernel.

3 Divergences

We will illustrate the concept of divergence showing how this phenomenon occurs in an implementation of the bitonic sorting algorithm [18] [6]. The code, which we took from Cederman *et al.* [7], is given in Figure 1; we will focus on the boxed part. We have labeled three interesting conditional branches with the entries in the divergence map that we have obtained via the precise profiling strategy described in Section 5. The *divergence map* is a function that maps branches to integer pairs. The first integer denotes the number of divergences, and the second denotes the number of times the block was visited by a warp.

Figure 2 shows the simplified PTX representation of this kernel. PTX is the high-level instruction set used to represent CUDA kernels [7]. We have augmented this control flow graph with numbers denoting the addresses of each instruction in the kernel code. Additionally, we have added to the figure the divergence map entries after the three conditionals that have been outlined in Figure 1.

The $O(n \ln^2 n)$ complexity penalizes the bitonic sort method when we are restricted to a sequential hardware; nevertheless, the ability to perform $n/2$ comparisons, where n is the array size, in parallel, makes this algorithm attractive to the GPU execution environment. However, we must remember that we are tied to our SIMD execution model. Bringing back the metaphor of Section 1: we have four functional units in this example, but only one instruction fetcher. Invariably the condition `(tid & k) == 0`, where `tid` is the thread identifier, will be true to some threads, and false to others. In this case, we will have a divergence.

Figure 3 shows an execution trace in a setting where we have a single warp with four threads, e.g. $w = \{t_0, t_1, t_2, t_3\}$. The *execution trace* is a sequence of cycles. In each cycle an instruction ι, present in the kernel, will be fetched. In face of divergences, some threads will not process any instruction. We indicate this fact by giving these threads the symbol • in the idle cycles. We assume that `values`, the input array, is $\{4, 3, 2, 1\}$, and that the id of thread t_n is $n, 0 \leq n \leq 3$. When $k = 2$ and $j = 1$, the input array causes two divergences. The first split happens at cycle $i = 3$, due to the branch `bra %p1, L2`, and it separates t_0 and t_2 from t_1 and t_3. This divergence happens because the condition `%ixj > %tid` is true only for t_0 and t_2. The second split happens at cycle $i = 6$, due to the branch `bra %p2, L3`, and it separates threads t_0 and t_2.

After the first iteration of the outer loop in Fig-

[6] `http://www.cs.chalmers.se/~dcs/gpuqsortdcs.html`
[7] See *PTX: Parallel Thread Execution, ISA v.1.3*

cycle	opcode	t_0	t_1	t_2	t_3
1	`xor`	ι_1	ι_1	ι_1	ι_1
2	`gt`	ι_2	ι_2	ι_2	ι_2
3	`bra`	ι_3	ι_3	ι_3	ι_3
4	`and`	ι_5	•	ι_5	•
5	`eq`	ι_6	•	ι_6	•
6	`bra`	ι_7	•	ι_7	•
7	`load`	ι_{14}	•	•	•
8	`load`	ι_{15}	•	•	•
9	`gt`	ι_{16}	•	•	•
10	`bra`	ι_{17}	•	•	•
11	`load`	•	•	ι_8	•
12	`load`	•	•	ι_9	•
13	`lt`	•	•	ι_{10}	•
14	`bra`	•	•	ι_{11}	•
15	`store`	ι_{18}	•	•	•
16	`store`	ι_{19}	•	•	•
17	`store`	•	•	ι_{12}	•
18	`store`	•	•	ι_{13}	•
19	`sync`	ι_4	ι_4	ι_4	ι_4

Figure 3: A snapshot of the execution trace over of the program in Figure 2. Each thread trace is given as a ordered sequence $[\iota_1, \ldots, \iota_n]$, where each ι_i is an instruction in the kernel, or a • indicating that the thread did not execute in that cycle.

ure 1, when $k = 2$, and $j = 1$, we have the following non-null entries in the divergence map: $\iota_3 \mapsto (1,1), \iota_7 \mapsto (1,1)$. After the second iteration, when $k = 4$, and $j = 2$, we observe that new divergences have happened, leading to $\iota_3 \mapsto (2,2), \iota_7 \mapsto (1,1)$. This last instance of the divergence map indicates that the instruction ι_3, e.g.: `bra %p1, L2` has been visited by two warps, and during each of these two visits a divergence took place.

4 Optimizing Bitonic Sort

In this section we show how we have used information from the divergence map to improve the performance of the implementation of Bitonic Sort seen in the previous section. The implementation of quicksort of Cederman *et al.* [7] uses the traditional quicksort to partition a large array of numbers into small chunks of data, which are then sorted via the algorithm shown in Figure 1. The speedup numbers that we give in this section refer to the whole quicksort algorithm, even though we only change the bitonic sort kernel. That is, we deliver 6-10% performance speed up by changing less than 10-12 assembly instructions in a program containing 895 instructions! Figure 1 shows that the suite of conditionals nested into the double loop suffers from a large number of divergences. Specifically, the condition `(tid & k) == 0` is traversed by warps 28 million times, and one third of these visits diverge. The map also shows that divergences are common in both the sub-clauses of this branch, namely `shared[tid] > shared[ixj]` and `shared[tid] < shared[ixj]`.

In order to improve the kernel, we start by noticing that the code trace formed by block L_3 followed by block L_5 is very similar to the trace $L_4 + L_6$. The only difference comes from the conditionals in program points 9 and 13. Armed with this observation, we bind the blocks L_5 and L_6 together, using the index manipulation trick that gives us the program in Figure 4 (a). Divergences might still happen; however, whereas the maximum divergent path in Figure 2 contains eight instructions, the worst divergence case in Figure 4 (b) contains six instructions. This optimization gives a speed-up of 6.75%.

The code in Figure 4 (b) still provides us with optimization opportunities. We can use the ternary selectors available in CUDA to merge basic blocks L_3 and L_4, thus removing the divergence at the end of block L_2. The modified source is shown in Figure 4 (c), and the equivalent PTX code is given in Figure 4 (d). An instruction such as `%a = sel %tid %ixj %p` assigns `%tid` to `%a` if `%p` is not zero. It assigns `%ixj` to `%a`, otherwise. In the new program, the worst case divergent path has only two instructions. This final optimization gives us a speed-up of 9.2%.

GPUs ask for new optimization techniques We have observed that the optimization techniques that improve the efficiency of CUDA kernels might not apply to sequential programs. For instance, the kernel in Figure 4 (d) has a path with 11 instructions, which is executed every time the branching condition evaluates to true. On the other hand, the equivalent code sequence, in Figure 2 contains nine instructions. In order to measure the relative performance between all these kernels in a sequential machine, we have compiled them to x86, using `gcc -O0` to preserve the original sequences of instructions, and we have run each program on a Intel x86 board, with 2G of RAM and 1.2MHz of clock. Running one iteration of each binary on a four million elements array, we observe that the original, non-optimized PTX code from Figure 2 is 19% faster than the program in Figure 4 (b), and 11% faster than the program in Figure 4 (d), the opposite of what we would find in CUDA!

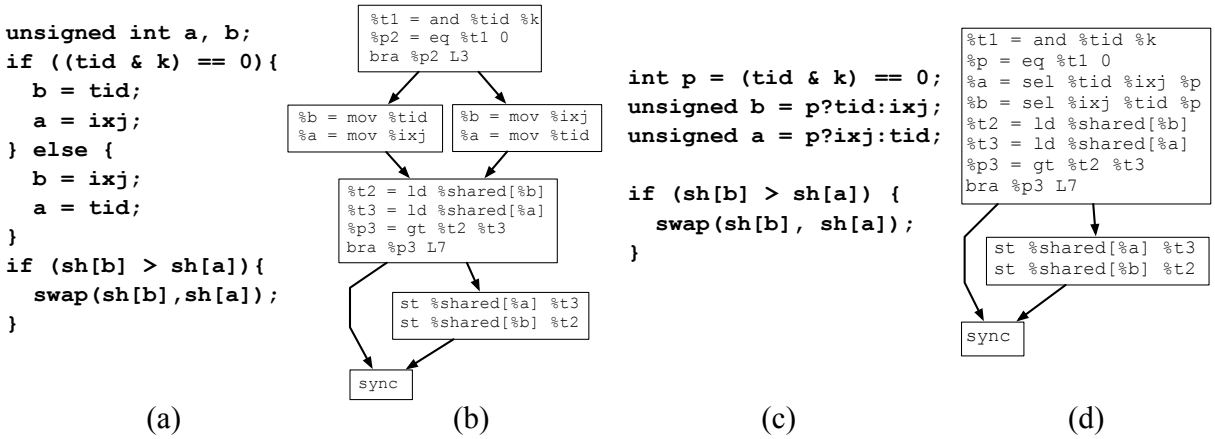

Figure 4: (a-b) Source code and PTX code after first optimization. (c-d) Source code and PTX code after second optimization.

5 Profiling via Instrumentation

We have implemented the divergence map as two arrays of integers, that we call τ and δ, such that $\tau[b]$ stores the number of warps that have visited basic block b, and $\delta[b]$ stores the number of divergences that took place at b. We insert the necessary instrumentation automatically in a three-phase process:

1. **Initialization**: when we reserve memory on the GPU address space to store our arrays, and initialize these arrays with zero's.

2. **Measurement**: when we compute the number of visits and divergences, storing the results in our arrays.

3. **Reading**: when we copy to the CPU the data accumulated during the profiling of the kernel program, in the GPU.

Because the initialization and reading phases are trivial, we will only describe the measurement phase.

We detect branches via program instrumentation. Currently we insert the code in Figure 5 at each branch. This figure shows the instrumentation of block L_2 of the example in Figure 2. We split each instrumented basic block b into three new blocks: b_{up}, b_{bottom} and $b_{find\ writer}$. Notice that not every branch is divergent, and we are improving our profiler to avoid instrumenting non-divergent branches. The code that performs the instrumentation executes two tasks: (i) in blocks b_{up} and $b_{find\ writer}$ we find a thread – *the writer* – to update the arrays δ e τ. (ii) in block b_{bottom} we detect and report the divergence.

We let the writer to be the thread with the lowest identifier among the active threads in the warp. In Figure 5 the variable `%laneid` denotes the thread identifier inside the warp. We cannot simply choose as the writer the thread with `%laneid = 0`, because this thread may be idle due to a previous divergence. Thus, in blocks b_{up} and $b_{find\ writer}$ we loop through the live threads, looking for the one with lowest `%laneid`.

Once we have a suitable writer, we perform the detection of divergences via voting. The PTX instruction `%p = vote.uni.pred, %q` sets `%p` to true if all the threads in the warp find the same value for the predicate `q`. Thus, we vote on the predicate that controls the outcome of a potentially divergent branch. If every warp thread agree on the predicate, no divergence happens at that particular moment of the program execution; otherwise, a divergence takes place, and it is necessary to increment the δ array. The instruction `@iAmWriter %`$\delta[L_2]$ `= atom.add %`$\delta[L_2]$ `1`, in the ninth program point in Figure 5, adds one to $\delta[L_2]$ if, and only if, the predicate `@iAmWriter` is true.

The instrumentation requires the serialization of warp threads at branching points, because the updating of the divergence map is performed via atomic operations. Thus, the worst case complexity cost of the instrumentation, per branch, is proportional to the number of warps in the running kernel times the number of threads per warp. However, given that nested

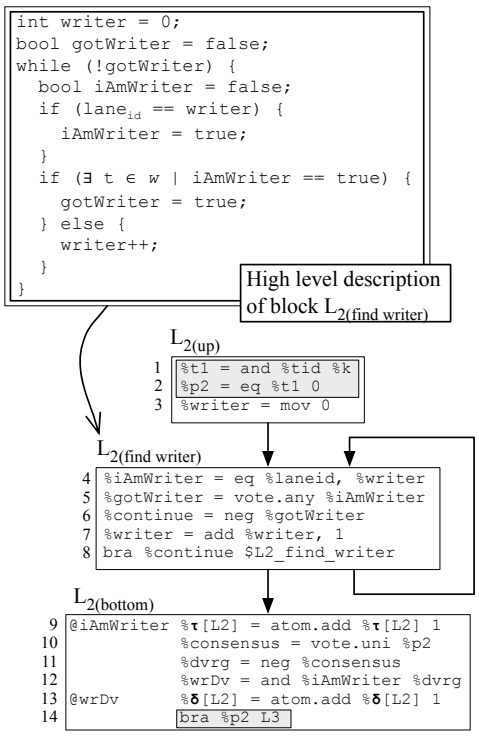

Figure 5: The code inserted to instrument block L_2 in our example. Above we have a higher level description of the instructions in block $L_{2(find\ writer)}$. Code from the original program is marked in gray.

divergences are rare, the thread with lowest identifier in the warp will be often present among the active threads. Thus, the loop in block $L_{2(find\ writer)}$, in Figure 5, tends to find a valid writer after the first iteration. We give an empirical evaluation of the instrumentation cost in Section 6.

6 Experimental Results

We have evaluated experimentally 11 publicly available CUDA benchmarks, instrumenting a total of 25 kernels, and finding divergences in 19. In order to guarantee reproducibility, we have also made our compiler, and all the scripts used in these experiments available in our webpage. We run these experiments on a Nvidia GeForce GTX 260 powered video card.

The Benchmarks: These experiments use the following benchmarks, which, due to space constraints we will refer by two letters only: Quicksort (`qs`), is the program that gave us the bitonic sort kernel used throughout the paper. Scan of Large Arrays (`sc`), is NVIDIA's implementation of parallel prefix sum [8]. Cudaseg (`cs`) does segmentation of biomedical images [9]. The rest of our benchmarks comes from the Rodinia Benchmark Suite [19]: Back Propagation (`bp`), BFS (`bf`), CFD (`cf`), HeartWall (`hw`), Hotspot (`hp`), Needleman-Wunsch (`nw`), SRAD (`sr`) and Stream Cluster (`st`). In this section we will only show results for kernels that had at least one divergent branch; six kernels in our benchmark suite did not show divergences, and we omit them. All these benchmarks are actual applications, publicly available in their author's webpages, that use kernels with meaningful sizes – Table 2 shows the size of each kernel. Because the benchmarks might contain more than one kernel, we discriminate them with the first and last letter of their names.

Divergences in general purpose applications: Table 1 shows the number of divergences in our benchmarks. We see that a large proportion of the conditional branches present in typical CUDA applications – between 50% and 60% – give origin to divergences. The figure also isolates the program points with the largest number of divergences per kernel. From these figures we see that some program points suffer from a non-neglible number of divergences. Interesting is the fact that these benchmarks have gone through the eye of many developers, and are extremely optimized; yet, divergences still are very common. Table 1 shows that the most visited branches tend to be the most divergent, although there are exceptions, like Stream Cluster. Therefore, traditional profiling techniques, such as those employed on `gprof` [8], can easily pin-point the most executed program regions, but these might not necessarily be the biggest sources of divergences.

Probing the instrumentation overhead: Table 2 gives some figures about the overhead imposed by our instrumentation. We see that our instrumentation approach adds a substantial overhead to the target kernel. In terms of code size, the instrumented program tends to grow between 2 and 3 times. The overhead is even bigger in terms of time, multiplying the execution time of applications by a factor of up to 1,500 in the parallel prefix scan (`sc.pn`). Nevertheless, these numbers are similar to other instrumentation based profiling strategies [20]. More important: our instrumentation does not change the semantics

[8] `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html`

[9] `http://code.google.com/p/cudaseg/`

Kernel	BB	CC	DB	MD / NV	MV
bf.Kl	8	5	3	126K/375K	375K
bf.K2	4	2	1	126K/375K	375K
bp.ld	10	5	3	61K/131K	131K
bp.as	2	1	1	1/32K	32K
cs.ui	44	16	11	47M/366M	366M
cf.cx	26	12	2	9.1M/18.2M	18.2M
hp.cp	34	17	9	15K/15K	15K
hw.kl	221	131	62	76M/144M	144M
nw.n1	17	7	5	2M/2.1M	2.1M
nw.n2	17	7	5	2M/2.1M	2.1M
qs.Lt	101	58	34	25M/32M	32M
qs.P1	32	16	8	2M/2M	2M
qs.P2	14	8	4	2M/2M	2M
qs.P3	5	3	1	1/15	57
sc.pn	16	8	3	1M/14M	14M
sc.ud	3	1	1	197K/1.5M	1.5M
sr.s1	32	13	8	9.2M/9.2M	9.2M
sr.s2	13	5	3	9.2M/9.2M	9.2M
st.pl	14	6	2	2.6M/3.3M	844M

Table 1: **The occurrence of divergences in general purpose benchmarks.** BB: number of basic blocks; CC: number of conditional branches; DB: number of branches that cause divergences; MD: maximum number of run-time divergences; NV: number of times that MD branch is visited; MV: maximum number of times any branch is visited.

Kernel	LoC	OSz	ISz	TmO	OvI (%)	TND
bf.Kl	18	57	136	1673	3,727	1M
bf.K2	13	28	68	1326	10,734	126K
bp.ld	33	102	178	1K	42,857	126K
bp.as	55	94	121	1K	643	1
cs.ui	103	393	615	79M	545	184M
cf.cx	127	1039	1209	125K	100	9.3M
hp.cp	112	236	471	409	54,321	45K
hw.kl	1327	1675	3392	418K	14,007	78M
nw.n1	80	274	379	79.1K	3,712	5.2M
nw.n2	83	277	382	78K	3,780	5.6M
qs.Lt	270	568	1336	2M	939	72M
qs.P1	111	231	453	133K	886	2M
qs.P2	66	90	208	227K	765	2M
qs.P3	27	43	96	78	177	1
sc.pn	24	138	256	64K	153,411	2M
sc.ud	19	40	67	20K	23,179	197K
sr.s1	154	285	468	8K	4,061	721K
sr.s2	99	133	212	4.6K	5,461	328K
st.pl	44	107	199	1.8M	12,687	2.9M

Table 2: **Checking the instrumentation overhead.** LoC: number of lines of CUDA code in original kernel; OSz: kernel size, in number of PTX instructions; ISz: instrumented program size (PTX); TmO: time taken by original program (μs); OvI: time overhead imposed by instrumentation (%); TND: total number of divergences found.

of the program, as it does not use any of the program variables. Hence, by observing divergences we do not change the pattern of divergences in the program.

Most divergences occur in few branches: We found an interesting property in the analyzed benchmarks: divergences are very concentrated. Table 3 illustrates this pattern inherent to all benchmarks. We notice that nine kernels have more than 90% of the divergences in one branch, and only one kernel – cudaseg – contains more than six highly divergent branches. This pattern of concentration seems to point that the developer should focus on a few program regions in order to optimize the effect of divergences on SIMD applications.

Optimizing SRAD: In addition to bitonic sort (Section 4), we have used the divergence map to optimize Rodinia's SRAD. We got 11.5% of time speed up by merging the contents of different branches. We have reported these gains to the authors of Rodinia, and have been told that our optimized kernel will be part of the next edition of this benchmark suite. We speculate that it is possible to optimize the other applications too, but, as we are not familiar with the source code, we have not tried it.

Kernel	Hot	BR	Kernel	Hot	BR
bf.Kl	3	0.60	qs.Lt	5	0.09
bf.K2	1	0.50	qs.P1	1	0.06
bp.ld	3	0.60	qs.P2	1	0.12
bp.as	1	0.10	qs.P3	1	0.33
cs.ui	9	0.56	sc.pn	2	0.25
cf.cx	1	0.08	sc.ud	1	1.00
hp.cp	4	0.24	sr.s1	6	0.47
hw.kl	1	0.10	sr.s2	2	0.40
nw.n1	4	0.57	st.pl	1	0.17
nw.n2	4	0.57			

Table 3: **Concentration of divergences.** Hot: number of branches responsible for more than 90% of divergences. BR: hot branches divided by total number of branches in the kernel.

7 Conclusion

This paper has introduced the divergence map, a concept that helps application developers to implement more efficient programs on single-program-multiple-data execution environments, such as the CUDA architecture. The divergence map highlights the program regions that cause thread divergences, and thus degrade performance. We build divergence maps via

dynamic profiling. Our experiments have shown that the divergence map is an important tool, as traditional profilers would not be able to correctly point the program locations that account for the majority of thread divergences. Additionally, we have showed how the divergence map has helped us to manually optimize well-known CUDA applications. All the software described in this paper, in particular our version of the Ocelot compiler that contains our profiler, is publicly available [10].

References

[1] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *PPoPP*. ACM, 2008, pp. 73–82.

[2] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *TOG*, vol. 27, no. 3, pp. 1–15, 2008.

[3] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, "Programming model for a heterogeneous x86 platform," in *PLDI*. ACM, 2009, pp. 431–440.

[4] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *PPoPP*. ACM, 2010, pp. 105–114.

[5] M. Harris, "The parallel prefix sum (scan) with CUDA," NVIDIA, Tech. Rep. –, 2008.

[6] A. Kerr, G. F. Diamos, and S. Yalamanchili, "A characterization and analysis of PTX kernels," in *IISWC*. IEEE, 2009, pp. 3–12.

[7] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *Journal of Experimental Algorithmics*, vol. 14, pp. 1.4–1.24, 2009.

[8] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler (with retrospective)," in *PLDI*, 1982, pp. 49–57.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, 2002.

[10] A. Srivastava and A. Eustace, "Atom: a system for building customized program analysis tools," in *PLDI*. ACM, 1994, pp. 196–205.

[11] S. Eyerman and L. Eeckhout, "Per-thread cycle accounting in smt processors," *ASPLOS*, vol. 44, no. 3, pp. 133–144, 2009.

[12] M. Ji, E. W. Felten, and K. Li, "Performance measurements for multithreaded programs," in *SIGMETRICS*. ACM, 1998, pp. 161–170.

[13] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, pp. 37–46, 1995.

[14] N. R. Tallent and J. M. Mellor-Crummey, "Effective performance measurement and analysis of multithreaded applications," *PPOPP*, vol. 44, no. 4, pp. 229–240, 2009.

[15] Z. Xu, B. P. Miller, and O. Naim, "Dynamic instrumentation of threaded applications," in *PPoPP*. ACM, 1999, pp. 49–59.

[16] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO*. IEEE, 2007, pp. 407–420.

[17] M. Boyer, K. Skadron, and W. Weimer, "Automated dynamic analysis of CUDA program," in *STMCS*, 2008.

[18] K. E. Batcher, "Sorting networks and their applications," in *AFIPS*. ACM, 1968, pp. 307–314.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*. IEEE, 2009, pp. 44–54.

[20] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of java profilers," in *PLDI*. ACM, 2010, p. XXXX.

[10] `http://www2.dcc.ufmg.br/laboratorios/llp/wiki/doku.php?id=coutinho`