Exploiting Multi-Loop Parallelism on Heterogeneous Microprocessors

Michael Zuzak and Donald Yeung

Department of Electrical and Computer Engineering
University of Maryland at College Park
{mzuzak,yeung}@umd.edu

Abstract. Heterogeneous microprocessors integrate CPUs and GPUs on the same chip, providing fast CPU-GPU communication and enabling cores to compute on data "in place." These advantages will permit integrated GPUs to exploit a smaller unit of parallelism. But one challenge will be exposing sufficient parallelism to keep all of the on-chip compute resources fully utilized. In this paper, we argue that integrated CPU-GPU chips should exploit parallelism from multiple loops simultaneously. One example of this is nested parallelism in which one or more inner SIMD loops are nested underneath a parallel outer (non-SIMD) loop. By scheduling the parallel outer loop on multiple CPU cores, multiple dynamic instances of the inner SIMD loops can be scheduled on the GPU cores. This boosts GPU utilization and parallelizes the non-SIMD code. Our preliminary results show exploiting such multi-loop parallelism provides a 3.12x performance gain over exploiting parallelism from individual loops one at a time.

1 Introduction

Traditionally, GPUs have been implemented as discrete chips on daughter cards, but recently, processor manufacturers have been producing ***heterogeneous microprocessors*** in which the CPU and GPU are integrated on the same die [1, 2, 4]. Such heterogeneous chips provide a single image of physical memory to both the CPU and GPU cores, resulting in a seamless shared address space. This allows all cores to compute on data "in place," eliminating copying between separate CPU and GPU memories. In addition, the tight integration permits extremely fast CPU-GPU communication–*e.g.*, through off-chip physical memory.

The vastly improved CPU-GPU communication speeds, along with not having to copy data, imply that integrated GPUs can exploit a much smaller amount of parallelism compared to discrete GPUs which require massive parallelism to amortize their large startup latencies [5]. This will allow integrated GPUs to accelerate a wide range of loops. Furthermore, the support for shared memory

This research was supported in part by National Science Foundation grant #CCF-1618963.

between CPU and GPU will also greatly simplify programming–*e.g.*, programmers will be spared the onerous task of having to identify what data to communicate. Together, flexibility to off-load finer-grained loops coupled with reduced programming effort will enable programmers to map more complex programs onto integrated GPUs.

As researchers try to accelerate more complex programs, a major challenge will be parallelizing codes to keep heterogeneous microprocessors fully utilized. The conventional approach is to parallelize loops one at a time, and to schedule each loop separately on the GPU. In this case, only the GPU runs parallel code, and it only runs one parallel loop at a time. Moreover, all unparallelized code regions are scheduled on a single CPU core. This is effective for programs with very large SIMD loops that dominate the program's execution.

Unfortunately, this approach is *not* effective for more complex programs. While there may still be many GPU-friendly SIMD loops in complex codes, the amount of work in each loop can vary significantly. In many cases, there may not be sufficient parallelism to fully utilize the GPU cores [6]. Moreover, complex programs also contain code with control divergence and irregular memory access patterns. And, these non-SIMD code regions tend to account for significant portions of execution time. By running them serially, significant performance degradation will occur. It will also underutilize the multiple CPU cores in a heterogeneous microprocessor, which along with the GPU are also a tremendous source of compute power.

We propose to exploit parallelism from multiple loops simultaneously when possible to more effectively utilize heterogeneous microprocessors. Such *multi-loop parallelism* is a generalization of existing parallel idioms, like *nested parallelism* or *pipeline parallelism*. In this paper, we focus on the former in which one or more SIMD loops are nested underneath a parallel outer loop, a common code structure in our benchmarks. We schedule the parallel outer loop on the CPU cores. The resulting multiple CPU threads then spawn multiple instances of the inner SIMD loop(s), often simultaneously, which get scheduled on the GPU cores. (While the frequency of spawns can be high if the inner loop has smaller amounts of work, the high-speed communication between a CPU and an integrated GPU enables the exploitation of such finer-grained SIMD loops). This exposes more parallelism for the GPU cores and parallelizes the non-SIMD outer loop. Our preliminary results, which estimate performance by combining separate CPU and GPU simulations, show that multi-loop parallelism outperforms single-loop parallelism by 3.12x across 3 OpenMP benchmarks.

The rest of this paper is organized as follows. Section 2 presents our new parallelization scheme. Then, Section 3 undertakes a quantitative evaluation of its effectiveness. Next, Section 4 discusses related work. Finally, Section 5 concludes the paper.

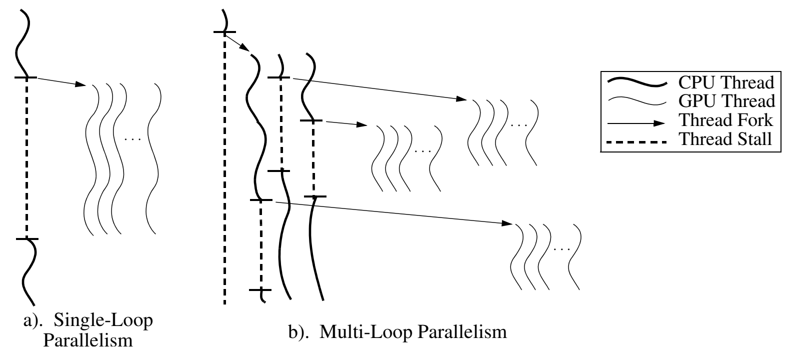


Fig. 1. CPU and GPU threads in a). single-loop and b). multi-loop parallelism.

2 Multi-Loop Parallelism

Traditional GPU workloads [7, 8] consist of massively parallel SIMD loops within which programs spend the vast majority of their time. (The massive loops have been necessary to amortize the large kernel initiation costs associated with discrete GPUs). In such workloads, it is sufficient to parallelize the SIMD loops individually, and to schedule them one at a time on the GPU. Figure 1a illustrates this *single-loop parallelism* case. The program starts out running non-SIMD code serially in a single CPU thread. When the CPU thread reaches a parallel SIMD loop, it spawns the corresponding GPU kernel which gets scheduled on the GPU. The GPU's cores then execute the SIMD loop in parallel as the CPU thread stalls. When GPU execution completes, the CPU thread is notified and continues executing.

In systems with integrated GPUs, the kernel initiation costs are much lower. This raises the possibility to use GPUs not only for massively parallel loops, but also for smaller loops (*i.e.*, with fewer loop iterations). Despite containing less work, such loops can still be performance critical if they occur within loop nests that execute them a large number of times.

Unfortunately, scheduling smaller SIMD loops on the GPU one at a time as is done in Figure 1a will perform poorly for two reasons. First, the small amounts of parallel work may not fully utilize the GPU. GPUs consist of *streaming multiprocessors* or *SMs*, each of which contain a large number of cores (also known as *streaming processors* or *SPs*). To keep all of its cores fully utilized, each SM manages a large number of hardware contexts, and employs hardware multithreading to schedule the threads residing within its contexts onto the cores in a time multiplexed fashion. Smaller SIMD loops may not yield a sufficient number of threads to fill all of the available hardware contexts, leading to vacant hardware multithreading slots. Another reason smaller SIMD loops may perform poorly under single-loop parallelism is the computations in between the SIMD loops–*i.e.*, the rest of the loop nest–may also be important. As shown in

Figure 1a, these non-SIMD portions of code would execute serially, limiting the overall gains due to Amdahl's Law.

To address this problem, we propose exploiting parallelism from multiple loops simultaneously–*i.e.*, *multi-loop parallelism.* In addition to executing SIMD loops on the GPU, our approach tries to *find additional parallelism outside of the SIMD loops to enable parallel execution on multiple CPU cores as well.* One source of multi-loop parallelism is nested parallelism. In this case, we look for one or more parallel SIMD loops that are nested underneath an outer loop that is also parallel.

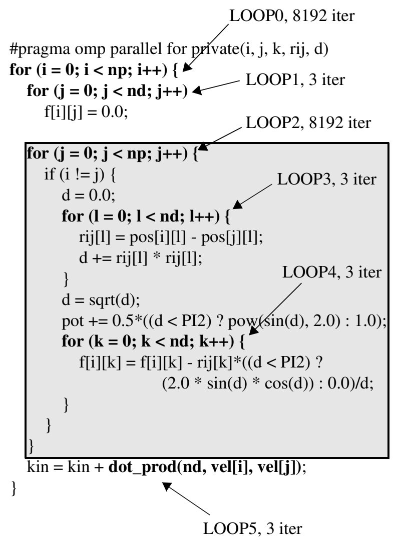
Figure 1b illustrates the multi-loop parallelism case. Like Figure 1a, the program starts out running serially in a single CPU thread. The CPU thread again spawns multiple threads, but this time it spawns CPU threads corresponding to a parallel outer (non-SIMD) loop that gets scheduled on multiple CPU cores. Then, each CPU thread reaches a nested parallel SIMD loop, and spawns the corresponding GPU kernel which gets scheduled on the GPU. Exploiting such multi-loop parallelism increases the performance of heterogeneous microprocessors in two ways. First, the multiple dynamic instances of the SIMD loops provide more parallelism to fill the GPU's hardware contexts and boost the GPU's utilization, especially when each SIMD loop is smaller. And second, portions of the loop nests outside of the SIMD loops that would have otherwise executed serially now execute in parallel on the CPU cores, thus addressing Amdahl's Law.

2.1 Code Examples

We surveyed different programs to look for multi-loop parallelism–specifically nested parallelism–and to understand the nature of these loop structures. We avoided looking at dense numeric codes commonly found in GPU benchmark suites. Instead, we focused on codes with more complex loop nests that one would not normally think of accelerating using GPUs.

Figures 2–4 present three representative examples from our code survey: MD and FFT6 from the OpenMP source code repository [9], and ART from the SPEC OMP 2001 suite [10]. All of these benchmarks have been parallelized using OpenMP [11]. In the figures, we show the main parallelized loop from each benchmark–*i.e.*, the loop identified by the "#pragma omp" directive. We treat this as the parallel outer loop for one instance of multi-loop parallelism. We also show all of the loops nested underneath the parallel outer loop, labeling each inner loop and indicating its iteration count. (For the inner-most loops in ART–*i.e.*, in the "compute_values_match()" function from Figure 4–we only show the loop labels in place of the full code due to the large size of this example).

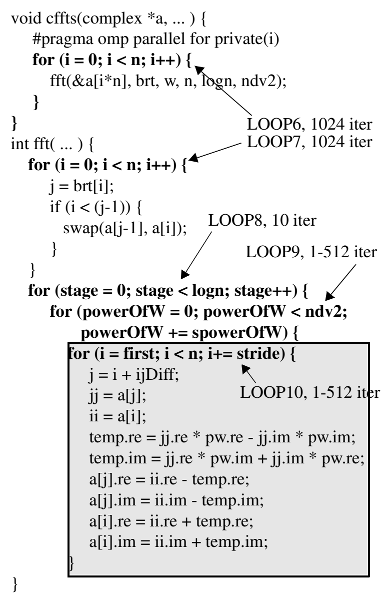
The code examples in Figures 2–4 illustrate several important features of the programs that our techniques try to target. First, there are no massive loops that dominate the computation. Instead, the codes contain a number of smaller loops. Some loops exhibit a trivial number of iterations (≤ 11) while others contain modest iteration counts (100s to a few 1000 iterations). Second, some loops exhibit irregularity that are also problematic for GPU execution. A GPU's SMs implement a *single instruction multiple thread* or *SIMT* execution model.



```
#pragma omp parallel for private(i, j, k, rij, d)
for (i = 0; i < np; i++) {
   for (j = 0; j < nd; j++)
      f[i][j] = 0.0;

   for (j = 0; j < np; j++) {
      if (i != j) {
         d = 0.0;
         for (l = 0; l < nd; l++) {
            rij[l] = pos[i][l] - pos[j][l];
            d += rij[l] * rij[l];
         }
         d = sqrt(d);
         pot += 0.5*((d < PI2) ? pow(sin(d), 2.0) : 1.0);
         for (k = 0; k < nd; k++) {
            f[i][k] = f[i][k] - rij[k]*((d < PI2) ?
                              (2.0 * sin(d) * cos(d)) : 0.0)/d;
         }
      }
   }
   kin = kin + dot_prod(nd, vel[i], vel[j]);
}
```

Fig. 2. Code example from the MD benchmark with multi-loop parallelism.

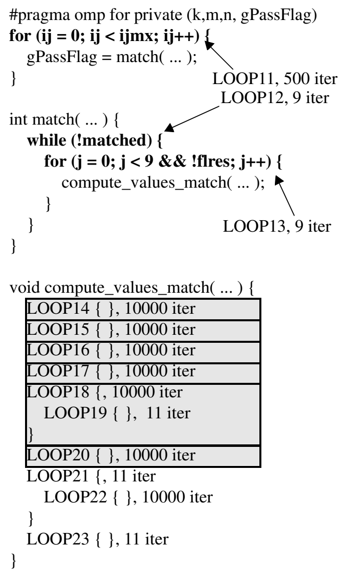


```
void cffts(complex *a, ... ) {
   #pragma omp parallel for private(i)
   for (i = 0; i < n; i++) {
      fft(&a[i*n], brt, w, n, logn, ndv2);
   }
}
int fft( ... ) {
   for (i = 0; i < n; i++) {
      j = brt[i];
      if (i < (j-1)) {
         swap(a[j-1], a[i]);
      }
   }
   for (stage = 0; stage < logn; stage++) {
      for (powerOfW = 0; powerOfW < ndv2;
            powerOfW += spowerOfW) {
         for (i = first; i < n; i+= stride) {
            j = i + ijDiff;
            jj = a[j];
            ii = a[i];
            temp.re = jj.re * pw.re - jj.im * pw.im;
            temp.im = jj.re * pw.im + jj.im * pw.re;
            a[j].re = ii.re - temp.re;
            a[j].im = ii.im - temp.im;
            a[i].re = ii.re + temp.re;
            a[i].im = ii.im + temp.im;
         }
}
```

Fig. 3. Code example from the FFT6 benchmark with multi-loop parallelism.

Under the SIMT model, threads are executed in groups, called *warps* (which are also the unit of hardware multithreading). While threads within a warp are allowed to execute different control paths, for maximum performance, intra-warp threads should follow the same control path and perform the same operation every cycle. Some loops from our code examples exhibit control divergence which can degrade throughput (*e.g.*, LOOP2 in Figure 2 and LOOP7 in Figure 3). Moreover, while threads within a warp are allowed to access arbitrary memory locations, for maximum performance, intra-warp threads should access contiguous memory locations whenever they execute memory operations. Some loops from our code examples exhibit strided memory accesses (*e.g.*, LOOP10 in Figure 3) that can increase the memory bandwidth requirements. Finally, besides containing smaller amounts of work and exhibiting control and memory divergence, our codes also exhibit complex loop structure. There is a large number of nesting levels, and in some cases, nesting occurs across multiple function calls. ART in Figure 4 is an extreme example of code distributed across a large number of deeply nested loops.

Interestingly, these complex loop nests in Figures 2–4 lead to nested parallelism. In addition to the parallel loops that are explicitly identified by the OpenMP pragmas, a careful examination of the other loops inside the loop nests reveals a number of them are parallel as well. (Although not shown in Figure 4, all of the inner loops in ART, labeled LOOP14–LOOP23, are parallel). We find



```
#pragma omp for private (k,m,n, gPassFlag)
for (ij = 0; ij < ijmx; ij++) {
   gPassFlag = match( ... );
}                                    LOOP11, 500 iter
                                      LOOP12, 9 iter
int match( ... ) {
   while (!matched) {
      for (j = 0; j < 9 && !flres; j++) {
         compute_values_match( ... );
      }
   }                                  LOOP13, 9 iter
}

void compute_values_match( ... ) {
   LOOP14 { }, 10000 iter
   LOOP15 { }, 10000 iter
   LOOP16 { }, 10000 iter
   LOOP17 { }, 10000 iter
   LOOP18 {, 10000 iter
      LOOP19 { },  11 iter
   }
   LOOP20 { }, 10000 iter
   LOOP21 {, 11 iter
      LOOP22 { }, 10000 iter
   }
   LOOP23 { }, 11 iter
}
```

Fig. 4. Code example from the ART benchmark with multi-loop parallelism.

this is fairly common in OpenMP programs because the marked parallel loops often appear at the outer levels of deep loop nests, as in our code examples. Such coarse-grained parallelism is almost impossible for a compiler to extract automatically, but is natural for a programmer to express given his/her knowledge of the code at the algorithm level. Moreover, programmers are incentivized to express coarse-grained parallelism since it is a good match for CPU cores, the main target for OpenMP programs. Given such explicitly parallel outer loops, nested parallelism arises whenever one or more inner loops are found to be parallel, which is quite likely in OpenMP programs when each parallel region contains many inner loops.

Notice, both CPUs and GPUs are needed to exploit the multiple levels of parallelism in Figures 2–4. Inner loops with non-trivial iteration counts have a greater chance for GPU acceleration (even if they exhibit control and memory divergence), whereas loops at outer levels are more appropriate for CPUs. Such codes are a good match for heterogeneous microprocessors.

3 Experimental Results

This section conducts a preliminary evaluation of multi-loop parallelism for the codes in Figures 2–4. We do not yet have these codes running on simulators of heterogeneous microprocessors. Instead, we employ simulators that model CPUs and GPUs *separately*, and then we estimate the integrated CPU-GPU performance by combining the separately acquired results.

Our study uses SimpleScalar [12] to simulate a CPU core, and GPGPU-Sim [13] to simulate a GPU. We configure SimpleScalar to model a 4-way out-of-order core running at 2.4 GHz with an 128-entry reorder buffer. This CPU core has a 2-level cache hierarchy with a split 16KB L1 cache and a unified 256KB L2 cache. We use this configuration in both a single-core system as well as a 4-core system. We configure GPGPU-Sim to model an Nvidia GTX-480 running at 700 MHz. We simulate 8 streaming multiprocessors, each containing 16 streaming processors that support 32 threads per warp. Each SM also contains a 16KB L1 cache, and all 8 SMs share a 786KB L2 cache. This GPU configuration along with the CPU configuration–*i.e.*, 8 SMs opposing 4 CPU cores–resembles recent integrated GPUs from AMD (*e.g.*, the 7000-series heterogeneous APU processors [3]). Lastly, we do not simulate hardware cache coherence between the CPU and GPU, so our current experiments assume software is responsible for managing coherence. In both simulators, we assume off-chip main memory incurs a latency of 50 ns and provides 88.7 GB/s of memory bandwidth.

To begin our study, we first estimate the performance provided by single-loop parallelism–*i.e.*, off-loading loops one at a time as shown in Figure 1a. We timed every loop from Figures 2–4 on both SimpleScalar and GPGPU-Sim. For SimpleScalar, we simulated each OpenMP region once using a binary in which each loop entry and exit point was marked so that we could break down the execution time per loop. For GPGPU-Sim, we created multiple CUDA kernels to off-load each loop individually onto the GPU, and simulated each kernel separately on GPGPU-Sim. (Although GPGPU-Sim assumes discrete GPUs, we omit the data transfer times since these would not be incurred by integrated GPUs computing on the data "in place"). After acquiring these results, we were able to identify the loops that run faster on the GPU. These are the loops shaded gray in Figures 2–4.

To estimate the performance for off-loading these GPU-friendly loops, we performed a second run of each OpenMP region on SimpleScalar, but this time, we flushed the CPU's caches at each loop entry and exit point. (The original GPU simulations for each loop already start with cold caches and also include a flush at the end). Flushing ensures the most up-to-date data values are used each time computation is off-loaded to the GPU even when there is no CPU-GPU cache coherence. However, it adds overhead to every off-load event. (Later, we will discuss the implications of this overhead). Then, we constructed a trace of execution through each OpenMP region, alternating between the CPU core (with flushing) and the GPU at the off-loaded loop entry and exit points. We also added a fixed 100-cycle delay for each off-load event to account for initiating a new kernel on the GPU. (This is a conservative number considering the CPU and GPU are on the same chip).

Figure 5 presents the results. The bars in Figure 5 labeled "Sequential" report the execution time when the entire OpenMP execution trace runs on the CPU core, and the bars labeled "Single-Loop" report the estimated execution time when the execution trace adopts the GPU performance for those loops that run faster on the GPU. All bars are normalized to the sequential bars. As Figure 5

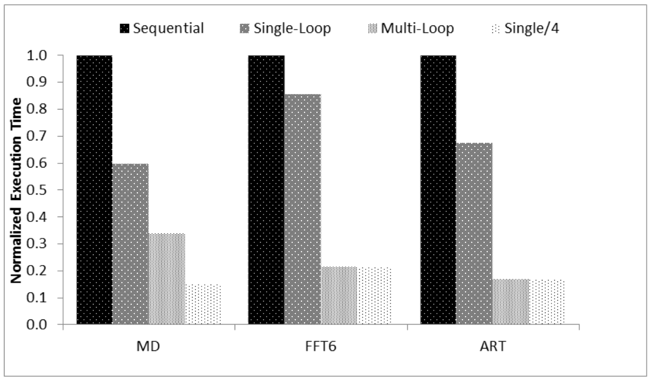


Fig. 5. Normalized execution time for sequential, single-loop and multi-loop parallelism, and multi-loop parallelism without contention.

shows, the GPU provides a speedup of 1.67x, 1.17x, and 1.48x for MD, FFT6, and ART, respectively. On average, the speedup is 1.44x.

While these speedups are significant, they are also quite disappointing considering the capability of the GPU. The problem is the complex nature of the codes in Figures 2–4, as described earlier. In particular, the codes contain a mix of SIMD and non-SIMD loops, so only a portion of each OpenMP region can run on the GPU. As Figures 2–4 show, there is a lot of unshaded code that must run on the CPU core. This limits the overall gain from the GPU due to Amdahl's Law. To make matters worse, the gains for the SIMD loops are mixed. Figure 6 compares the execution time of each shaded (off-loaded) loop from Figures 2–4 running on SimpleScalar and GPGPU-Sim. In some cases, the gains are large (14.3x speedup for LOOP18), but in other cases the gains are small (1.53x speedup for LOOP10). This is due to the characteristics of our more complex codes: small number of iterations (LOOP10), control flow divergence (LOOP2), and non-unit stride memory accesses (LOOP10, LOOP14, LOOP16, and LOOP18).

Next, we estimate the performance for multi-loop parallelism. As before, we construct a trace of execution through each OpenMP region, but this time we build a parallel trace. We assume the outer-most parallel loops identified by the OpenMP pragmas in Figures 2–4 run on 4 CPU cores, so we divide all of their iterations into 4 blocks that are executed simultaneously by 4 parallel traces. Within each trace, we again alternate between the CPU core and the GPU (*i.e.*, for the unshaded and shaded codes in Figures 2–4, respectively), selecting the CPU execution or GPU execution from the SimpleScalar/GPGPU-Sim simulations to estimate the speed of each trace. We also assume the 4 CPU cores *share* the 8-SM GPU, so there is contention whenever 2 or more CPU cores off-load loops onto the GPU. To model this, we schedule loops in the order that they arrive at the GPU, delaying later loops behind earlier loops. Moreover, as a loop's thread blocks complete and the next loop's thread blocks begin execution, there may be multiple kernels in the GPU at the same time. We ran additional

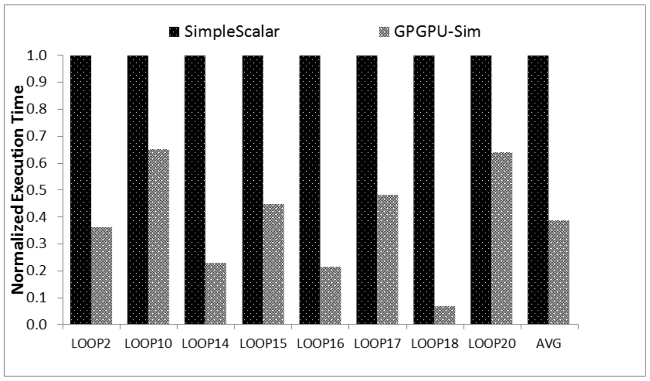


Fig. 6. Per-loop execution time on the GPU and CPU.

simulations on GPGPU-Sim to measure the speed of each off-loaded loop when paired with another off-loaded loop in the GPU, trying all possible ways of dividing the 8 SMs between the pair. (For ART, we did this for all pairings of LOOP14, LOOP15, LOOP16, LOOP17, LOOP18, and LOOP20).

In Figure 5, the bars labeled "Multi-Loop" report the estimated execution time for multi-loop parallelism normalized to the "Sequential" bars. As Figure 5 shows, multi-loop parallelism significantly out-performs single-loop parallelism. Compared to the "Sequential" bars, multi-loop parallelism provides an overall speedup of 2.94x, 4.62x, and 5.91x for MD, FFT6, and ART, respectively. On average, the gain is 4.49x which is 3.12x greater than single-loop parallelism's gain. This is due to the parallel execution of the non-SIMD code in the OpenMP regions, thus addressing the Amdahl's Law limitation of single-loop parallelism. It is also due to executing multiple loops simultaneously on the GPU, providing more work (especially for smaller loops like LOOP10) to better utilize the GPU's cores.

We find single-loop parallelism often under-utilizes the GPU, leaving ample GPU compute bandwidth for multi-loop parallelism. To show this, the bars labeled "Single/4" in Figure 5 report the single-loop parallelism execution time divided by 4, which represents the ideal speedup of multi-loop parallelism (*i.e.*, in the absence of GPU contention). For FFT6 and ART, there is almost no difference between the "Multi-Loop" and "Single/4" bars, indicating virtually no contention. For MD, the ideal speedup is another 2.27x faster, indicating contention is an issue in this benchmark. (But even in this case, multi-loop parallelism is still 1.8x faster than single-loop parallelism). For MD and other benchmarks that may experience GPU contention, programmers could try to alleviate the demand on GPU resources by reducing the number of parallel CPU threads. (Further study is needed to determine whether this can help).

Hardware Coherence Impact. In the above, we assumed that the only sharing between CPU and GPU cores occurs through off-chip DRAM. There is no on-chip cache coherence, hence the use of flushing in our experiments. Although this is consistent with many previous heterogeneous microprocessors, there is a trend recently towards providing CPU-GPU cache coherence in hardware. In such coherent heterogeneous microprocessors, there would be no need for flushing operations. As a result, the performance for both single-loop and multi-loop parallelism in Figure 5 would likely increase relative to the sequential baseline. However, we do not expect the relative gains *between* single-loop and multi-loop parallelism to change substantially since *both* approaches would benefit from the elimination of the flushing operations. Investigating multi-loop parallelism in the context of coherent CPU-GPU caches is an important research direction.

4 Related and Future Work

Simultaneously utilizing CPU and GPU cores in a heterogeneous microprocessor is not a new idea. Recent work has already proposed doing this [14, 15, 16]. But these existing techniques still only exploit single-loop parallelism: they schedule iterations from a single SIMD loop across both types of cores. As we have shown, single-loop parallelism cannot keep the GPU (let alone the CPU *and* GPU) fully utilized for complex codes with small SIMD loops, nor can it parallelize the non-SIMD code. By exploiting parallelism from multiple loops within a loop nest, we expose greater amounts of SIMD parallelism and parallelize the non-SIMD code as well.

Multi-loop parallelism is also related to DTBL [6] or dynamic thread-block launch. In DTBL, a loop off-loaded by the CPU onto the GPU can initiate other instances of itself from the GPU. (This supports certain forms of dynamic parallelism–for example, vertex expansion during recursive graph search). Like multi-loop parallelism, DTBL enables multiple dynamic instances of a loop to execute simultaneously on the GPU. However, DTBL only exploits the GPU. It does not try to schedule loops onto the CPU and GPU simultaneously, and hence, is not designed for heterogeneous microprocessors. Moreover, DTBL only exposes SIMD parallelism, and thus cannot parallelize non-SIMD loops.

In the future, we hope to extend our research. One key extension is to repeat our experiments on a simulator of an actual heterogeneous microprocessor in order to validate our current results (which were obtained using separate CPU / GPU simulators). This new study would allow us to evaluate the impact of different architectural assumptions, like CPU-GPU cache coherence (see Section 3), which can only be quantified on an integrated simulator. And, it could also highlight new sources of resource contention that we do not currently model. For example, in a heterogeneous microprocessor, the CPU and GPU share a common DRAM interface which can lead to contention for external memory bandwidth (even starvation from the CPU's standpoint). Mitigating such contention effects may require new techniques, such as prioritizing the CPU's memory requests over the GPU's memory requests.

Finally, another future research direction is to investigate new parallelization idioms. In this paper, we focused on a specific form of multi-loop parallelism involving nested parallel loops, but we envision other forms of multi-loop parallelism are possible. Distributed parallel loops, perhaps involved in *pipeline parallelism*, are potentially one alternative. For example, different pipeline stages consisting of non-SIMD and SIMD loops could be mapped to CPU and GPU cores. Another possibility is to combine multiple forms of parallelism, for example task-level and loop-level parallelism. In this case, threads executing different tasks or functions could run on CPU cores in parallel, each of which may execute SIMD loops that are in turn off-loaded onto GPU cores. By expanding the forms of parallelism our techniques can handle, we will be able to generalize to a wider range of applications.

5 Conclusion

Given the wide availability of heterogeneous microprocessors, we believe an exciting research direction is to find new computations that can benefit from integrated GPUs. We argue this will require new parallelization methods that can make effective use of both CPU and GPU cores simultaneously. In this paper, we propose one such novel method, multi-loop parallelization, and analyze its benefits for nested parallel loops. Our results show exploiting multi-loop parallelism can outperform the conventional approach of parallelizing loops one at a time by 3.12x for programs with complex loop nests. In the future, we plan to investigate different parallelization strategies that can support other types of looping structures.

References

[1] "Intel Corporation. Intel Sandy Bridge Microarchitecture. *http://www.intel.com.*"

[2] N. Brookwood, "AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. AMD White Paper." 2010.

[3] "Compute Cores: A New Era of Computing. *http://http://www.amd.com/en-us/innovations/software-technologies/processors-for-business/compute-cores.*"

[4] M. Wilkins, "NVIDIA Jumps on Graphics-Enabled Microprocessor Bandwagon," 2011.

[5] C. Gregg and K. Hazelwood, "Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer," in *Proceedings of the International Symposium on Performance Analysis of systems and Software*, 2011.

[6] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs," in *Proceedings of the International Symposium on Computer Architecture*, Portland, OR, June 2015.

[7] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proceedings of the International Symposium on Workload Characterization*, Atlanta, GA, December 2010.

[8] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei Hwu, "The Parboil Technical Report," March 2012.

[9] "OpenMP Source Code Repository. *http://www.pcg.ull.es/ompscr/.*" 2004.

[10] "SPEC OMP 2001. *https://www.spec.org/omp2001/.*" 2001.

[11] "The OpenMP API Specification for Parallel Programming. Intel Corporation. *http://www.openmp.org/wp/.*" 2014.

[12] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison, CS TR 1342, June 1997.

[13] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Boston, MA, April 2009.

[14] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive Heterogeneous Scheduling for Integrated GPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, AB, Canada, August 2014.

[15] V. T. Ravi and G. Agrawal, "A Dynamic Scheduling Framework for Emerging Heterogeneous Systems," in *Proceedings of the 18th International Conference on High Performance Computing*, December 2011.

[16] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal, "Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations," in *Proceedings of the International Conference on Supercomuting*, Tsukuba, Ibaraki, Japan, June 2010.