Nested MIMD-SIMD Parallelization for Heterogeneous Microprocessors

DANIEL GERZHOY, XIAOWU SUN, MICHAEL ZUZAK, and DONALD YEUNG,
University of Maryland at College Park

Heterogeneous microprocessors integrate a CPU and GPU on the same chip, providing fast CPU-GPU communication and enabling cores to compute on data "in place." This permits exploiting a finer granularity of parallelism on the integrated GPUs, and enables the use of GPUs for accelerating more complex and irregular codes. One challenge, however, is exposing enough parallelism such that both the CPU and GPU are effectively utilized to achieve maximum gain.

In this article, we propose exploiting nested parallelism for integrated CPU-GPU chips. We look for loop structures in which one or more regular data parallel loops are nested within a parallel outer loop that can contain irregular code (e.g., with control divergence). By scheduling the outer loop on multiple CPU cores, multiple dynamic instances of the inner regular loop(s) can be scheduled on the GPU cores. This boosts GPU utilization and parallelizes the outer loop. We find that such *nested MIMD-SIMD parallelization* provides greater levels of parallelism for integrated CPU-GPU chips, and additionally there is ample opportunity to perform such parallelization in OpenMP programs.

Our results show nested MIMD-SIMD parallelization provides a 16.1x and 8.67x speedup over sequential execution on a simulator and a physical machine, respectively. Our technique beats CPU-only parallelization by 4.13x and 2.40x, respectively, and GPU-only parallelization by 2.74x and 2.26x, respectively. Compared to the next-best scheme (either CPU- or GPU-only parallelization) per benchmark, our approach provides a 1.46x and 1.23x speedup for the simulator and physical machine, respectively.

CCS Concepts: • **Computer systems organization → Parallel architectures**; *Multiple instruction, multiple data*; *Single instruction, multiple data*; *Heterogeneous (hybrid) systems*; • **Computing methodologies →** *Graphics processors*;

Additional Key Words and Phrases: Heterogeneous microprocessor, GPU, SIMD, MIMD, nested parallelism

ACM Reference format:
Daniel Gerzhoy, Xiaowu Sun, Michael Zuzak, and Donald Yeung. 2019. Nested MIMD-SIMD Parallelization for Heterogeneous Microprocessors. *ACM Trans. Archit. Code Optim.* 16, 4, Article 48 (December 2019), 27 pages.
https://doi.org/10.1145/3368304

1 INTRODUCTION

For many years, processor manufacturers have been producing *heterogeneous microprocessors* in which a CPU and GPU are integrated on the same die. For example, Intel [1] and AMD [2] have

Authors' address: D. Gerzhoy, X. Sun, M. Zuzak, and D. Yeung, 1323 A. V. Williams, University of Maryland at College Park, College Park, MD 20742; emails: {dgerzhoy, xsun24, mzuzak, yeung}@umd.edu.

1544-3566/2019/12-ART48
https://doi.org/10.1145/3368304

done so for x86 processors, and Apple [3] has done so for ARM-based SoCs. 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 copy operations between separate CPU and GPU memories. In addition, the tight integration permits fast CPU-GPU communication, either through off-chip physical memory or on-chip cache coherence.

Because integrated GPUs do not need to copy data and enjoy high-speed communication with the CPU, they are capable of exploiting a finer granularity of parallelism compared to discrete GPUs, which in contrast require massive parallelism to amortize large startup latencies [4]. This will allow integrated GPUs to accelerate a wider range of loops. Furthermore, the support for shared memory between CPU and GPU will also greatly simplify programming—for instance, 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 GPU researchers try to accelerate more complex programs, a major challenge will be effectively parallelizing these irregular codes for heterogeneous microprocessors. GPUs are commonly used to accelerate data parallel loops with regular parallelism. The conventional approach is to parallelize such 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. (All unparallelized code regions are scheduled on a single CPU core.) For programs that contain large regular loops that dominate execution time, this approach works well.

But for more complex programs, running regular loops one at a time on the GPU can lead to poor performance and/or leave potential performance gains on the table. One problem is that while complex programs do exhibit GPU-friendly loops, the amount of parallelism can vary significantly. In many cases, individual loops may contain only modest levels of parallelism. Such smaller loops can still be profitably off-loaded onto integrated GPUs, especially given the low communication overheads discussed earlier. But, they may achieve lower speedups, and if executed one at a time, each loop cannot fully utilize the GPU's cores [5]. Another problem is that complex programs tend to contain code with control divergence and irregular memory access patterns that can perform poorly on GPUs. These irregular code regions can account for significant portions of execution time. If they are run serially on a single CPU core, not only are the CPU cores underutilized but also Amdahl's law will limit the performance gains that are possible.

To achieve higher performance for complex programs, it is necessary to expose greater amounts of parallelism such that the GPU and CPU cores available in a heterogeneous microprocessor are both more fully utilized. On the GPU side, given the problem of smaller regular loops, one way to boost parallelism is to off-load *multiple loops* onto the GPU simultaneously when possible. Recently, researchers have investigated concurrent kernel launch [6, 7] that makes exposing such multi-kernel GPU parallelism possible. On the CPU side, given the problem of serial irregular code, parallelizing the irregular code regions so they can run on multiple CPU cores is needed to address Amdahl's law. In other words, *heterogeneous parallelism*—or, parallelization of CPU code regions in addition to GPU code regions—is necessary to achieve high performance.

But how can we uncover both multi-kernel and heterogeneous parallelism from the irregular code structures that are found in complex programs? Some researchers have already begun examining this question. In particular, Wende et al. [8] demonstrate a CPU-GPU parallelization scheme for a molecular thermodynamics code, called *GLAT*. The authors observe that the GLAT code processes two different types of molecules, all of which can be performed in parallel. Their approach maps the processing of one type of molecule onto CPU cores and the other type of molecule onto

GPU cores. By exposing the *distributed parallelism* that exists within the GLAT code, both CPU and GPU cores can be kept busy at the same time.

In this work, we study a new parallelization scheme for heterogeneous microprocessors that is based on *nested parallelism*. Specifically, we propose exploiting nested parallel structures in which one or more regular parallel loops are nested within an outer parallel region. The latter could be an outer loop that is also parallel, or it could be code exhibiting task-level parallelism. Although the inner loops are regular and can be gainfully executed on GPUs, the outer parallel region can be problematic. In the case of a parallel outer loop, there could be irregular code that performs poorly on GPUs. In the case of a task-level parallel region, it would not even be possible to run the code on a GPU. We schedule the outer parallel loop/region on the MIMD CPU cores. The resulting multiple CPU threads then spawn multiple instances of the inner regular loops, often simultaneously, which get scheduled on the SIMD (or SIMT) GPU cores. Hence, this achieves our goal of providing both multi-kernel GPU parallelism and CPU parallelism, all from a single nested parallel structure.

We call this technique *nested parallelization* for MIMD-SIMD architectures, or *nested MIMD-SIMD parallelization*. Our technique can increase utilization in heterogeneous microprocessors, and the nested parallelism it exploits is also quite common in practice. For example, we find that it occurs frequently in programs parallelized using OpenMP [9]. For complex code regions, OpenMP pragmas often appear at the top of nested code structures and encapsulate many nested loops. This happens because (1) OpenMP programs are parallelized by humans who tend to express outer loop parallelism given their algorithm-level understanding of the code, and (2) programmers are incentivized to expose coarse-grained parallelism since it tends to result in higher performance on multi-core CPUs. Given explicitly parallel outer regions, nested parallelism arises whenever one or more inner loops are parallel, which is quite likely if each parallel region contains many inner loops.

This article makes several contributions in the context of parallelization schemes for heterogeneous microprocessors:

- We propose using nested MIMD-SIMD parallelization to expose multi-kernel and heterogeneous parallelism for CPU and GPU cores. (Currently, our parallelization technique is performed by hand. It also requires the GPU to support multi-kernel launch.) To show that such parallelism is quite common, we provide several illustrative examples taken from existing OpenMP programs.
- We use a loop transformation found in vectorizing compilers to increase the likelihood for finding regular parallel loops for GPUs within OpenMP parallel regions. Although certain inner loops may be serial, portions of them can still exhibit GPU-friendly computations. We perform *loop fission* to extract the SIMD portions to form separate parallel loops, enabling nested MIMD-SIMD parallelization even when the original loop only contains single-level parallelism.
- We conduct an in-depth experimental evaluation of our new parallelization scheme. Our evaluation begins with experiments on a cycle-accurate simulator, gem5-gpu [10]. The simulator models a heterogeneous microprocessor employing four CPU cores and a fairly aggressive GPU with 16 streaming multi-processors (SMs) supporting 24,576 hardware threads. Seven OpenMP programs drive the experiments. Our simulation results show that nested MIMD-SIMD parallelization provides a 16.1x speedup over a single CPU core. Our results also show nested MIMD-SIMD parallelization beats CPU-only parallelization by 4.13x and GPU-only parallelization by 2.74x. Compared against the next-best scheme (either CPU- or GPU-only parallelization) across all of our benchmarks, nested MIMD-SIMD parallelization provides a 1.46x speedup.

- In addition to the simulator, we also conduct experiments on a physical machine with an Intel Core i7-6700. The physical machine employs four CPU cores as well but has a more modest GPU. On the physical platform, the gain of nested MIMD-SIMD parallelization drops a bit, to 8.67x. This is mainly due to the less aggressive GPU in the physical platform compared to the simulator platform. The physical machine results also show that nested MIMD-SIMD parallelization beats CPU-only parallelization by 2.40x and GPU-only parallelization by 2.26x. Compared against the next-best scheme, nested MIMD-SIMD parallelization provides a 1.23x speedup on the physical machine.

The rest of this article is organized as follows. Section 2 begins by discussing related work. Then, Section 3 presents our new parallelization scheme, and Section 4 introduces our loop fission transformation. Next, Section 5 discusses the methodology used for the simulation study, and Section 6 presents our simulation results. Section 7 follows by presenting results on the physical machine. Finally, Section 8 concludes the article.

2 RELATED WORK

Several researchers have measured the performance benefits of heterogeneous microprocessors. In particular, Daga et al. [11] and Spafford et al. [12] evaluate AMD's Fusion architecture [2]. Both examine how CPU-GPU integration in Fusion addresses the communication bottlenecks faced by discrete systems [4]. However, Daga and Spafford only consider traditional GPU workloads, such as the SHOC [13] and HPC Challenge [14] benchmark suites. They do not look for new and potentially more complex codes that become enabled by CPU-GPU integration.

The work by Arora et al. [15] recognizes that heterogeneous microprocessors are capable of handling a wider range of programs, so they consider a mix of traditional GPU workloads (Rodinia [16]), as well as benchmarks from the SPEC CPU 2000/2006 suites [17]. The latter contain more irregular codes exhibiting a variety of data parallel loops, which is also the focus of our work. But unlike our research, Arora is concerned mainly with CPU performance. (They observe that CPUs will execute much more serial code as parallel loops are off-loaded onto the GPU.) In contrast, our research is concerned with the performance of both CPU and GPU as new parallelization techniques try to leverage both types of cores in concert.

Besides studying the benefits of heterogeneous microprocessors, several research efforts have developed parallelization techniques to make more effective use of both CPU and GPU cores simultaneously. Specifically, this article is based on our own prior work [18] that first introduced the idea of mapping nested parallelism to integrated CPU-GPU chips. Compared to our previous work, the current work adds loop fission for extracting SIMD portions from serial loops. In addition, the experimental evaluation in this article (using both the cycle-accurate simulator and the physical machine) is completely new.

Preceding our work, several researchers have proposed scheduling iterations from the same data parallel loop across both CPU and GPU cores [19–21]. As we show in this article, however, such *homogeneous parallelism* from a single loop cannot keep the GPU (let alone the CPU *and* GPU) fully utilized given smaller loops, which can occur in some complex codes. Exploiting parallelism from a single loop only also ignores the limitations of the irregular code regions outside of that loop. By exploiting heterogeneous parallelism from both regular and irregular loops, our approach exposes greater amounts of parallelism for the GPU and parallelizes the irregular code on CPUs as well.

Wende et al. [8] demonstrate a CPU-GPU parallelization scheme on the GLAT molecular thermodynamics code. Similar to our work, their approach extracts parallelism from different loops for execution on CPU and GPU cores. But whereas they identify distributed parallel loops, we exploit nested parallel loops. More importantly, their study is specific to a single benchmark only.

In contrast, we look at a greater number of benchmarks. We also argue that our approach will be relevant for many irregular codes parallelized using OpenMP, which encompasses a very large number of programs.

Our work is also related to dynamic parallelism. Recent GPUs allow a GPU thread to initiate another GPU kernel [22]. Researchers have developed several parallelization schemes around this idea of internal kernel launches, including dynamic thread block launch (DTBL) [5], nested parallelism in CUDA (CUDA-NP) [23], and lazy nested parallelism (LazyNP) [22]. In DTBL, the child threads are dynamic instances of the original kernel launched by the CPU (i.e., the parallelism occurs within recursive code). In CUDA-NP and LazyNP, the child threads belong to parallel loops nested within the original kernel launched by the CPU. Like our parallelization scheme, CUDA-NP and LazyNP also exploit nested parallelism. However, DTBL, CUDA-NP, and LazyNP only schedule loops on the GPU. In particular, the outer parallel loop/code runs on the GPU, whereas in our nested MIMD-SIMD parallelization technique, it runs on the CPU. By making use of the CPU, our parallelization scheme can support irregular parallel loops that perform poorly on the GPU or code with task-level parallelism that cannot run on the GPU at all.

In addition to these parallelization schemes, there has been work that focuses primarily on new kernel launch mechanisms. Our nested MIMD-SIMD parallelization scheme requires multi-kernel launch. Prior work has investigated mechanisms for off-loading multiple simultaneous kernels onto the GPU [6, 7], either from a single CPU thread or from multiple CPU threads. Although our techniques rely on these mechanisms, our main focus is on identifying the parallel idioms (e.g., nested parallel loops) that give rise to multiple simultaneous kernels. Previous research has focused more on the launch mechanisms themselves.

Finally, instead of using the GPU to accelerate regular inner loops as we propose in our work, it is also possible to do so via SIMD instruction extensions available in today's CPUs [24]. SIMD instruction extensions have lower overhead than GPU kernel launches, so they do not incur the off-loading costs that exist in our techniques. However, SIMD instruction extensions cannot exploit as much parallelism due to the much narrower datapath in the CPU compared to the GPU. It is possible to combine nested MIMD-SIMD parallelization with SIMD instruction extensions, using the latter for smaller inner loops that are the most sensitive to kernel launch overhead. This could provide even higher performance than our techniques applied alone. In this article, we focus on understanding nested MIMD-SIMD parallelization by itself. We leave its combination with SIMD instruction extensions for future work.

3 NESTED MIMD-SIMD PARALLELIZATION

Traditional GPU workloads [16, 25], commonly used to study discrete GPUs, consist of massive regular data parallel loops within which programs spend the majority of their time. In such workloads, it is sufficient to off-load the regular loops sequentially, and to schedule them onto the GPU one at a time. Figure 1(a) illustrates this *single-loop SIMD parallelization* scheme. In Figure 1(a), the program starts running code serially in a single CPU thread. When the CPU thread reaches a regular data parallel loop, it off-loads the corresponding GPU kernel that gets scheduled on the GPU. The GPU's cores execute the kernel in parallel as the CPU thread stalls. (The CPU thread could continue if its code is independent of the GPU kernel, but this would not provide much gain for this case as explained in the following). When GPU execution completes, the CPU thread is notified and continues execution, possibly off-loading other kernels later on.

This parallelization scheme works well for traditional GPU workloads, in part because their off-loaded loops contain significant amounts of work (i.e., W_{SIMD} in Figure 1(a) is large). Thus, the runtime overhead associated with each off-load operation is amortized, even for discrete GPUs that can exhibit very long off-load latencies. The off-loaded loops from traditional GPU workloads

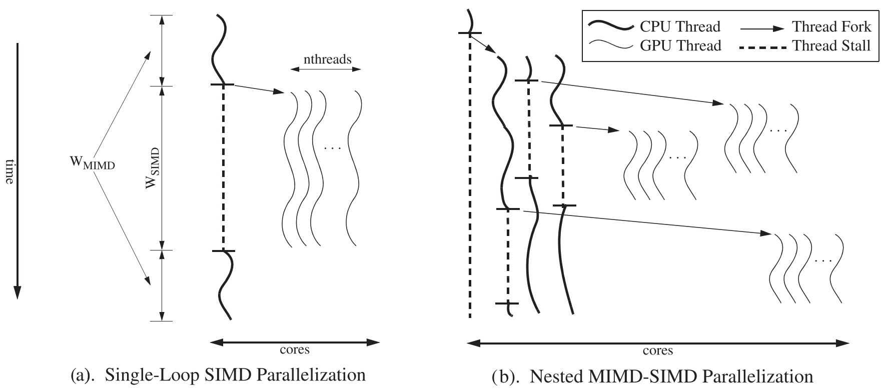


Fig. 1. CPU and GPU concurrency from single-loop SIMD parallelization (a) and nested MIMD-SIMD parallelization (b). W_{SIMD} is the amount of work in each loop parallelized for the GPU, W_{MIMD} is the amount of work outside of each GPU loop, and *nthreads* is the number of GPU threads from each GPU loop.

also provide very significant parallelism (i.e., *nthreads* in Figure 1(a) is large) such that a single kernel can effectively utilize the GPU's parallel hardware and hence attain large speedups. They also tend to dominate the work performed in the other code regions (i.e., $W_{SIMD} \gg W_{MIMD}$ in Figure 1(a)). This means serially executing the code outside of the GPU loops, even after the CPU thread stalls for the GPU to finish, does not degrade performance much.

In this work, we try to move away from traditional GPU workloads, and instead leverage GPUs for more complex and irregular programs that do not exhibit the aforementioned characteristics. For instance, the programs we target may contain regular loops with smaller amounts of work, or small W_{SIMD}. Although off-loading onto discrete GPUs would be prohibitive, the reduced overheads of integrated GPUs potentially make off-loading such finer-grained loops possible.

Besides a smaller W_{SIMD}, the programs we target may also have regular loops with a smaller *nthreads*, so individual loops may not exhibit sufficient parallelism for the GPU. A GPU consists of SMs, each of which contain a large number of cores (also known as *streaming processors* (SPs)). To keep all of its cores busy, each SM manages a large number of hardware thread contexts and employs hardware multi-threading to schedule the threads onto the cores. When *nthreads* is less than the number of hardware thread contexts, some of the contexts will be vacant—in other words, the GPU becomes underutilized *spatially*. Thus, we do not fully exploit the potential of the GPU, which can result in reduced performance gains for the off-loaded loops.

In addition to a small W_{SIMD} and/or small *nthreads*, another problem is that complex programs can spend significant time in the code outside of the GPU loops (i.e., $W_{MIMD} \approx W_{SIMD}$), where there can be irregular code, especially data-dependent branches causing control divergence, that performs poorly on GPUs. Again, the single-loop SIMD parallelization scheme loses its effectiveness, this time because it only parallelizes the regular loops and ignores the potentially irregular code outside of those loops. As such, the overall gains would be limited due to Amdahl's law. Note, a large W_{MIMD} will also result in less frequent off-loading operations onto the GPU. This can cause the GPU to be idle for significant periods of time—in other words, the GPU becomes underutilized *temporally*.

To enable GPUs for more complex and irregular programs, our work explores the possibility of exploiting parallelism from multiple parallel code structures simultaneously. In addition to executing regular loops on the GPU, our approach tries to utilize additional parallelism outside of the regular GPU loops to enable parallel execution on multiple MIMD CPU cores as well. One source of such multi-level parallelism is *nested parallelism*. In our work, we look for one or more regular parallel loops that are nested within an outer parallel code structure, usually another loop. (In one of our benchmarks, the outer parallel code is a work queue, and the work queue operations are performed in a task-parallel fashion.) We call this approach *nested MIMD-SIMD parallelization*.

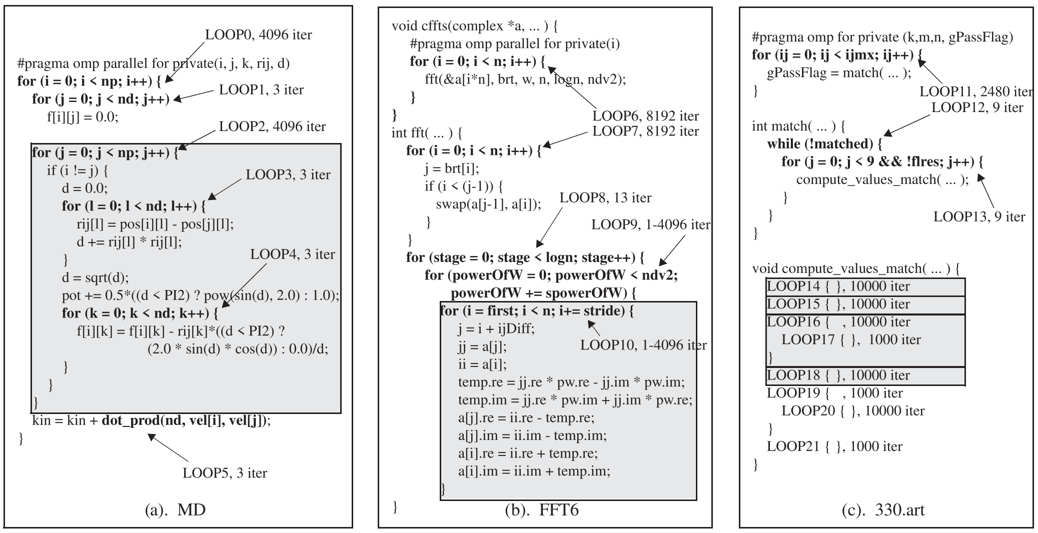
Figure 1(b) illustrates our nested MIMD-SIMD parallelization scheme. Like Figure 1(a), the program starts out serially in a single CPU thread. The CPU thread spawns multiple threads, but this time it spawns CPU threads corresponding to a parallel outer loop or region that gets scheduled on multiple CPU cores. These CPU threads execute parallel code potentially containing data-dependent control statements (or possibly other irregular code structures) that can perform poorly or not run at all on GPUs. Each CPU thread then reaches a nested regular parallel loop and spawns the corresponding GPU kernel that gets scheduled on the GPU. Such nested MIMD-SIMD parallelization increases the performance of heterogeneous microprocessors in two possible ways. First, the multiple dynamic instances of the regular parallel loops provide more parallelism to boost the GPU's utilization, both spatially and temporally. And second, portions of the nested parallel regions outside of the regular GPU loops (W_{MIMD}) that would have otherwise executed serially now execute in parallel on the CPU cores, addressing Amdahl's law.

Nested MIMD-SIMD parallelization launches multiple simultaneous kernels onto the GPU to address the small W_{SIMD} and/or small *nthreads* problems; however, it also incurs the overhead for those kernel launches. In some cases, it may be possible to merge multiple kernels together to reduce launch overhead. This requires synchronizing the CPU threads (e.g., via a barrier) so that they all arrive at their respective kernel launches at the same time, and then having one CPU thread launch a single larger kernel that contains the work of all of the individual smaller kernels. (If the CPU threads execute a parallel outer loop, we could perform loop fission on that outer loop to create a separate loop containing the kernel launches, which could then be replaced by a single launch of a merged kernel.) But this optimization introduces other problems. It only works for parallel outer loops; it incurs synchronization overhead (e.g., load imbalance at the barrier), and it may increase the total memory footprint (see Section 4 on memory overhead of loop fission). In this work, we do not employ this optimization but instead use other techniques to mitigate kernel launch overhead.

3.1 Code Examples

We studied many programs to look for opportunities for nested MIMD-SIMD parallelization, and to understand their parallel code structures and characteristics. Our investigation focuses on OpenMP programs [9]. OpenMP is used by programmers to express parallelism for both CPUs and GPUs. (For the latter, OpenMP supports offload directives.) In this work, we focus on OpenMP programs for CPUs, so the parallel code we study contains *half* of the nested parallelism that we seek (parallelism for the CPU). Moreover, OpenMP is one of the most popular parallel programming environments, so there exist numerous OpenMP programs. Many of these programs are more complex in nature and contain loop nests that are challenging to accelerate via GPUs.

Figure 2 presents three examples from our code study: MD and FFT6 from the OpenMP source code repository [26], and 330.art from the SPEC OMP 2001 benchmark suite [27]. In the figure, we show an explicitly parallelized code region from each benchmark (i.e., the code identified by the "#pragma omp" directive). In all of the examples from Figure 2, these directives are for parallel outer loops. We treat these as the parallel outer regions that are the source of CPU parallelism for our parallelization technique. We also show all of the loops nested within the parallel outer loop, labeling each inner loop and indicating its iteration count. (For the inner-most loops in 330.art—i.e., in the "compute_values_match()" function from Figure 2(c)—we only show the loop labels in place of the full code due to the large size of this example.)



```
#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]);
}
```

```
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;
      }
}
```

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

void compute_values_match( ... ) {
  LOOP14 { }, 10000 iter
  LOOP15 { }, 10000 iter
  LOOP16 {  , 10000 iter
    LOOP17 { }, 1000 iter
  }
  LOOP18 { }, 10000 iter
  LOOP19 {  , 1000 iter
    LOOP20 { }, 10000 iter
  }
  LOOP21 { }, 1000 iter
}
```

Fig. 2. Code examples from the MD (a), FFT6 (b), and 330.art (c) benchmarks exhibiting opportunities for nested MIMD-SIMD parallelization. Our technique schedules the parallel outer loops (OpenMP pragmas) on CPUs and the parallel inner loops (shaded boxes) on GPUs.

The nested loops in Figure 2 exhibit several interesting characteristics. First, they are smaller loops with low iteration counts. Some loops have a trivial number of iterations (i.e., ≤ 11), whereas others contain modest iteration counts (i.e., hundreds to a few thousand iterations). Later in Section 6, we will simulate a GPU that has more than 24,000 hardware thread contexts. If these loops were to be converted into kernels and executed on this GPU, none of them would provide enough parallelism to fully utilize the GPU. (In Section 7, we will consider a smaller physical GPU with 5,000 hardware thread contexts. Even for this more modest GPU, some of the loops in Figure 2 would still lack sufficient parallelism.) This is the small *nthreads* problem alluded to in Figure 1.

The nested loops in Figure 2 exhibit small iteration counts, but some of them also exhibit irregularity that would make GPU execution challenging. A GPU's SMs implement a *single instruction multiple thread* (SIMT) execution model. Under the SIMT model, threads are executed in groups, called *warps*. Although threads within a warp are allowed to execute different control paths, for maximum performance, intra-warp threads should follow the same control path. Some loops from our code examples exhibit control divergence that could degrade performance (e.g., LOOP2 and LOOP7 in Figure 2(a) and (b)). Moreover, although 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 2(b)). Along with the small *nthreads* problem, these irregular code characteristics can also limit the acceleration that our programs can get from GPUs.

Our code examples in Figure 2 also exhibit complex loop nest structure. There can be a large number of nesting levels, and nesting can occur across multiple function calls. FFT6 and 330.art are examples of this, with the latter providing an extreme example of code distributed across a large number of deeply nested loops. Within such complex loop structures, there are often significant amounts of computation occurring outside of the inner-most loops that cannot be off-loaded to the GPU. For example, in both FFT6 and 330.art, there are data-dependent control statements (the "if" and "while" statements in Figure 2(b) and (c), respectively) that would cause performance-degrading control divergence if the outer loops were executed on the GPU. In the case of 330.art, the control statement changes the number of times the inner loops execute, which would result in severe control divergence. These outer-loop computations constitute the W_{MIMD} from Figure 1 and, if executed serially on the CPU, would limit performance due to Amdahl's law. This is the $W_{MIMD} \approx W_{SIMD}$ problem alluded to in Figure 1.

But regardless of the loop characteristics, the key factor for our technique is the presence of nested parallel loops. In particular, along with the parallel outer loops that are explicitly identified by the OpenMP pragmas in Figure 2, our technique requires one or more of the nested loops to be parallel as well.[1] A careful examination of the nested loops reveals that a number of them are indeed parallel. (Although not shown in Figure 2(c), all of the inner loops in 330.art, labeled LOOP14–LOOP21, are parallel). We find that this is fairly common in OpenMP programs because the marked parallel loops often appear at the outer levels of deep loop nests. Such coarse-grained parallelism is difficult for a compiler to extract but is natural for a programmer to express given his or 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. 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.

Both CPU and GPU can be leveraged to exploit the nested parallelism in Figure 2. Inner loops with non-trivial iteration counts have a greater chance for GPU acceleration (even if they exhibit control and memory divergence) and are good candidates for off-loading to the GPU. In contrast, outer loops are more appropriate for the CPU. As discussed earlier, by mapping outer loops to multiple CPU cores, we generate several instances of the inner loops for multi-kernel off-loading, thus overlapping the kernels in space (if they have a small *nthreads*) and in time. We also parallelize the (potentially irregular) outer loop code to address the $W_{MIMD} \approx W_{SIMD}$ problem and Amdahl's law.

Notice that uncovering nested parallelism is not sufficient. To be successful, our parallelization technique must also carefully select which parallel loops to exploit. Because the parallelism for the CPU comes from an OpenMP pragma, it is safe to always assume that scheduling the outer parallel loop/code onto the CPU will be profitable. However, not all inner loops can be profitably scheduled onto the GPU. Because of the small *nthreads* and irregular code problems discussed earlier, some inner loops may in fact exhibit slowdown on the GPU. Developing tools to assess profitability (e.g., cost models) would be useful. Assessment could be performed either statically or dynamically. For example, the latter may employ inspector-executor techniques to decide at runtime whether or not to off-load. Such techniques could save programmer effort and also provide the benefit of adapting to changes in runtime behavior. In this work, we manually determine which inner loops to schedule onto the GPU, and we do not try to adapt the decision at runtime.

3.2 Speedup Analysis

The speedup resulting from nested MIMD-SIMD parallelization comes from both the CPU and GPU working together. For instance, let us consider an OpenMP parallel region with serial execution time, T. Assume that some fraction of the serial execution, f_{SIMD}, is spent in one or more regular loops that can be off-loaded onto the GPU with speedup of S_{SIMD}. (f_{SIMD} is related to $\frac{W_{SIMD}}{W_{SIMD}+W_{MIMD}}$ from Figure 1 but is based on execution time rather than computational work.) Further assume that when the OpenMP region is executed in parallel on multiple CPU cores (without the GPU), it achieves a speedup of S_{MIMD}. Then, the speedup from nested MIMD-SIMD

[1]By parallel, we mean that the nested loop either has no cross-iteration dependences or it computes a reduction.

parallelization can be estimated as follows:

$$Speedup = \frac{T}{\left(\frac{T\times(1-f_{SIMD})+\frac{T\times f_{SIMD}}{S_{SIMD}}}{S_{MIMD}}\right)} = \frac{S_{MIMD}}{(1-f_{SIMD})+\frac{f_{SIMD}}{S_{SIMD}}}. \qquad (1)$$

Notice that in the first line of Equation (1), the GPU speedup only applies to the regular loops from the OpenMP region, so S_{SIMD} appears as part of the familiar Amdahl's law equation. Then, the CPU speedup, S_{MIMD}, is applied to the entire GPU-accelerated execution time.

Equation (1) is an upper bound on speedup from parallel execution on both the CPU and GPU because it neglects the overheads that can result from the parallelization. In particular, although simultaneously off-loaded kernels from multiple CPU threads can boost the GPU's utilization, it can also cause contention for hardware resources. For example, two kernels with *nthreads* greater than the number of hardware thread contexts would cause contention for execution resources in the GPU if launched simultaneously. In addition, multiple kernels with a large combined working set may cause contention for cache resources in the memory hierarchy. In our evaluation, we will see examples of such parallel overheads. Nevertheless, the preceding equation still provides valuable intuition on the expected gains.

4 ENABLING FURTHER SIMD PARALLELIZATION

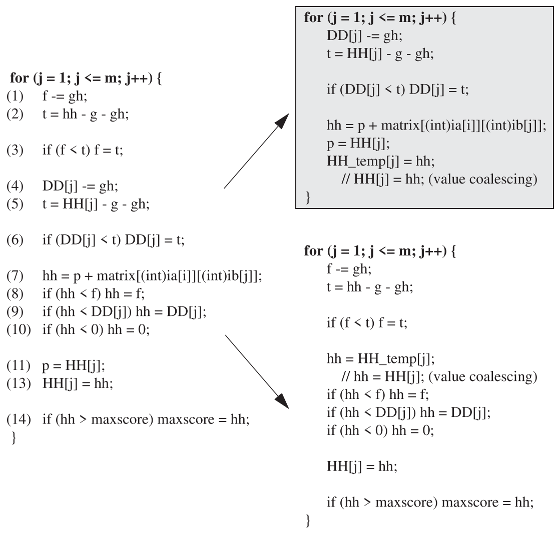
As discussed in Section 3, one of our goals is to identify regular loops nested inside OpenMP parallel regions for execution on the GPU. Although OpenMP parallel regions can be quite extensive, and hence contain many candidate loops, there may still be regions for which nested data parallel loops do not exist. Indeed, we do find that OpenMP parallel regions often contain nested loops that are serial.[2]

However, although a particular loop may be serial in its original form, there may still be regular data parallel computations buried inside the otherwise serial loop. One reason for this is because arrays are ubiquitous, and computations performed on them are often SIMD in nature (i.e., the same operation is performed on every array element). Such computations appear even inside serial loops. Hence, it may be possible to extract regular parallel loops from serial loops by separating their SIMD and serial portions.

Previously, we discussed the possibility of performing *loop fission* on parallel outer loops (see Section 3). In this section, we use loop fission to extract regular data parallel computations within serial loops that are nested within OpenMP parallel regions. This can create more regular loops and thus greater opportunities for GPU acceleration, especially for complex programs that exhibit irregular code. More specifically, for a given serial loop, we identify code statements that do not participate in the loop-carried dependences within the loop body and group them together (if they exist), isolating them from the other code statements that are serialized across loop iterations. Then, we replace the original serial loop with two new loops: one that is parallel containing the data parallel code statements, and another that is serial containing the remaining code statements. When forming the new loops, we copy the loop header from the original loop so that the same number of iterations is performed as before. In general, loop fission has been used to extract parallelism from non-parallel loops [28], sometimes for vectorization purposes [29]. We use a similar technique to extract code that could be off-loaded onto GPUs. In our work, the steps for implementing our loop fission optimization are performed manually.

Figure 3 illustrates our loop fission transformation. In the figure, we show a code example taken from the 358.botsalgn benchmark, which is part of the SPEC OMP 2012 benchmark suite [30]. The

[2]By serial, we mean that the nested loop has cross-iteration dependences.



```
for (j = 1; j <= m; j++) {
(1)    f -= gh;
(2)    t = hh - g - gh;

(3)    if (f < t) f = t;

(4)    DD[j] -= gh;
(5)    t = HH[j] - g - gh;

(6)    if (DD[j] < t) DD[j] = t;

(7)    hh = p + matrix[(int)ia[i]][(int)ib[j]];
(8)    if (hh < f) hh = f;
(9)    if (hh < DD[j]) hh = DD[j];
(10)   if (hh < 0) hh = 0;

(11)   p = HH[j];
(13)   HH[j] = hh;

(14)   if (hh > maxscore) maxscore = hh;
}
```

```
for (j = 1; j <= m; j++) {
    DD[j] -= gh;
    t = HH[j] - g - gh;

    if (DD[j] < t) DD[j] = t;

    hh = p + matrix[(int)ia[i]][(int)ib[j]];
    p = HH[j];
    HH_temp[j] = hh;
      // HH[j] = hh; (value coalescing)
}
```

```
for (j = 1; j <= m; j++) {
    f -= gh;
    t = hh - g - gh;

    if (f < t) f = t;

    hh = HH_temp[j];
      // hh = HH[j]; (value coalescing)
    if (hh < f) hh = f;
    if (hh < DD[j]) hh = DD[j];
    if (hh < 0) hh = 0;

    HH[j] = hh;

    if (hh > maxscore) maxscore = hh;
}
```

Fig. 3. Loop fission performed on an inner loop from the 358.botsalgn benchmark. The transformation produces a regular parallel loop (shaded box) that can be off-loaded to the GPU and a serial loop that remains on the CPU.

left half of the figure illustrates one of the inner loops nested inside an OpenMP parallel region as it appears in the benchmark code. This loop computes the results for two arrays: "DD" and "HH." The computation of the "DD" array (lines #4 and #6 of Figure 3) can be performed in parallel. The computation of the "HH" array, however, in general cannot. Part of the "HH" array's computation (line #7), which produces its initial value as the temporary scalar "hh," is parallel. But the conditional update of that initial value (line #8) is serial. The problem with this conditional statement is that it depends on "f," which requires consideration of all previously computed "hh" values (lines #2 and #3). So taken as a whole, this loop is serial even though it contains parallel portions.

The right half of Figure 3 illustrates the two loops created by loop fission. In the top-right corner of the figure, we show one of the loops (shaded gray) that contains all of the parallel computations from the original loop. In particular, it contains the computation of the "DD" array and the initial computation of the "HH" array. This shaded loop exhibits regular data parallelism and can be off-loaded onto the GPU. In the bottom-right corner of the figure, we show the other loop that contains the remaining code from the original loop. This loop is serial and must execute on the CPU.

Notice that the second loop depends on the first loop since the SIMD computation only performs part of the "HH" array's computation, which must then be completed by the serial loop. All of the temporary "hh" values produced by the SIMD loop must be communicated to the serial loop for this to happen. One way to achieve this is to create a temporary array, referred to as "HH_temp" in Figure 3, to facilitate the required communication. Unfortunately, this can increase the total data footprint compared to the original loop. Previous work [31] has proposed *value coalescing*, which can eliminate this extra memory overhead. The idea is to utilize free array elements (when available) to communicate temporary values rather than to create a dedicated array. For example, in Figure 3, each HH array element ("HH[j]") is dead after the last read to the array in the SIMD loop. Thus, it can be used to perform the communication, eliminating the need for the "HH_temp" array. The two commented lines of code in Figure 3 illustrate this value coalescing optimization.

Value coalescing cannot be applied all the time, but when possible, it can reduce memory overhead. In cases where temporary arrays are still needed, it is possible for the added overhead to

Table 1. Simulation Parameters Used in the Experiments

CPU		GPU	
Number of cores	4	Number of SMs	16
Clock rate	2.6 GHz	Clock rate	900 MHz
Issue width	8	Number of SPs per SM	32
Issue queue size	64	Warp size	32
Reorder buffer size	192	Maximum warps per SM	48
L1 I/D cache (private per core)	32 KB/64 KB	L1 cache (private per SM)	128 KB
L2 cache (private per core)	2 MB	L2 cache (shared by all SMs)	2 MB
Main Memory			
Each channel			64-bit, 1.848 GHz, DDR3
Total bandwidth, 4 channels			110 GB/s

The modeled heterogeneous microprocessor resembles an Intel integrated chip containing a Core i7 CPU and an Iris Pro Graphics 580 GPU.

outweigh the benefits of loop fission. For our benchmarks in Section 5.2, the loop fission optimization was profitable in all of the cases we considered. But in general, assessing its profitability may be necessary.

5 SIMULATION METHODOLOGY

The rest of this article conducts a quantitative evaluation of our parallelization techniques. We begin with a simulation-based evaluation using the gem5-gpu simulator [10, 32]. The goal of this simulation study is to provide deep insights into the behavior of our techniques. Later, in Section 7, we will present a physical machine evaluation to demonstrate that our techniques also work in actual systems.

5.1 Simulator

We configure the gem5-gpu simulator as shown in Table 1. In particular, we model a multi-core CPU with four out-of-order superscalar cores and a two-level cache hierarchy. We assume an integrated GPU with 16 SMs, each comprised of 32 SPs. Each SM can support up to 48 warps, with each warp containing 32 threads. Across the entire integrated GPU, there are a total of 24,576 concurrent threads. Associated with each SM is an L1 cache backed by an L2 cache shared by all of the SMs in the GPU. Our gem5-gpu simulator uses the "fusion" configuration in which a unified shared address space is visible to all of the cores from both the CPU and GPU, and all of the cores share a DDR3 main memory system. Our simulator runs C and CUDA code after compilation for the CPU and GPU, respectively. (CUDA was chosen over OpenCL because the OpenCL available for the simulator does not include shared virtual memory support, which is required for the fusion configuration.)

In addition, our gem5-gpu simulator also employs a MOESI cache coherence protocol to maintain coherence. Coherence is enforced between the CPU's private caches, and the GPU's L2 cache also participates in the coherence protocol.[3] Together, the simulated configuration specified in Table 1 along with the cache coherence protocol resembles Intel heterogeneous microprocessors—for example, a Core i7 CPU integrated with an Iris Pro Graphics 580 GPU [33]. This is a reasonably sized heterogeneous microprocessor and is more capable than the physical system we consider in Section 7.

[3]The GPU's L1 caches are write through but are not kept coherent. They are flash invalidated between kernel launches.

Table 2. OpenMP Programs Used in the Experimental Evaluation

Benchmark	Suite	Input	# SIMD	Lines Changed	Sim (Billion Instructions)
MD	OMP repo	4,096	1	69	1.65
FFT6	OMP repo	8,192	1	60	1.83
330.art	OMP2001	ref	4	158	3.15
358.botsalgn	OMP2012	ref	2	191	4.82
359.botsspar	OMP2012	ref	3	92	60.40
367.imagick	OMP2012	ref	1	38	21.40
372.smithwa	OMP2012	ref	1	68	1.47

One issue with the gem5-gpu simulator is that it lacks support for multiple GPU kernels launched simultaneously from different CPU threads, which our techniques exploit. In particular, the CUDA library that comes bundled with the simulator is not thread safe. In addition, although the original GPGPU-Sim simulator [34] (upon which gem5-gpu's GPU model is based) can support multiple kernels, gem5-gpu only schedules a single kernel onto the GPU at a time. We modified both the CUDA library and the kernel scheduler within the simulator so that multiple concurrent kernels can be supported properly.

Last, as alluded to earlier, kernel launch overhead affects the performance of our parallelization techniques. Thanks to a unified address space and the hardware coherence mechanisms provided by gem5-gpu's "fusion" configuration, CPU-GPU data movement in our simulated platform exhibits very low overhead. However, software overhead associated with queuing and scheduling kernels onto the GPU can still be costly. This overhead comes from both the user-level CUDA API mentioned earlier and from executing system calls to access the GPU's hardware interfaces.

In our experiments, we run the gem5-gpu simulator in system call emulation mode and introduce an artificial delay to the emulated system call handler. We set the handler delay such that the total latency for launching a kernel is 4.5 μs. This corresponds to a fairly aggressive launch latency for a heterogeneous microprocessor that has been reported in the literature [35]. We also consider lower launch latencies to see the impact on benchmarks that are highly sensitive to launch overhead.

5.2 Benchmarks

Using our modified gem5-gpu simulator, we evaluate nested MIMD-SIMD parallelization. We surveyed several OpenMP benchmarks across different benchmark suites to identify those benchmarks that exhibit the nested parallel structures our technique targets. About half the benchmarks we surveyed exhibited such nested parallel structures. This yielded seven benchmarks, which are listed in Table 2. The benchmarks are MD and FFT6 from the OpenMP source code repository; 330.art from the SPEC OMP 2001 suite; and 358.botsalgn, 359.botsspar, 367.imagick, and 372.smithwa from the SPEC OMP 2012 suite. Input sizes of $N = 4{,}096$ particles and $N = 8{,}192$ points were used for MD and FFT6, respectively. For all of the SPEC benchmarks, the ref inputs were used.

As discussed in Section 5.1, our experiments on gem5-gpu are run in system emulation mode. Under system emulation, there is no direct support for OpenMP. Thus, we manually ported the OpenMP pragmas appearing in our benchmarks to pthreads and then linked the pthread code against the M5 threads library [36], which provides pthreads compatibility in system emulation mode. We did this for the most important OpenMP region from each benchmark. (In many benchmarks, there is only one OpenMP region.) Most of the time, the OpenMP region we ported is a loop with load-balanced iterations. For these benchmarks, we statically partitioned the loops across the M5 threads. One exception is 359.botsspar, which performs blocked LU decomposition on a sparse matrix. If different parts of the matrix are statically assigned to threads, then load imbalance can occur depending on the fill pattern of the matrix. For 359.botsspar, we instead created a work queue and dynamically distributed work to the M5 threads. (This emulates the OpenMP code for 359.botsspar, which uses dynamic scheduling across the threads.) The outer parallel region in 359.botsspar is the most irregular across all of our benchmarks and is the closest to exhibiting task-level parallelism.

After creating the pthreads versions of our benchmarks, we examined the loops nested inside the original OpenMP parallel regions to find parallelism for the GPU. In five benchmarks, we were able to identify at least one and as many as four nested GPU-friendly loops. (The profitability of off-loading these inner loops was assessed manually.) The shaded boxes in Figure 2 show the identified loops for three of these benchmarks. In the case of 358.botsalgn and 372.smithwa, we found serial loops within the original OpenMP parallel regions; however, we were able to extract regular parallel loops from these serial loops using our loop fission transformation from Section 4. This transformation was performed by hand. (In addition, value coalescing—as discussed in Section 4—was applied to 358.botsalgn, but it was not possible to apply to 372.smithwa.) The shaded box in Figure 3 shows the extracted loop for 358.botsalgn. In Table 2, the column labeled "# SIMD" reports the number of regular data parallel loops identified for each benchmark, including those uncovered via loop fission.

Once these regular parallel loops were identified, we created GPU kernels out of every one of them by porting them to CUDA. Then, we compiled our C/CUDA benchmarks to create binaries for the gem5-gpu simulator. We used GCC to generate the CPU code and NVCC to generate the GPU code. In both cases, we compiled with the highest level of optimization possible. The "Lines Changed" column in Table 2 reports the lines of CUDA and C code added to the OpenMP region.

Finally, we ran the compiled benchmarks on our modified gem5-gpu simulator. We first fast forwarded past the initialization code to reach the benchmark's main computation, which is always the parallel OpenMP region we ported for each benchmark. After fast forwarding, we turned on detailed simulation and gathered statistics in the OpenMP region. The last column of Table 2 reports the number of instructions (in billions) executed during detailed simulation.[4]

For MD and 367.imagick, we were able to simulate the entire OpenMP region, but for the other benchmarks, the OpenMP regions were too large to simulate to completion. Instead, we chose a representative portion of each OpenMP region to simulate. For FFT6, 330.art, 358.botsalgn, and 372.smithwa, the OpenMP regions consist of a parallel outer loop that performs very similar computations across all loop iterations. Thus, we just simulated several iterations from the beginning of the parallel loop. In 359.botsspar, the OpenMP region performs partial pivoting and sub-matrix elimination underneath the pivot element. Unfortunately, the computation is very different for different pivots, especially affecting the relative amounts of work performed by the CPU versus the GPU. We identified the pivot for which the ratio of CPU-to-GPU computation is representative of the entire OpenMP region and simulate the computation associated with that pivot element only.

6 SIMULATION RESULTS

Figure 4 presents the main results from our simulator-based evaluation. It compares different parallelization schemes. We begin by examining parallelization for the CPU only (i.e., following the OpenMP pragmas that the benchmarks already come with). The bars labeled "MIMD" in Figure 4 report the execution time of these OpenMP pragmas (after translation to pthreads) assuming four

[4]These instruction counts correspond to executing the simulated portion of the OpenMP region on a single CPU thread.

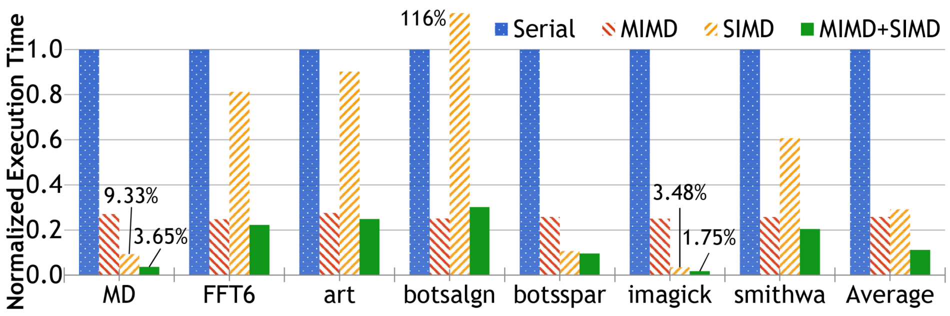


Fig. 4. Normalized execution time for no parallelization (Serial), CPU parallelization (MIMD), single-loop SIMD parallelization (SIMD), and nested MIMD-SIMD parallelization (MIMD+SIMD).

Table 3. S_{MIMD}, S_{SIMD}, and f_{SIMD} Parameters for Our Benchmarks

Bench	S_{MIMD}	S_{SIMD}	f_{SIMD}	UpperB	Actual	CPU	GPU
MD	3.70	9.19	0.99	33.9	27.4	0.12	0.43
FFT6	4.03	2.85	0.32	5.09	4.63	0.71	0.038
art	3.63	52.1	0.12	4.09	4.01	0.99	0.0031
botsalgn	3.99	2.72	0.38	5.24	4.13	0.67	0.011
botsspar	3.87	9.40	0.98	30.5	10.5	0.12	0.40
imagick	4.00	22.4	0.99	85.2	57.1	0.018	0.98
smithwa	3.90	22.3	0.55	8.19	4.89	0.95	0.16
average	3.87	17.3	0.62	24.6	16.1	0.51	0.29

The "UpperB" and "Actual" columns report upper-bound and actual speedup, respectively, of nested MIMD-SIMD parallelization. The last two columns report CPU and GPU utilization. (Parameters for FFT6 and botsalgn reflect the results shown in Figure 5).

CPU threads. These execution times are normalized to each benchmark's sequential running time on one CPU thread (i.e., the 1.0 bars labeled "Serial" in Figure 4). The column labeled "S_{MIMD}" in Table 3 reports the speedup that the "MIMD" bars achieve over the "Serial" bars.

Comparing the "MIMD" and "Serial" bars in Figure 4, we see that parallelization for the CPU provides consistent performance gains: most of the MIMD bars are near 0.25. As Table 3 shows, S_{MIMD} is between 3.63x and 4.03x. (FFT6 achieves a slight superlinear speedup due to the increased cache capacity that comes with scaling.) On average, the CPU speedup is 3.87x. These results show that the original OpenMP parallelization effectively exposes the outer-loop parallelism in the benchmarks and achieves good CPU performance.

Next, we consider parallelization schemes that off-load the inner loops we identified within OpenMP regions onto the GPU. In particular, the bars labeled "SIMD" in Figure 4 report the normalized execution time when off-loading such nested regular parallel loops one at a time from a single CPU thread (i.e., the single-loop SIMD parallelization scheme).

Figure 4 shows that single-loop SIMD parallelization also performs well. In fact, its performance gain is similar to the gain achieved by OpenMP parallelization, providing a 70.8% reduction in execution time and a 3.43x speedup as shown by the "Average" bars in Figure 4. But compared to CPU-only parallelization, GPU-only parallelization is not as consistent across individual benchmarks. For MD, 359.botsspar, and 367.imagick, there are very large gains—between 89.3% and 96.5% reduction in execution time compared to the "Serial" bars. For the other benchmarks, however, the gains are more modest and show between 9.8% and 39.3% reduction, with 358.botsalgn actually showing

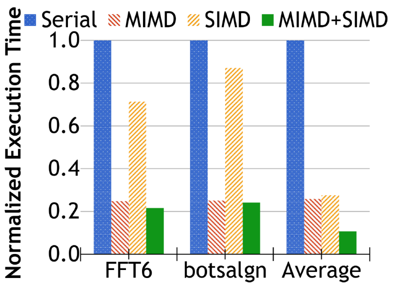


Fig. 5. Normalized execution time assuming user-level kernel launch (0.1-μs latency). The "Average" bars include the other five benchmarks from Figure 4.

an *increase* in execution time of 16.2%. These results demonstrate that exploiting single-loop SIMD parallelization within OpenMP programs is beneficial in general, but the complex nature of these codes can cause the the GPU's performance to vary significantly across different benchmarks.

One reason programs can experience reduced performance gains on the GPU is kernel launch overhead. As discussed in Section 3.1, many of the GPU loops in our OpenMP benchmarks exhibit small W_{SIMD}, so a low-latency kernel launch is crucial. The results in Figure 4 assume our baseline launch latency of 4.5 μs [35]. Although this is already a low latency, unfortunately, it is still too high for FFT6 and 358.botsalgn, the two most latency-sensitive benchmarks in Figure 4 due to especially small W_{SIMD}. Figure 5 shows the performance of these two benchmarks when the kernel launch latency is reduced to a much more aggressive 0.1 μs. To simulate this, we still include the CUDA library launch stub, but we set the system call handler delay in our simulator to zero. The "SIMD" bars in Figure 5 show the gain increases to 28.7% for FFT6 (compared to only 18.8% in Figure 4), and the performance degradation for 358.botsalgn turns into a 12.8% performance gain. (The other benchmarks do not benefit from a 0.1-μs launch latency.)

Finally, the bars labeled "MIMD+SIMD" in Figures 4 and 5 report normalized execution time when the nested GPU loops are off-loaded in parallel from multiple CPU threads simultaneously (i.e., using nested MIMD-SIMD parallelization). The last column in Table 3 labeled "Actual" reports the speedup of the "MIMD+SIMD" bars over the "Serial" bars.

In Figure 4, we see our nested MIMD-SIMD parallelization scheme achieves large gains, providing execution time reductions between 75.1% and 98.3% compared to the "Serial" bars. Specifically, Table 3 reports our nested MIMD-SIMD parallelization scheme provides 27.4x, 10.47x, and 57.1x speedup for MD, 359.botsspar, and 367.imagick, respectively. For 330.art and 372.smithwa, our technique provides 4.01x and 4.89x speedup, respectively. And for FFT6 and 358.botsalgn, we provide 4.50x and 3.32x speedup, respectively. As before, the last two benchmarks are the most sensitive to kernel launch overhead. When using a 0.1-μs launch latency, as is done in Figure 5, these speedups increase to 4.63x and 4.13x, respectively, as reported in Table 3. On average, nested MIMD-SIMD parallelization (assuming 0.1 μs launches for FFT6 and 358.botsalgn) provides a 16.1x speedup over the serial execution time.

Nested MIMD-SIMD parallelization provides large speedups, and it also beats CPU-only and GPU-only parallelization schemes by leveraging parallelism within both types of cores simultaneously. In particular, nested MIMD-SIMD parallelization outperforms CPU-only parallelization (the MIMD bars in Figures 4 and 5) by 4.13x on average, and outperforms GPU-only parallelization (the SIMD bars) by 2.74x on average. Notice that in Figures 4 and 5, nested MIMD-SIMD parallelization is always the best parallelization scheme (except for 358.botsalgn in Figure 4 due to launch latency sensitivity), but the next-best scheme is benchmark dependent. For MD, 359.botsspar, and 367.imagick, the next-best scheme is single-loop SIMD parallelization, but for the other

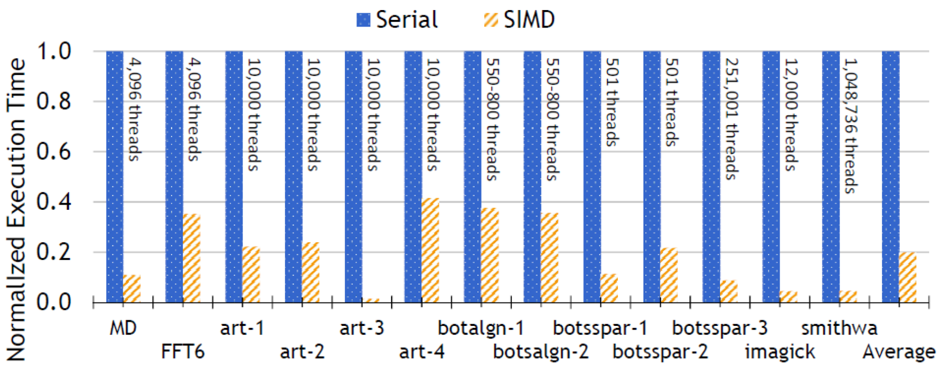


Fig. 6. Normalized execution time for individual off-loaded loops running on a single CPU thread (Serial) and on the GPU (SIMD). Vertical labels indicate *nthreads* for each kernel. (Results for FFT6 and botsalgn's kernels assume a 0.1-μs launch latency.)

benchmarks, the next-best scheme is OpenMP parallelization. Averaged across all benchmarks, our nested MIMD-SIMD parallelization scheme beats the next-best scheme by 45.8%.

6.1 Performance Breakdown

This section presents detailed results to break down and better understand the gains of nested MIMD-SIMD parallelization that were observed in our simulation study. First, Figure 6 isolates the GPU's benefit. Similar to Figures 4 and 5, the "SIMD" bars report execution time under single SIMD loop parallelization, but instead of end-to-end execution time, the "SIMD" bars in Figure 6 report the time individual off-loaded loops spend on the GPU. These bars are normalized against the corresponding execution time of the same loops running serially on the CPU—that is, the 1.0 bars labeled "Serial"—so comparing the "SIMD" and "Serial" bars in Figure 6 reveals the *per-kernel performance gain.* (330.art, 358.botsalgn, and 359.botsspar contain multiple kernels, so there is a separate result for each kernel.) In addition, the column labeled "S_{SIMD}" in Table 3 reports the speedup corresponding to the per-kernel performance gains shown in Figure 6. (In benchmarks with multiple kernels, Table 3 reports the time-weighted average per-kernel speedup.) Second, the vertical labels in Figure 6 report *nthreads* from Figure 1 (i.e., the number of threads that occupy the GPU each time the kernel is launched). Last, the fourth column of Table 3 reports f_{SIMD} from Equation (1)—that is, the fraction of end-to-end serial execution time that is spent in the off-loaded loop(s).

These data separate our benchmarks into four categories. The MD and 367.imagick benchmarks comprise the first category. Both MD and 367.imagick contain a single kernel for which the GPU achieves a large gain, as shown in Figure 6. Table 3 reveals that these kernels enjoy a 9.19x and 22.4x speedup, respectively. Moreover, Table 3 also shows that virtually all of the time is spent in these kernels: f_{SIMD} = 0.99 in both benchmarks. This is why the two benchmarks' "SIMD" bars in Figure 4 perform so well: the large per-kernel speedup translates directly into end-to-end performance gains because of the large f_{SIMD}. Figure 6 also reports that each MD and 367.imagick kernel has $nthreads$ = 4,096 and 12,000, respectively. In comparison, our simulated GPU from Table 1 can support up to 24,576 threads, so it can accommodate multiple invocations of these kernels simultaneously. Because f_{SIMD} is so large, when we parallelize the outer OpenMP loop on the CPU, there is very little delay between when a kernel finishes and when its associated CPU thread launches the next instance of the kernel. Hence, there are almost always multiple kernels running simultaneously on the GPU. For 367.imagick, we expect two simultaneous kernels since its *nthreads* is about half the number of available GPU hardware threads. This is why in Figure 4 the performance of nested MIMD-SIMD parallelization exceeds the performance of single-loop SIMD parallelization by roughly 2x. For MD, a smaller *nthreads* (4,096) would ostensibly permit the maximum four kernels (launched from the four CPU threads) to run on the GPU. Unfortunately, the high frequency of kernel launches in MD creates contention for the GPU's hardware launch queue that limits the number of simultaneous kernels. Nevertheless, Figure 4 still shows a 2.58x speedup for nested MIMD-SIMD parallelization over single-loop SIMD parallelization.

FFT6, 330.art, and 358.botsalgn comprise the second benchmark category. In this category, Figure 6 shows that the per-kernel gains are mixed. 330.art exhibits very strong per-kernel performance (kernel art-3, which is 330.art's most dominate kernel by far, achieves a 76.4x speedup), but the other two benchmarks exhibit more modest per-kernel performance. Table 3 shows that 358.botsalgn's per-kernel speedup is 2.72x (averaged over its two kernels, which achieve 2.66x and 2.81x speedup), whereas FFT6's per-kernel speedup is 2.85x.

Another issue is that Table 3 also shows that these benchmarks' f_{SIMD} is small, between 0.12 and 0.38. Both FFT6 and 330.art exhibit smaller f_{SIMD} due to significant amounts of code outside of their inner SIMD loops. For 358.botsalgn, we used loop fission to expose the nested SIMD loops. Loop fission tends to result in smaller f_{SIMD} because it introduces a serial loop executed on the CPU that is comparable in size to the SIMD loop it exposes for the GPU. Due to their smaller f_{SIMD}, the per-kernel gains for FFT6, 330.art, and 358.botsalgn are diminished by Amdahl's law. This is why these benchmarks exhibit the lowest single-loop SIMD performance in Figures 4 and 5. On the bright side, nested MIMD-SIMD parallelization still performs well. Because f_{SIMD} is small, these benchmarks spend a lot of time in the outer OpenMP loops, so they benefit significantly from CPU parallelization. On top of that, the GPU also contributes some gain. Similar to MD, Figure 6 shows that these benchmarks have an *nthreads* (between 550 and 10,000) that is smaller than what the GPU can handle. In addition, the smaller f_{SIMD} results in a lower kernel launch frequency. Thus, the GPU has the capacity, both spatially and temporally, to support kernel launches from multiple CPU threads.

In the third category, we have the 372.smithwa benchmark, which has some characteristics in common with benchmarks in the second category: a large per-kernel speedup, 22.3x—as shown in Table 3—that is diminished by a smaller f_{SIMD}, 0.55, which also is shown in Table 3. (Like 358.botsalgn, 372.smithwa is a benchmark for which we used loop fission, which is why $f_{SIMD} = 0.55$.) However, 372.smithwa's *nthreads* is more than 1 million, as its label in Figure 6 reports. Therefore, even a single kernel can fill all of the GPU's threading contexts. For nested MIMD-SIMD parallelization, this means multiple CPU threads will experience some serialization as they off-load kernels that are too big to execute simultaneously on the GPU. Nevertheless, we find that the CPU threads can still overlap a portion of the code they execute, providing some gain. In addition, the performance gain of single-loop SIMD parallelization in 372.smithwa is non-trivial (although on the small side, 372.smithwa's f_{SIMD} is still larger than those in benchmark category two, reducing the Amdahl's law impact). This is why the performance of nested MIMD-SIMD parallelization for 372.smithwa is quite good in Figure 4.

Finally, 359.botsspar comprises the last benchmark category. Similar to MD, Figure 6 shows significant per-kernel gains for 359.botsspar, and Table 3 shows that $f_{SIMD} = 0.98$. But like 372.smithwa, one of its kernels, botsspar-3, has enough threads (251K) to fully utilize the GPU by itself. The problem is that this kernel dominates the other two kernels (which only launch 501 threads each), accounting for 85% of the execution time. Although multiple botsspar-1 and botsspar-2 kernels can overlap in the GPU, there is only a small opportunity for this because most of the time is spent in the botsspar-3 kernel. For 359.botsspar, the main thing that matters is GPU performance on individual botsspar-3 kernels. This is why nested MIMD-SIMD parallelization achieves very little gain over single-loop SIMD parallelization in Figure 4.

Upper-Bound Speedup. From the S_{MIMD}, S_{SIMD}, and f_{SIMD} values in Table 3, we compute the upper-bound speedup for each benchmark using Equation (1). This result is shown in the column labeled "UpperB" in Table 3. Comparing the "UpperB" and "Actual" columns of Table 3, we confirm that the upper-bound speedup is always higher than the actual speedup. As mentioned, Equation (1) does not model all performance-degrading effects. For instance, it neglects contention for the GPU's hardware resources, particularly for multiple kernels trying to run simultaneously. Our results show that as long as this contention is small, Equation (1) is a good predictor of actual speedup. In the FFT6, 330.art, and 358.botsalgn benchmarks, this assumption holds true. For MD, there is some contention as discussed earlier, but still, the benchmark is able to have close to three simultaneous kernels. Thus, the average error between the upper-bound and actual speedups for these four benchmarks is only 15.6%. In the case of 359.botsspar and 372.smithwa, significant contention for GPU execution resources occurs. As discussed earlier, only a single instance of the large kernel from these two benchmarks (kernels botsspar-3 and smithwa from Figure 6) can run simultaneously. Although the situation is better for 367.imagick, still only two instances of this benchmark's kernel can run simultaneously. Therefore, there is a large stall component (CPU threads waiting for the GPU) that is not modeled in Equation (1). As a result, we see large errors in Table 3 of 192%, 67.5%, and 49.2% when comparing the upper-bound and actual speedups for 359.botsspar, 372.smithwa, and 367.imagick, respectively.

6.2 Processor Utilization and CPU Scaling

The last two columns of Table 3 report how nested MIMD-SIMD parallelization utilizes the heterogeneous microprocessor. In particular, the "CPU" column reports the fraction of time that the CPU is busy averaged over the four CPU cores; the "GPU" column reports the fraction of time that the GPU is busy averaged over the 16 SMs, but on any given busy cycle, it also accounts for fractional occupancy of the available hardware threading contexts. Because execution on the CPU and GPU alternates, and because the GPU can be spatially underutilized, the sum of the two statistics is usually less than or equal to 1.0. (But as 372.smithwa shows, exceptions can occur—e.g., one stalled CPU thread could keep the entire GPU busy.) These results loosely follow f_{SIMD}: when f_{SIMD} is large, GPU utilization is higher and CPU utilization is lower, whereas when f_{SIMD} is small, the opposite is true. (The relationship is not perfect because f_{SIMD} is computed over the serial execution, whereas utilization is computed over the parallel execution and includes GPU spatial underutilization.) Because f_{SIMD} varies across our benchmarks, so do the utilization results. In general, however, the results show that the CPU is more fully utilized than the GPU on the benchmarks we studied.

Given these results, we tried increasing the number of CPU cores to eight while keeping the GPU the same. Figure 7 shows the result of this experiment for MD, 367.imagick, and 372.smithwa. In the figure, we plot the same "Serial," "MIMD," and "MIMD+SIMD" bars from Figure 4 (the latter have been renamed to "MIMD_4" and "MIMD+SIMD_4"), but we also plot "MIMD_8" and "MIMD+SIMD_8" which increase the number of CPU cores to eight for the corresponding experiments. (The GPU configuration remains unchanged from Table 1.) Not surprisingly, the MIMD_8 bars in Figure 7 provide roughly another 2x speedup compared to the MIMD_4 bars, further demonstrating the effectiveness of OpenMP parallelization in our benchmarks already discussed for Figure 4. More interestingly, the MIMD+SIMD_8 bars show that our nested MIMD-SIMD parallelization technique continues to scale for both MD and 367.imagick, providing an additional 1.38x and 1.50x speedup, respectively, over MIMD+SIMD_4. Increasing the number of CPU cores speeds up the CPU computation and also increases the off-load frequency. As discussed earlier, the GPU is underutilized for most of our benchmarks, MD included, so the higher off-load rate can be supported, resulting in performance gains. 367.imagick is the only benchmark for which the GPU is already fully utilized

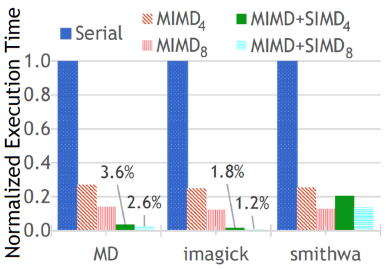


Fig. 7. Impact of scaling to eight CPU cores.

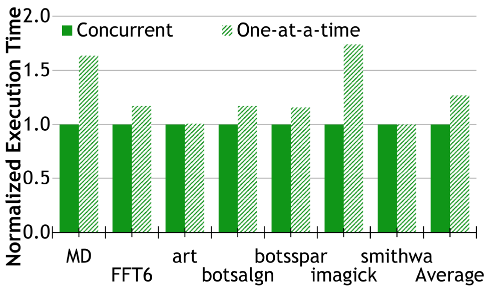


Fig. 8. Impact of concurrent kernel execution.

at four CPU cores (98% busy in Table 3). For 367.imagick, going from four to eight CPU cores does increase the number of simultaneous kernels on the GPU, but because the GPU is fully utilized at four cores, each kernel gets half as many execution resources at eight cores, so we would not expect a net performance gain. However, the larger number of simultaneous kernels increases the GPU's working set size, causing a higher cache-miss frequency and a higher degree of memory latency tolerance. Even though we cannot exploit any more TLP, an improved latency tolerance (MLP) provides the added performance for 367.imagick. Last, for 372.smithwa, the GPU is also underutilized (only 16% busy), so like MD, we do get scaling from MIMD+SIMD$_4$ to MIMD+SIMD$_8$, but the gain is not that large, and the same performance can be achieved by MIMD$_8$. 372.smithwa is one of the benchmarks for which we performed loop fission; consequently, significant communication occurs between the fissioned GPU and CPU loops. At four CPU cores, the communicated data is small enough to fit in the GPU's cache, so the communication mostly happens through the cache coherence protocol. But at eight CPU cores, the communicated data is too large and leaves the GPU's cache, so the communication mostly happens through slower DRAM.

6.3 Concurrent Kernel Execution

When contention for the GPU's execution resources does not stand in the way, nested MIMD-SIMD parallelization can result in multiple kernels being off-loaded onto the GPU simultaneously. We now evaluate for which benchmarks such simultaneous off-loading occurs. The "Concurrent" bars in Figure 8 show the baseline execution time of our nested MIMD-SIMD parallelization scheme. (These bars correspond to the "MIMD+SIMD" bars in Figure 4.) For the "One-at-a-time" bars in Figure 8, we re-run the experiments with concurrent kernel scheduling turned off in our simulator. Thus, if multiple kernels arrive at the GPU simultaneously, we serialize them—that is, each kernel runs to completion before we allow the next kernel to start. All bars are normalized to the baseline with concurrent kernel scheduling turned on.

Figure 8 shows concurrent kernel execution occurs frequently in MD and 367.imagick since one-at-a-time scheduling degrades performance significantly. It occurs to a small degree in 359.botsspar, where one-at-a-time scheduling causes a slowdown of 15.8%. In addition, it hardly occurs in 372.smithwa, as one-at-a-time scheduling makes no difference. As discussed in Section 6.1, 359.botsspar and 372.smithwa are dominated by a single kernel containing *nthreads* that exceeds the number of hardware thread contexts. The large *nthreads* forces serial execution of the dominant kernel regardless of whether the hardware supports concurrent kernel execution or not. (In 359.botsspar, there are two smaller kernels that also serialize under one-at-a-time scheduling. There is some performance impact for doing so, like the 15.8% mentioned, but the impact is relatively small since a majority of the execution time is spent in the dominant kernel.) For FFT6, 330.art, and 358.botsalgn, however, *nthreads* is much smaller. Concurrent kernel execution can

Table 4. Hardware Parameters for the Physical Intel Core i7-6700 with an HD Graphics 530 Integrated GPU

CPU		GPU	
Number of cores	4	Number of SMs	24
Clock rate	3.4–4.0 GHz	Clock rate	350–1,150 MHz
ISA	x86 + AVX512	ISA	GEN Assembly
		Warp size	32
L1 I/D cache (private)	32 KB/32 KB	Max warps per SM	7
L2 cache (private)	256 KB	L1/2 sampler cache (read-only private)	N/A
		L3 I/D cache (shared)	768 KB/1.5 MB
Last-Level Cache (shared by CPU and GPU)			8 MB
Main Memory			
Each channel			64-bit, 2.133 GHz, DDR4

Although the CPU cores have AVX512 instructions, we do not exploit them in our experiments.

occur to a small degree in these benchmarks as well, but again it is much less than in MD and 367.imagick. This is because these benchmarks' f_{SIMD} is small. Small f_{SIMD} results in a lower kernel launch frequency, which reduces the probability that multiple kernels arrive at the GPU simultaneously in the first place. Only in MD and 367.imagick do kernels arrive simultaneously (due to larger f_{SIMD}) and execute simultaneously (permitted by smaller *nthreads*) when concurrent kernel scheduling is turned on. Notice, however, that even though concurrent kernel execution is not dominant in FFT6, 330.art and 358.botsalgn, kernel launch from multiple CPU threads is still important in these benchmarks because it boosts GPU utilization along the temporal dimension, if not along the spatial dimension.

7 PHYSICAL MACHINE STUDY

This section presents our results on real hardware. Although the simulation study enabled us to carefully understand the behavior of our techniques, it does not provide real-world validation of our approach. This section demonstrates our parallelization scheme on an actual system.

7.1 Hardware Parameters

Our physical machine study is performed on an Intel Core i7-6700 whose parameters are listed in Table 4. This processor features four CPU cores with a base frequency of 3.4 GHz and a Turbo Boost frequency of up to 4.0 GHz. The CPU cores each have a split 32-KB L1 cache and a 256-KB L2 cache. The chip also contains an Intel HD Graphics 530 Integrated GPU with 24 SMs (which Intel refers to as "Execution Units"). Each SM supports up to seven warps, with each warp containing 32 threads. Across the entire integrated GPU, there are a total of 5,376 concurrent threads possible, considerably fewer than the 24.5K threads in our simulated GPU. Each SM on the physical machine also exhibits lower throughput: there are only two SPs per SM capable of running only four of the 32 per-warp threads at a time. The GPU cache consists of 1.5 MB of usable L3 shared by all of the SMs. (The GPU's L1 and L2 caches are sampler caches and are unavailable for GPGPU processing.) Coherence between the CPU's L2s and the GPU's L3 exists, but details about the cache coherence protocol are not well documented. Finally, there is an 8-MB LLC (also considered to be the "CPU's L3") that is shared by all cores in both the CPU and GPU.

7.2 Kernel Launch Mechanism

For the simulation study, our GPU kernels were implemented in CUDA (see Section 5.2). But the integrated Intel GPU in our physical machine only works with OpenCL. Hence, we ported all of our benchmark kernels to OpenCL 2.0. Unfortunately, we found that the kernel launch latency using the standard software queue interface in our OpenCL implementation incurs an overhead of at least 100μs. Because the performance of our benchmarks is highly sensitive to the launch latency, we had to address this issue before we could achieve performance gains on the physical machine.

We implemented a GPU launch daemon similar to the "Instant Mode" mechanism proposed in previous work [37]. The launch daemon runs on every available GPU thread and polls for a CPU kernel launch event. When the CPU triggers a kernel launch, the launch daemon executes the kernel, informs the CPU when the kernel is done, and then returns to polling for the next launch event.

The GPU launch daemon communicates with the CPU's cores using OpenCL Shared Virtual Memory buffers and is capable of executing kernels from different CPU threads concurrently as the kernel size and number of available threads allow. During kernel launch, each execution unit is assigned a number of threadblocks to execute, in chunks ranging from one threadblock to the max chunking factor: <total number of threadblocks> divided by <number of SMs>. The more threads and threadblocks in a benchmark, the more benefit there is to larger chunking factors that decrease communication overhead between the CPU cores and the GPU launch daemon.

With our GPU launch daemon, the launch latency reduces from 100 μs down to 14.8μs on average. This is a significant improvement but is still higher than the 4.5-μs baseline (and 0.1-μs aggressive) launch latency assumed for the simulation study. We will discuss implications of this when talking about the benchmarks in the next section.

7.3 Benchmarks

The same benchmarks listed in Table 2 are used for our physical machine study. We ported these benchmarks' CUDA kernels to OpenCL, added our launch daemon mechanism, and compiled the resulting C++/OpenCL code with GCC and Intel's OpenCL SDK using the highest level of optimization possible.

Unfortunately, we had trouble porting 358.botsalgn. This benchmark is the most sensitive to launch latency due to the small size of its GPU kernels. Although we were able to get performance gains on 358.botsalgn with a 0.1-μs launch latency (see Section 6), the 14.8-μs launch latency on our physical machine resulted in significant performance degradation. We were unable to get performance gains on the physical machine for 358.botsalgn by simply translating our CUDA implementation to OpenCL.

Upon closer examination of the 358.botsalgn code, we found a software transformation that can mitigate high launch latency. In each outer loop iteration of 358.botsalgn, multiple kernels are launched to the GPU. These kernels are actually *independent* and can execute concurrently. (This is not generally true across our benchmarks but occurs in 358.botsalgn.) In the CUDA implementation for our simulator, we only exploit concurrent kernels across CPU threads; each CPU thread still executes all of its kernels serially, which exposes the launch latency at every kernel launch. We modified the 358.botsalgn code so that each CPU thread launches multiple independent kernels concurrently, stalling only after all launched kernels complete. This hides much of the launch latency and provides even more kernel-level parallelism for the GPU to exploit. With this software optimization, the single-loop SIMD parallelization scheme for 358.botsalgn speeds up by 2.10x, and our nested MIMD-SIMD parallelization scheme speeds up by 2.32x on the physical machine. In our performance evaluation that follows, we assume that 358.botsalgn uses this software transformation.

7.4 Physical Machine Results

Figure 9 presents the results from our physical machine study. In the figure, the "Serial," "MIMD," "SIMD," and "MIMD+SIMD" bars are identical to the corresponding bars from Figure 4, except

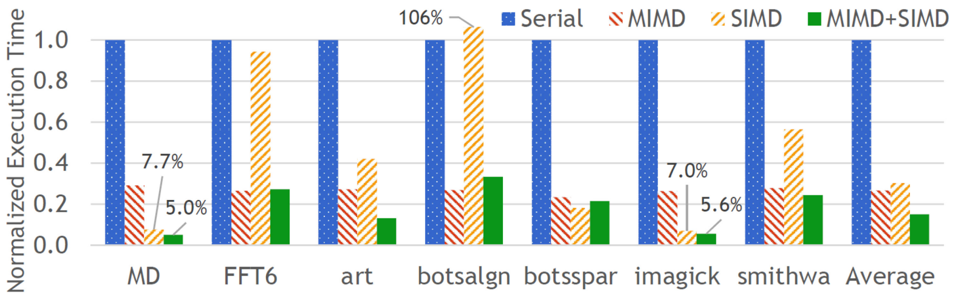


Fig. 9. Normalized execution time for no parallelization (Serial), CPU parallelization (MIMD), single-loop SIMD parallelization (SIMD), and nested MIMD-SIMD parallelization (MIMD+SIMD) on the physical machine.

they report the execution time on the physical machine rather than the simulator. All execution times are normalized to each benchmark's "Serial" bars.

Comparing the "MIMD" and "Serial" bars in Figure 9, we see the same behavior for CPU parallelization as in the simulator study: all of the "MIMD" bars are near 0.25, providing a speedup of 3.76x (compared to 3.87x on the simulator) averaged across all benchmarks. Thus, CPU parallelization is essentially as effective on the real hardware as it is on the simulator.

Figure 9 also shows that single-loop SIMD parallelization (the "SIMD" bars) performs well on the physical machine. Like our simulator results, the performance shown by the "SIMD" bars in Figure 9 varies across the benchmarks. MD, 359.botsspar, and 367.imagick exhibit large gains–between 81.8% and 93.0% reduction in execution time compared to the "Serial" bars. Three of the remaining benchmarks have more modest gains between 5.8% and 58.0%, whereas the last benchmark, 358.botsalgn, actually shows a performance drop of 6.3%. Averaged across all seven benchmarks, the "SIMD" bars in Figure 9 exhibit a 70% reduction in execution time compared to the "Serial" bars. Overall, single-loop SIMD parallelization achieves a 3.30x speedup over serial execution.

Similarly, the "MIMD+SIMD" bars in Figure 9 show that our parallelization scheme performs well on the physical machine, although not as well as on the simulator. Comparing the "MIMD+SIMD" and "Serial" bars, our parallelization scheme achieves execution time reductions between 66.7% and 95.0%. Averaged across all benchmarks, the execution time reduction is 81.4%, which translates to a 8.67x speedup. Moreover, our parallelization scheme outperforms CPU-only parallelization by 2.4x on average, and it outperforms GPU-only parallelization by 2.3x on average. Except for 358.botsalgn where CPU-only parallelization is slightly better, and for 359.botsalgn where GPU-only parallelization is slightly better, nested MIMD-SIMD parallelization is the best scheme. Averaged across all benchmarks, nested MIMD-SIMD parallelization beats the next-best scheme by 1.23x. But on the simulator, the speedup over serial is 16.0x (instead of 8.67x), and the speedup over the next-best scheme is 1.47x (instead of 1.23x).

One reason the simulator results are better is because the GPU in our physical machine supports less parallelism than the GPU in our simulator (5,376 threads vs. 24,576 threads). At the same time, the physical machine's CPU runs faster (between 3.4 GHz and 4.0 GHz) compared to the simulator's CPU (2.6 GHz). Both tend to reduce the relative gain for off-loading computations from the CPU onto the GPU. In addition, the physical machine's higher launch latency is a contributor, especially for benchmarks with a small W_{SIMD}. This is the reason for the poorer performance in 358.botsalgn on the physical machine compared to the simulator. (Even with the software optimization for 358.botsalgn, the "SIMD" bar is 6% worse than the "Serial" bar in Figure 9.) Last, the simulator employs more total last-level cache (LLC; 10 MB) compared to the physical machine (8 MB), and the simulator's cache memories, including the GPU's caches, all run at 2 GHz. On the physical

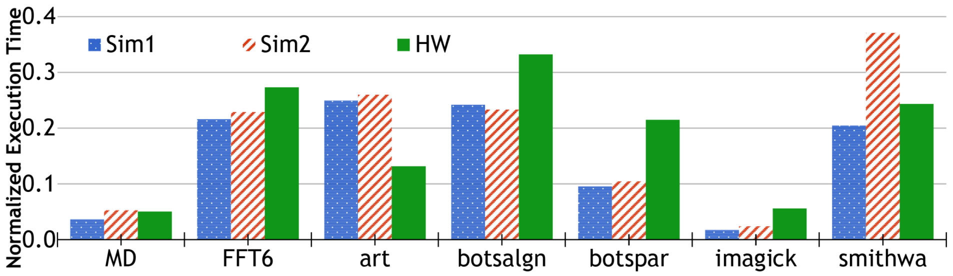


Fig. 10. Normalized execution time for the MIMD-SIMD parallelization technique running on two configurations of the simulator and the physical machine. The first simulator configuration follows Table 1, whereas the second is closer to the physical machine's configuration.

machine, the GPU shared L3 cache runs at the speed of the GPU, which ranges from 350 MHz to 1,150 MHz. This increases the time that the CPU must wait for GPU results to be transferred to the CPU after a GPU kernel has completed.

Nevertheless, Figure 9 demonstrates that our parallelization scheme can provide real performance gains on actual systems. Taken together, Figures 4, 5, and 9 show that our approach can be effective across different platforms with varying hardware capabilities.

7.5 Simulator Validation

In this section, we compare our physical machine results against our simulation results from Section 6 in detail. As discussed in Section 7.4, our simulator reports a higher speedup for MIMD-SIMD parallelization than the physical machine. Figure 10 shows this by plotting all of the "MIMD+SIMD" bars from Figure 4 (using the results from Figure 5 for FFT6 and 358.botsalgn) side by side with the "MIMD+SIMD" bars from Figure 9. In Figure 10, these experiments are labeled "Sim1" and "HW," respectively. Notice that each bar has been normalized to the corresponding "Serial" bar either from the simulator (in Figure 4) or the physical machine (in Figure 9). Thus, rather than compare absolute execution times, Figure 10 is comparing the relative gains achieved on the two platforms. In almost every benchmark, the simulator shows a larger gain (smaller normalized execution time) compared to the physical machine. Averaged across all benchmarks, the difference between the Sim1 and HW bars is 36%.

To see whether the simulation results could better agree with the hardware measurements, we changed the simulation parameters to match (when possible) the physical machine's configuration. In particular, we configured the simulator to have the same number of SMs as the physical GPU and reduced the number of warps per SM so that the total number of simultaneous threads matches the hardware (5,376). We also increased the CPU's clock rate to 3.4 GHz and decreased the speed of all caches to 1 GHz. Finally, we reduced the CPU's private L2s and the GPU's shared L2 to 1.5 MB each, yielding 7.5 MB of total capacity at the last level of caching (getting closer to the physical machine's 8-MB LLC). Unfortunately, there remain differences between the reconfigured simulator and the hardware that we could not address. Although the physical machine has an LLC that is shared across both the CPU and GPU, our gem5-gpu simulator does not model shared caches between the CPU and GPU. Furthermore, as mentioned in Section 7.1, we were unable to find documentation on the physical machine's coherence protocol. Thus, there are likely differences in how cache coherence is maintained in the simulator versus the hardware, especially between the CPU and GPU.

The bars labeled "Sim2" in Figure 10 report the normalized execution time for MIMD-SIMD parallelization on the reconfigured simulator. For MD, FFT6, 359.botsspar, and 367.imagick, the agreement between the Sim2 versus HW bars improved compared to Sim1 versus HW, with a significant improvement for MD. For 330.art and 358.botsalgn, the agreement got slightly worse, but for 372.smithwa, the agreement got significantly worse. Overall, the percentage difference between Sim2 versus HW improved slightly (32%) but is similar to the difference between Sim1 versus HW. To get better agreement, we believe that it is necessary to address the remaining differences between the simulator and physical machine discussed earlier.

8 CONCLUSION

Given the wide availability of heterogeneous microprocessors, we believe that an important research direction is to find new computations that can benefit from integrated GPUs. We argue that doing so will require new parallelization schemes that can make effective use of both CPU and GPU cores simultaneously. In this article, we propose *nested MIMD-SIMD parallelization,* which maps parallel outer loops/regions onto CPU cores and parallel inner loops with regular data-level parallelism onto GPU cores. We find that such nested parallel structures are quite common in OpenMP programs. Given coarse-grained OpenMP parallel regions, which parallel programmers are incentivized to express in OpenMP, nested parallelism occurs anytime one or more of the nested inner loops exhibit regular data-level parallelism. Even when the inner loops originally written by a programmer are serial, there may still be data parallel statements buried within the otherwise serial loops. We employ a loop fission transformation to separate these data parallel statements from the serial portions and expose the nested parallelism that we seek.

We conducted an in-depth evaluation of nested MIMD-SIMD parallelization using seven OpenMP programs on a simulator with an aggressively sized GPU and a physical machine with a more modest GPU. On the simulator, our results show that exploiting nested MIMD-SIMD parallelism provides a 16.1x speedup over serial execution. On the physical machine, the gain of nested MIMD-SIMD parallelization drops a bit, to a 8.81x speedup. This is mainly due to the less aggressive GPU in the physical platform compared to the simulator platform. Our parallelization scheme also beats CPU-only parallelization by 4.13x and 2.40x on the simulator and physical machine, respectively, and beats GPU-only parallelization by 2.74x and 2.26x, respectively. Compared against the next-best scheme (either CPU- or GPU-only parallelization) across all of our benchmarks, nested MIMD-SIMD parallelization provides speedups of 1.46x and 1.27x on the simulator and physical machine, respectively. This demonstrates that our parallelization scheme can be effective over different heterogeneous microprocessors with varying capabilities.

Although this article focused on nested MIMD-SIMD parallelization, we believe that additional parallelization schemes will be needed to effectively use heterogeneous microprocessors. One possibility is to parallelize computations exhibiting *software pipelining* [38]. For example, different software pipeline stages may consist of data parallel and irregular code regions that could be simultaneously mapped onto CPU and GPU cores. Another important direction for future work is to automate our parallelization techniques. As mentioned in Section 3.1, OpenMP supports offload directives, and some compilers can already automatically translate code marked with offload directives into GPU code. A practical approach is to extend the OpenMP offload directive so that it can work with multi-kernel launch. (Compiler passes could also be added to perform loop fission before GPU code generation occurs.) We hope to pursue these directions for future work.

REFERENCES

[1] Intel Corporation. [n.d.]. Intel Sandy Bridge Microarchitecture. Available at http://www.intel.com.
[2] N. Brookwood. 2010. *AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience*. White Paper. AMD.
[3] Apple Inc. [n.d.] iPhone. Available at: https://www.apple.com/eg/iphone/.
[4] Chris Gregg and Kim Hazelwood. 2011. 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.*

[5] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2015. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs. In *Proceedings of the International Symposium on Computer Architecture.*

[6] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. 2009. Enabling task parallelism in the CUDA scheduler. In *Proceedings of the Workshop on Programming Models for Emerging Architectures held in conjunction with the Symposium on Parallel Architectures and Compilation Techniques.*

[7] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. 2012. Towards efficient GPU sharing on multicore processors. *ACM SIGMETRICS Performance Evaluation Review* 40, 2 (Sept. 2012), 119–124.

[8] Florian Wende, Frank Cordes, and Thomas Steinke. 2012. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing.*

[9] OpenMP. 2014. The OpenMP API Specification for Parallel Programming. Available at http://www.openmp.org.

[10] Jason Power, Joel Hestness, Mar S. Orr, Mark D. Hill, and David A. Wood. 2014. gem5-gpu: A heterogeneous CPU-GPU simulator. *IEEE Computer Architecture Letters* 13, 1 (Jan. 2014), 34–36.

[11] Mayank Daga, Ashwin M. Aji, and Wu Chun Feng. 2011. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In *Proceedings of the Symposium on Application Accelerators in High-Performance Computing.*

[12] Kyle Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, and Jeffrey S. Vetter. 2012. The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *Proceedings of the ACM International Conference on Computing Frontiers.*

[13] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units.*

[14] J. Dongarra and P. Luszczek. 2005. *Introduction to the HPC Challenge Benchmark Suite.* Technical Report. University of Tennessee–Knoxville.

[15] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B. Baden, and Dean M. Tullsen. 2012. Redefining the role of the CPU in the era of CPU-GPU integration. *IEEE Micro* 32, 6 (Nov./Dec. 2012). 4–16.

[16] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the International Symposium on Workload Characterization.*

[17] SPEC. 2015. SPEC's Benchmarks. Retrieved November 5, 2019 from http://www.spec.org/benchmarks.html.

[18] Michael Zuzak and Donald Yeung. 2017. Exploiting multi-loop parallelism on heterogeneous microprocessors. In *Proceedings of the 10th International Workshop on Programmability and Architectures for Heterogeneous Multicores.*

[19] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshave Pingali. 2014. Adaptive heterogeneous scheduling for integrated GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques.*

[20] Vignesh T. Ravi and Gagan Agrawal. 2011. A dynamic scheduling framework for emerging heterogeneous systems. In *Proceedings of the 18th International Conference on High Performance Computing.*

[21] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. 2010. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the International Conference on Supercomputing.*

[22] Guray Ozen. 2017. *Compiler and Runtime Based Parallelization and Optimization for GPUs.* Ph.D. Dissertation. Universitat Politecnica de Catalunya (UPC).

[23] Yi Yang and Huiyang Zhou. 2014. CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

[24] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christopher W. Ueberhuber. 2005. Efficient utilization of SIMD extensions. *Proceedings of the IEEE* 93, 2 (Feb. 2005), 1–15.

[25] John A. Stratton, Christopher Rodrigues, I.-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen Mei Hwu. 2012. *The Parboil Technical Report.* IMPACT Technical Report (IMPACT-12–01).

[26] A. J. Dorta, C. Rodriguez, F. d. Sande, and A. Gonzalez-Escribano. 2005. The OpenMP Source Code Repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing.* 244–250. DOI: 10.1109/EMPDP.2005.41

[27] SPEC. 2001. SPEC OMP 2001. Retrieved November 5, 2019 from https://www.spec.org/omp2001/.

[28] Steven Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, CA.

[29] Michael Kruse and Hal Finkel. 2018. *Loop optimization framework.* arXiv:1811.00632.

[30] SPEC. 2012. SPEC OMP 2012. Retrieved November 5, 2019 from https://www.spec.org/omp2012/.

[31] Michael Kruse and Tobias Grosser. 2018. DeLICM: Scalar dependence removal at zero memory cost. In *Proceedings of the International Symposium on Code Generation and Optimization.*

[32] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, et al. 2018. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (May 2011), 1–7.

[33] Intel Corporation. [n.d.]. Intel Core i7-6770HQ Processor. Retrieved November 5, 2019 from https://ark.intel.com/products/93341/Intel-Core-i7-6770HQ-Processor-6M-Cache-up-to-3_50-GHz.

[34] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software.*

[35] Daniel Lustig and Margaret Martonosi. 2013. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *Proceedings of the International Symposium on High Performance Computer Architecture.*

[36] Gem5. 2009. Gem5 M5threads. Retrieved November 5, 2019 from https://github.com/gem5/m5threads.

[37] Michael Mrozek and Zbigniew Zdanowicz. 2016. GPU daemon: Road to zero cost submission. In *Proceedings of the 4th International Workshop on OpenCL.*

[38] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization.*

Received March 2019; revised October 2019; accepted October 2019