# ABSTRACT

Title of Thesis: EXPLOTING NESTED PARALLELISM ON HETEROGENEOUS PROCESSORS

Michael Jeffrey Zuzak, Master of Science, 2016

Thesis Directed By: Associate Professor and Director of Computer Engineering Education, Doctor Donald Yeung, Department of Electrical and Computer Engineering

Heterogeneous computing systems have become common in modern processor architectures. These systems, such as those released by AMD, Intel, and Nvidia, include both CPU and GPU cores on a single die available with reduced communication overhead compared to their discrete predecessors. Currently, discrete CPU/GPU systems are limited, requiring larger, regular, highly-parallel workloads to overcome the communication costs of the system. Without the traditional communication delay assumed between GPUs and CPUs, we believe non-traditional workloads could be targeted for GPU execution. Specifically, this thesis focuses on the execution model of nested parallel workloads on heterogeneous systems. We have designed a simulation flow which utilizes widely used CPU and GPU simulators to model heterogeneous computing architectures. We then applied this simulator to non-traditional GPU workloads using different execution models. We also have proposed a new execution model for nested parallelism allowing users to exploit these heterogeneous systems to reduce execution time.

# EXPLOITING NESTED PARALLELISM IN HETEROGENEOUS COMPUTING SYSTEMS

by

Michael Jeffrey Zuzak

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2016

Advisory Committee:
Professor Donald Yeung, Chair
Professor Bruce Jacob
Professor Manoj Franklin

© Copyright by
Michael Jeffrey Zuzak
2016

# Table of Contents


Table of Contents ..... ii
List of Tables ..... iii
List of Figures ..... iv
Introduction ..... 1
  What is Nested Parallelism? ..... 2
  Taking a Deeper Look at Multigrain Parallelism ..... 5
Prevalence of Multigrain Parallelism ..... 7
  Is Multigrain Parallelism Prevalent? ..... 8
  Results of NAS Code Review ..... 9
  Results of LULESH Code Review ..... 10
  Analysis of Results ..... 12
Multigrain Parallel Execution Models ..... 12
  Proposed Multigrain Execution Models ..... 14
Running Inner Nested Loops on a GPGPU ..... 17
  Finding Loops ..... 17
  CPU Simulator ..... 17
  GPU Simulator ..... 18
  GPU Inner Loop Simulations ..... 19
  Results ..... 19
    ART ..... 19
    MD ..... 22
    FFT6 ..... 23
  Conclusions ..... 26
Concurrency Analysis ..... 26
  Concurrency Analysis ..... 27
  Assumptions Made ..... 27
  Functionality ..... 28
  Determining Which Core Runs Code ..... 29
  Concurrency Experiment ..... 29
    ART ..... 30
    MD ..... 31
    FFT6 ..... 32
Conclusions ..... 33
Related Works ..... 35
Future Work ..... 37
Appendices ..... 40
Bibliography ..... 44

# List of Tables

Table 1 ART Benchmark Execution Time in Cycles ................................................ 21
Table 2 MD Benchmark Execution Time in Cycles .................................................. 23
Table 3 FFT6 Benchmark Execution Time in Cycles ............................................... 24
Table 4 ART Benchmark Concurrent Execution Results ........................................... 31
Table 5 MD Benchmark Concurrent Execution Results ............................................. 32
Table 6 FFT6 Benchmark Concurrent Execution Results .......................................... 33

# List of Figures

Figure 1 Nested Parallel Code Region ..... 2
Figure 2 More Complex Nested Parallel Region ..... 3
Figure 3 Examples of Nested Parallel Code Constructs Selected From MD, FFT6, and ART ..... 3
Figure 4 Nested Parallel Code Constructs from CG, Equake, and ART ..... 5
Figure 5 NAS Code Review Results Aggregate ..... 9
Figure 6 NAS Code Review Results by Benchmark ..... 10
Figure 7 LULESH Code Review Results ..... 11
Figure 8 Multigrain Region from ART ..... 14
Figure 9 Execution Models of Multigrain Parallel Code Constructs. (Mono-Parallelism on Left, Dual-Parallelism on Right) ..... 15
Figure 10 More Complex Dual-Parallelism Execution Model Example ..... 17
Figure 11 Nested Region of Interest in ART Benchmark ..... 20
Figure 12 ART Benchmark Normalized Execution Time ..... 21
Figure 13 MD Nested Region of Interest ..... 22
Figure 14 MD Benchmark Normalized Execution Time ..... 23
Figure 15 FFT6 Benchmark Normalized Execution Time ..... 24
Figure 16 FFT6 Loop 1 ..... 24
Figure 17 FFT6 Loop 2 ..... 25
Figure 18 FFT6 Loop 3 ..... 25
Figure 19 ART Benchmark Normalized Concurrent Execution Results ..... 31
Figure 20 MD Benchmark Normalized Concurrent Execution Results ..... 32
Figure 21 FFT6 Benchmark Normalized Concurrent Execution Results ..... 33
Figure 22 : Execution Time Speed-up Multiplier for Concurrent Simulator Execution Models ..... 34
Figure 23 Simple Scalar Out of Order Configuration Parameters ..... 40
Figure 24 GPGPU-Sim Configuration File ..... 43

# Introduction

In the multicore world we currently live in, researchers constantly develop new ways to parallelize tasks being executed. Novel ways of finding and exploiting different types of parallelism inherent in the problems we solve is one of the cheapest and easiest ways to achieve runtime speedup. This can be seen by the amount of research into programming languages and APIs that help programmers identify and exploit parallelism [6, 23]. Nested parallel constructs are the focus of this research as they are one of the easiest code constructs to parallelize due to its identifiable parallel structure.

Intel and AMD have both begun releasing heterogeneous chips, and are proposing further heterogeneous designs [5, 19]. By introducing both SIMT and CPU cores on the same die, communication times are reduced over the discrete CPU/GPU designs relying on a communication bus to transfer data [5, 19]. Traditionally, the communication between the CPU and GPU has limited the types of workloads that could be forwarded to those with enough work and regularity to offset the base data transfer cost [21]. We believe current execution models are outdated on heterogeneous architectures. In this paper we explore novel execution models to exploit nested parallelism in a heterogeneous system where the communication time between a CPU and a GPU is reduced.

*What is Nested Parallelism?*

The basis of this investigation is nested parallel coding constructs. This term is often loosely defined, so for the sake of clarity we will give our definition of a nested parallel coding construct. The most basic example of a code region exhibiting nested parallelism is shown in the figure below.

```
for(i=0;i<n;i++)
{
    for(j=0;j<m;j++)
    {
        //Computational Code
    }
}
```

*Figure 1 Nested Parallel Code Region*

Often times, these loops have significant GPU-friendly parallelism and can be sped up by offloading the inner loop to SIMT cores from a single CPU thread. This method performs well with large, regular loops in which the inner, GPU-targeted region contains the majority of the work, but as always, there are exceptions to the rule.

In loops, such as the one shown below, the above described execution model will under-perform because these smaller loops may not fully utilize SIMT cores. The computations between GPU regions may also be significant in terms of runtime, which limits the GPU's performance due to Amdahl's Law.

```
#pragma omp parallel for private(cor,j,k,xc,vertices)
for (i = 0; i < ARCHelems; i++) {
  for (j = 0; j < 4; j++)
     cor[j] = ARCHvertex[i][j];
  if (cor[0] == Src.sourcenode || cor[1] == Src.sourcenode ||
     cor[2] == Src.sourcenode || cor[3] == Src.sourcenode) {
     for (j = 0; j < 4; j++)
        for (k = 0; k < 3; k++)
           vertices[j][k] = ARCHcoord[cor[j]][k];
     centroid(vertices, xc);
     source_elms[i] = 2;
     if (point2fault(xc) >= 0)
        source_elms[i] = 3;
  }
}
```

*Figure 2 More Complex Nested Parallel Region*

## Examples of Nested Parallelism

Although the nested parallelism code sample we showed is quite simple, it does not have to be. Many widely used benchmarks display divergent control flow, loop break conditions, and other unpredictable code constructs within the nested parallelism. Some of this complexity is shown in the loops displayed below.

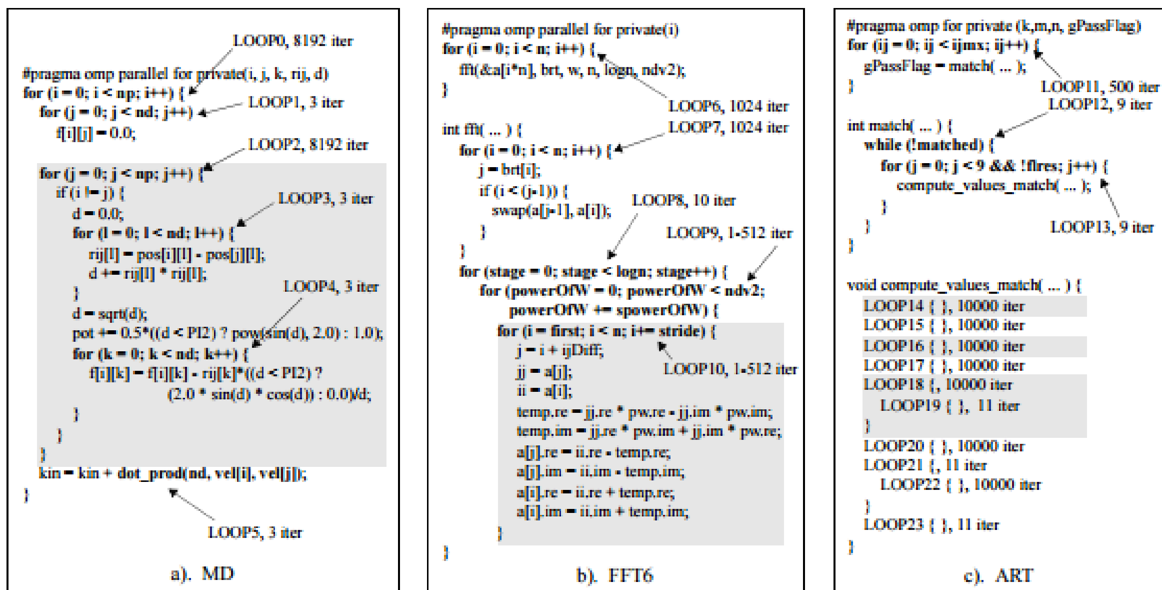


*Figure 3 Examples of Nested Parallel Code Constructs Selected From MD, FFT6, and ART*

The loops in the figure all display nested parallel constructs, but they also represent a more complex subset of nested parallelism. Specifically, these loops contain more complex outer loops, with simpler inner loops. These outer loops are either low iteration or unpredictable. For example, in ART the while(!matched) statement in the outer loop creates an indeterminate length loop making it unpredictable. The low iteration count and irregular control flow makes these outer loops a more coarse-grain of parallelism. Traditionally and as implemented in the benchmark suite, this coarse-grain parallelism is more attractive for parallel CPU execution. Upon inspection of the inner loops for ART, many, but not all of them are computational, stepping or striding through an array to perform the same computation many times with a large iteration count. These regular, often array-based computations make these inner-loops attractive to a GPU, however three of these inner-loops also exhibit control flow divergence and contain little work, therefore in a discrete system, these loops would under-perform on a SIMT core. With the communication overhead in current GPU systems often in the tens to hundreds of microseconds [8], it does not provide any speed-up to offload loops of this nature. With the surging of heterogeneous architectures, GPU cores and CPU cores have been brought closer together and given us the ability to more rapidly forward work between cores. This makes smaller, less-traditional workloads such as those shown above more attractive for offloading.

It is the dichotomy present in these inner and outer loops, with the outer loops displaying a coarser, more CPU targeted parallelism, and the inner loops displaying a finer grain, more GPU target parallelism that we are targeting in this study. For the

remainder of this paper we will call these types of nested parallel loops “multigrain parallel code constructs” to highlight the difference between the coarse-grain outer loop and the fine-grain inner loop. We believe that with the advent of heterogeneous architectures the execution model of these loops warrants another look.

_Taking a Deeper Look at Multigrain Parallelism_

We have compiled OpenMP regions from three benchmarks we have reviewed to discuss the characteristics of multigrain parallelism in more detail. See the figure below for the code [3, 25].

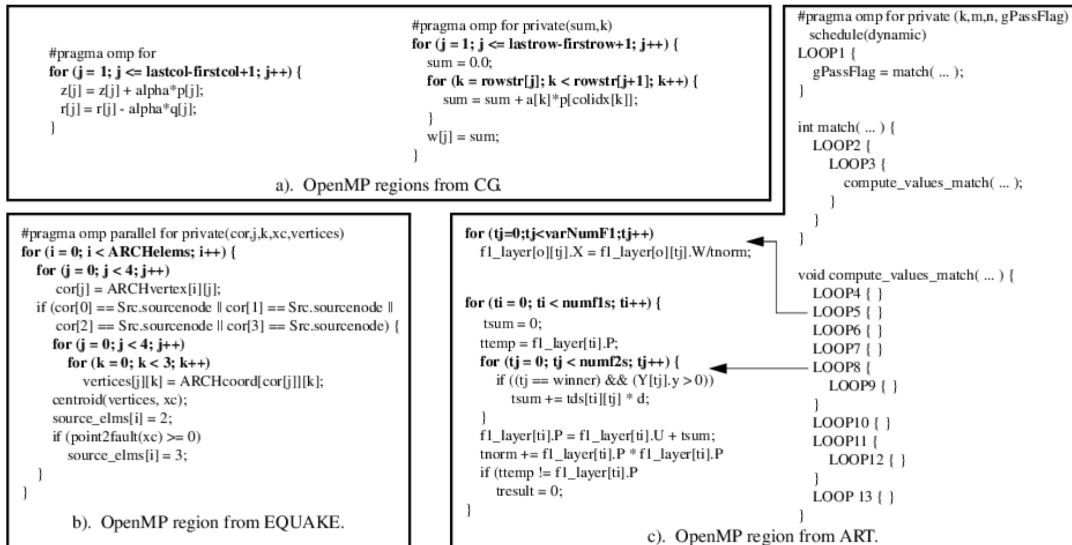


*Figure 4 Nested Parallel Code Constructs from CG, Equake, and ART*

Before taking a deeper look, one can note, that even though all these constructs are nested, they are very different. The prime similarity between these code samples is that they all include some sort of array type calculation in the inner loop, although that statement is nearly ubiquitous with programming and code in general.

Starting with CG, notice that the code has two OpenMP regions, one nested, and one non-nested region. The non-nested parallel loop is a simple calculation and

stored into an array. It appears regular and the size of the loop is predictable as lastcol and firstcol do not change mid-loop. The nested OpenMP region is several sparse matrix reductions which are then stored in an array. This first loop, while not nested is very regular and predictable in behavior. There is not a significant amount of work in this loop so traditionally it would not be targeted for SIMT cores. The second loop is nested in nature and has an irregular structure due to the varying size of the inner loop.

Equake displays a very dissimilar type of nested parallelism compared to CG. In Equake, the inner loops do not contain enough work to overcome the overhead of communication regardless of the heterogeneity of the architecture. The outer loop contains more work in it, but at the expense of control flow divergence introduced by the “if” statements. Visual inspection does not provide an obvious mapping of this code to either a GPU or a CPU due to the irregularity, problem size, and divergence. The loop appears to have control flow divergence due to the two if statements which if so, would make the loops poor targets for GPU execution.

The final example of multigrain parallelism we have selected is that of ART. In ART, there is a divergent and low iteration outer loop that clearly does not target the GPU well. We would propose executing this divergent outer parallelism using multiple CPU cores if we were to parallelize. The inner loops; however, might target well to the GPU, but not all of them do. Each of these loops has varying degrees of divergence, work, and irregularity that influence how well they can be executed by the GPU.

There is a fair amount of diversity between the three loops chosen for deeper inspection. An optimal execution model would need to select only multigrain parallel loops from non-multigrain parallel loops and also be able to efficiently execute these loops.

## Prevalence of Multigrain Parallelism

Nested parallelism has been extensively studied as an easy source of speedup in the multicore era [7, 12, 18]. The primary draw here is the easily recognizable and parallelizable nature of the regions. The code, already in the proper ISA, can then be forwarded, generally with very few hazards, to other cores to share work and get execution-time speedups. When transferring this idea to heterogeneous computing and to the specific type of nested parallelism we are targeting, there are four primary difficulties introduced:

1. The GPU and CPU have an introduced communication delay.
2. CPU’s and GPU’s use different ISAs.
3. Divergent and irregular loops do not target well to GPUs. Often these loop characteristics must be identified at runtime.
4. Loops must have a parallel outer loop with an independent inner loop.

These four factors make forwarding these loops to the GPU less attractive. Recently however, heterogeneous architectures are reducing the concerns brought up by point one. Many developments have also been made recently in compiling CPU targeted parallel code, for example OpenMP to CUDA [23]. These breakthroughs reduce the barrier that point two once imposed. This leaves points three and four that must be addressed before applying heterogeneous processing to these loops. In hopes

of shedding some insight on points three and four, a code study was completed to help classify code that is multigrain parallel.

### *Is Multigrain Parallelism Prevalent?*

To answer this question, we first identified the types of loops that could be targeted to a GPU without taking account of the actual performance of these loops. More specifically, we hoped to find nested loops that followed the spirit of our model of multigrain parallelism in which the:

1. Outer loop must be parallelizable
2. Inner loop must be parallelizable
3. Inner loop must be of fixed/predictable size

The idea was to determine the prevalence of multigrain parallel code constructs with the intention of determining loop performance and optimum execution heuristics later. For multigrain parallelism to be worth being exploited using heterogeneous architectures, it must be both common enough in code and contain significant work to allow for sizeable execution time improvements. We studied the NAS benchmark suite for this code study. The NAS benchmark suite is "a small set of programs designed to help evaluate the performance of parallel supercomputers." [3]. The suite contains five kernel benchmarks and three pseudo application benchmarks. The hope of the benchmark is to represent common – variable size workloads that parallel supercomputers might see. The kernels tended to be scientific in nature.

To profile the NAS suite, we stepped through, line by line, the functions that took over 1% of runtime according to gprof. We then categorized code in these functions into "contains multigrain parallelism" or "does not contain multigrain

parallelism" categories, and tallied up the percent of execution containing multigrain parallelism with functional granularity.

*Results of NAS Code Review*

We determined that of the almost 99% of NAS execution time surveyed, 59.3% of the functions contained nested parallel constructs.

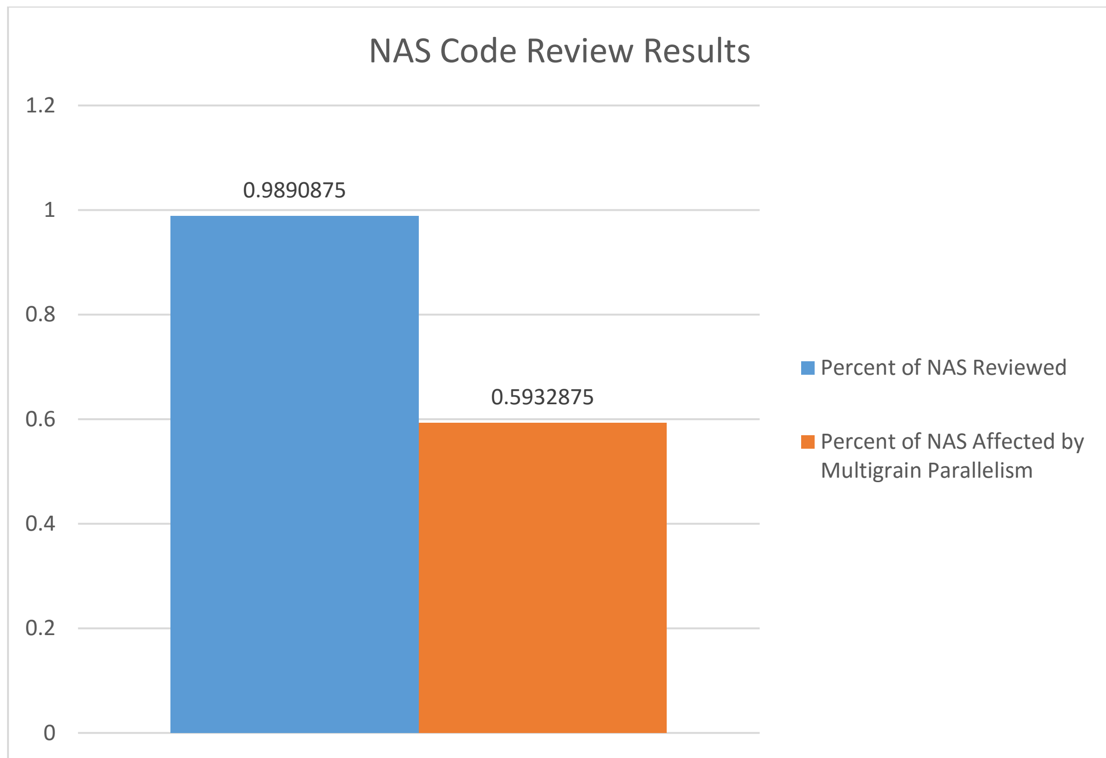


*Figure 5 NAS Code Review Results Aggregate*

This implies that by exploiting multigrain parallelism, 60% of the runtime of suite runtime could be affected in the best case. Realistically, there is work other than the multigrain constructs in these regions, but with 60% of the functions containing multigrain parallelism and many containing nothing else, there appears to be significant opportunity for execution time reduction.

Continuing to break down these results, you can see that multigrain parallelism tends to be an all or nothing phenomenon. Most of the benchmarks exhibited a great deal of multigrain parallelism (with respect to runtime), or none at all. There was very little middle ground. From further inspection, it was the benchmarks that included matrix computation, such as dot products, which contained multigrain parallelism. Below, the benchmark specific results for the study are posted.

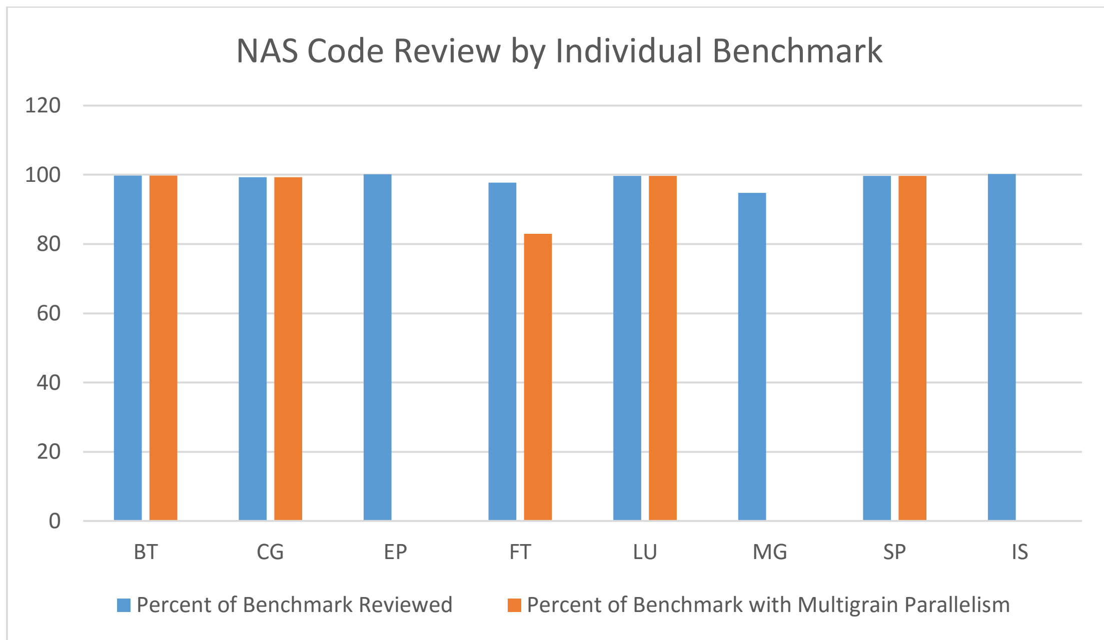


*Figure 6 NAS Code Review Results by Benchmark*

With nearly 60% of NAS functions containing multigrain parallelism, the exploitation of this parallelism could open the door to a significant speedup.

*Results of LULESH Code Review*

After gathering the results of the NAS code review, we gathered results from a full software application as opposed to the computational kernels that NAS offers. NAS consists primarily of micro-kernel benchmarks, which while representing real parallel workloads, do not tend to represent the complexity in most useful scientific

applications. These NAS benchmarks aim to simply gather common compute loops that dominate scientific codes, instead of solving an end to end scientific problem that may include, but not be dominated by, these compute loops.

To get a better idea of a real world application, we performed the same code review as performed on NAS on the LULESH benchmark, which is one of the five challenge problems in the DARPA UHPC program. Specifically, LULESH is the shock hydrodynamics challenge problem. This code has been widely studied, and represents a more irregular and holistic type of scientific code than the NAS suite. The results of the code review are shown below.

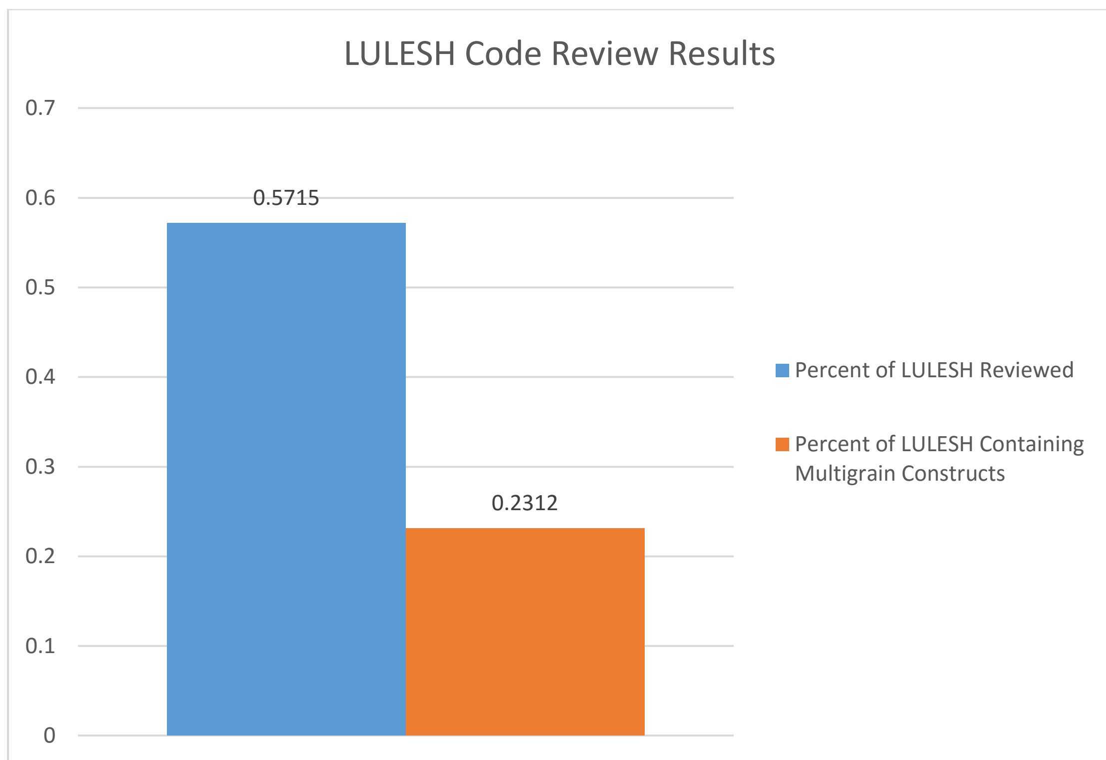


*Figure 7 LULESH Code Review Results*

The overall code reviewed is below 100% due to the large number of less than 1% execution time functions. These functions were ignored due the minor effect on overall runtime. If the remainder of the code exhibited multigrain parallelism at the

same rate as the reviewed portion, roughly 40% of the code would be multigrain. Upon reviewing the code, this can largely be attributed to the significant amount of data setup, aggregation, and preparation loops contained in LULESH. These setup portions were ignored in the micro-kernels of NAS which omitted everything except the computational loops.

*Analysis of Results*

From the code study, it appears that nested parallelism is present in fairly significant amounts in scientific parallel codes. Although the 59.3% of execution time governed by functions containing multigrain parallelism in NAS is most likely higher than that present in common commercial and academic applications, we argue that as a fairly generous estimate, it is more likely that the rate lies somewhere around the 30-40% line exhibited in LULESH. If 30-40% of code execution time is affected by multigrain parallelism, exploiting these multigrain loops could lead to a sizeable speedup in overall code runtime.

## Multigrain Parallel Execution Models

To begin, we broke down all possible ways to handle a nested parallel construct.

1. Execute entirely on the CPU serially
2. Execute entirely on the GPU
3. Execute outer portion serially on CPU, and forward inner portion to GPU
4. Execute outer portion in parallel on CPU, forward inner portion to GPU
5. Execute outer portion in parallel on the CPU. Each thread serially executes inner portion on the CPU

The focus of this research on heterogeneous architectures makes options 2, 3, and 4 the only models that could exhibit speedup from the benefits of heterogeneity. Execution model 2 opts for entirely GPU execution of both inner and outer loop. This generally would not be favorable due to the coarse nature of the outer loop in multigrain parallelism. Due to the outer loops being poorly targeted for GPU execution, we will ignore this model. Execution model 1 is CPU-only and not parallel. In general, it provides the most basic model from which baseline execution times can be measured. Execution model 5 creates a parallel CPU execution, ignoring the GPU entirely. This model is the de facto standard execution model we have found in benchmarks such as SPEC 2001 OpenMP. Because this model receives no benefit from having a low communication overhead GPU, we will use execution model 5 execution time as the second baseline to compare execution models 3 and 4. Therefore, to summarize, the execution models of interest are models 3 and 4 which will be compared to execution models 1 and 5 as baselines.

We have dubbed option 3 “Dual Parallelism” due to the coarser grain outer loop which runs on the CPU in parallel, and the finer grain inner loop which runs on the GPU in parallel. Dual parallelism focuses on exploiting the strength of CPUs in the execution of large coarse chunks of code that can be irregular, divergent, and unpredictable, while using the GPU which executes regular computational code very effectively.

We have dubbed option 2 “Mono Parallelism.” Option 2 would require a very regular, high iteration outer loop that also contains a very regular inner loop that runs similarly from iteration to iteration of the outer loop. These types of loops would

offer the GPU more work than only forwarding the inner loop as option 3 does, but require a more constrained loop to run effectively. Both of these options benefit significantly from the reduced communication delay heterogeneous architectures provide.

*Proposed Multigrain Execution Models*

Now we will take an in depth look at execution models two and three with respect to ART. The region of interest in ART is pictured again below.

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

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

*Figure 8 Multigrain Region from ART*

There are two ways to handle executing the parallel loops of the type found in ART. The first execution model we dubbed mono parallelism where a single CPU forks the entire inner loop. This is the standard execution model of parallel systems involving a GPU. To execute the pictured loop in a mono parallel fashion, a single CPU core would fork all inner loop work to a SIMT core until all processing is

complete. At this point the GPU would return and the CPU would resume execution. This is shown in the figure below on the left side. Point one represents the start of the inner loop where data is forked to a SIMT core. The inner loop is then executed on the GPU until completion where the CPU resumes.

The second execution model of interest is dual parallelism. For ART, this would consist of a parallel CPU forking the inner loops as they were reached. It is displayed on the right side in the figure below. On reaching the outer loop, point 2 in the figure, multiple CPU threads are spawned. As these threads reach the inner loop work, point 3 in the figure, threads are forked to a SIMT core. When this work is completed the CPU resumes execution. This execution model has SIMT cores service multiple CPUs simultaneously.

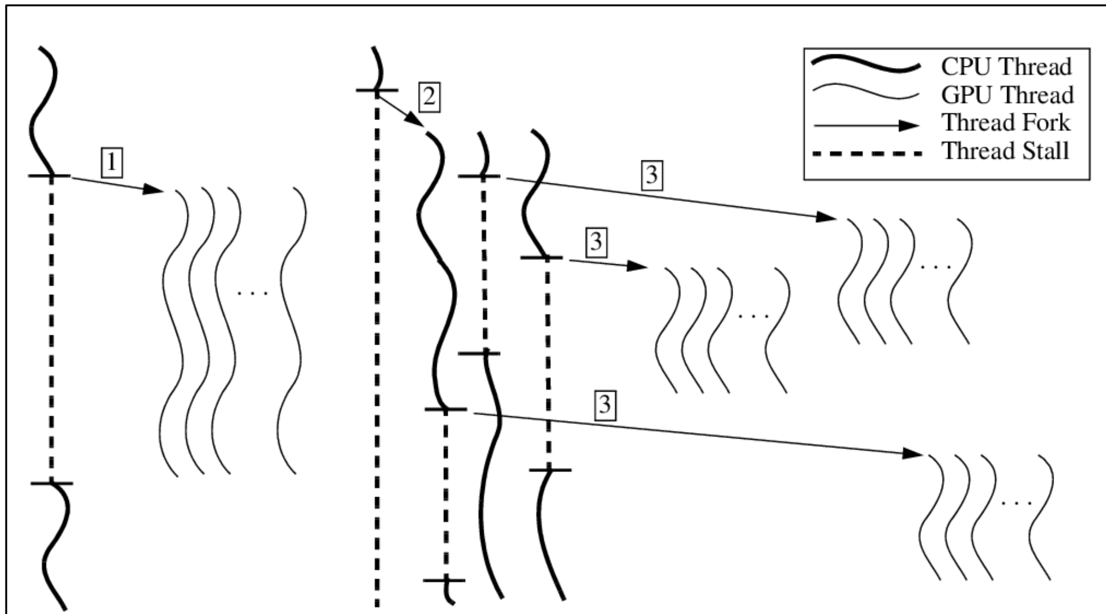


*Figure 9 Execution Models of Multigrain Parallel Code Constructs. (Mono-Parallelism on Left, Dual-Parallelism on Right)*

The dual model allows both CPU and GPU parallelism to be utilized, but comes with the risk of overloading the GPU. The multigrain scenario carries some risk in overloading the GPU, which would cause CPU threads to hang while earlier

workloads are pushed through the GPU. In this scenario, the overall execution time would be bottlenecked by the GPU, and would be much greater than execution time under single-grain parallel execution model. With smaller, less divergent workloads, the multigrain loop has a greater potential for speedup, where many different parallel regions can be serviced by the GPU concurrently allowing a larger reduction in execution time.

This multigrain execution model becomes more complicated by more complex and divergent regions such as that in ART. In the ART example, 8 parallel inner loops have to be split between the GPU and CPU. It is likely that some of these loops will run poorly on SIMT cores. In fact, in our example ART, overall runtime would increase using the above execution model and forking all inner loops to the GPU. This suggests moving workloads back and forth between SIMT and CPU cores based on the core best suited to handle the workload. The execution model suggested is shown in the figure below. For ART, when the outer loop of the multigrain region is reached, the loop would be forked onto multiple CPUs, displayed by point 2 in the figure. These CPUs would execute inner loop workloads that the scheduler left on the CPU. Workloads that run more efficiently would be forked to SIMT cores, as shown by point 3 in the figure. Point 4 demonstrates a back and forth effect that could occur if part of a workload runs better on the CPU and part runs better on a GPU. In this case, the thread of execution would bounce between CPU and SIMT cores. This effect would significantly increase the execution time of a non-heterogeneous system, but the closeness of the cores on a heterogeneous system makes this dual parallel model more likely to yield improvement.

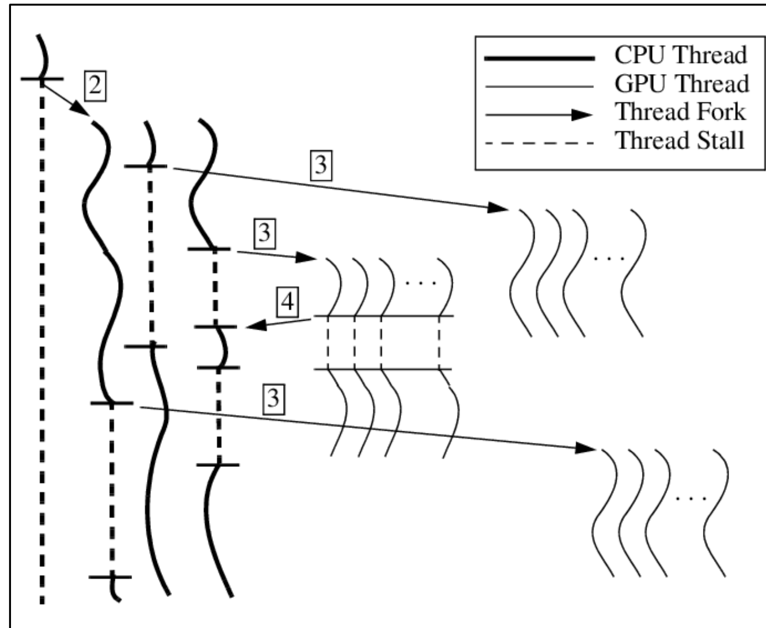

*Figure 10 More Complex Dual-Parallelism Execution Model Example*

## Running Inner Nested Loops on a GPGPU

*Finding Loops*

To investigate the possible benefit of exploiting multigrain parallelism, we reviewed the SPEC2000 suite and the OpenMP Source Code Repository to find multigrain loops [25, 26]. We then selected three of the benchmarks from these suites (ART, FFT, and MD) and profiled them running on both the CPU and GPU and the effect of the different processors on benchmark performance.

*CPU Simulator*

To simulate CPU execution, we selected the Simple Scalar simulator. We use the out-of-order model in Simple Scalar. We have configured this simulator to mimic a single core on a multicore CPU chip.

The simulator we used included a 2-level adaptive branch predictor with a 64 entry RAS stack. We use an issue width of 4, with 4 integer ALUs, 2 integer multiplier/dividers, 2 floating point ALUs, 2 floating point multipliers, 2 floating point dividers and a register update unit of size 128. There was a 2 level cache hierarchy implemented with both the data and instruction level 2 cache having a latency of 20 cycles, and the data and instruction level 1 cache with a 1 cycle latency. There are 4 memory ports operating with a 18 cycle latency to get the first data chunk, followed by a 2 cycle latency for each additional chunk. For a more detailed description, the full CPU simulator configuration file is included in the appendix of this thesis.

*GPU Simulator*

To simulate the GPU execution of code, we used the simulator GPGPU-sim. GPGPU-sim provides a detailed simulation model of NVIDIA GPUs running CUDA or OpenCL workloads [5]. The simulator allowed configuration of the GPU being simulated. In order to mimic the GPU cores common to heterogeneous architectures, we started with a stock Nvidia GTX480 GPU card and reconfigured the parameters.

We configured the simulator to these parameters based on research of current heterogeneous systems. To select the number of SIMT cores, we investigated several of the most prevalent heterogeneous systems available. Intel's i7-3770 processor includes 4 Ivy Bridge CPU cores and 16 execution units on the Intel HD 4000 unit, which are similar to SIMT cores. We also looked into AMDs Steamroller architecture which includes 2 to 4 CPU cores and 3 to 8 compute units. Intel and AMD are the two most prominent examples of heterogeneous architectures; therefore, we chose to

model their ratio of a 4 to 1 GPU to CPU core ratio to a 2 to 1 GPU to CPU core ratio. For this research, we took the more conservative end of heterogeneous processing and allotted 8 GPU cores, which in our system implies a 2 to 1 GPU to CPU core ratio. We then scaled down all other assets on the GTX-480 chip to reflect this change, leaving us with essentially half a GTX-480 GPGPU being simulated. For a more detailed description of the simulator configuration, see the appendix.

*GPU Inner Loop Simulations*

We began our investigation by determining whether the inner loops of multigrain parallel constructs would run well on a GPU. To do this, we converted the inner loops of the selected nested constructs into CUDA to be simulated on GPGPU-sim. Due to the often small problem size and divergent behavior, it was unclear whether there would be any speedup obtained executing the code on GPU over a CPU. To compare these two execution models, we ran regions of interest on both Simple Scalar and GPGPUsim, comparing the resulting execution times. We assumed the processor in Simple Scalar had a 3.3 GHz clock, which mimics a mid-range Sandy Bridge CPU, with the processor in GPGPU-sim having a 1.4 GHz clock rate (which is the clock rate of the GTX480 processor).

*Results*

ART

ART is a benchmark taken from SPEC OpenMP 2000 benchmark suite. A majority of the computation takes place in an OpenMP region including the function match(). See the figure below for the format of the benchmark.

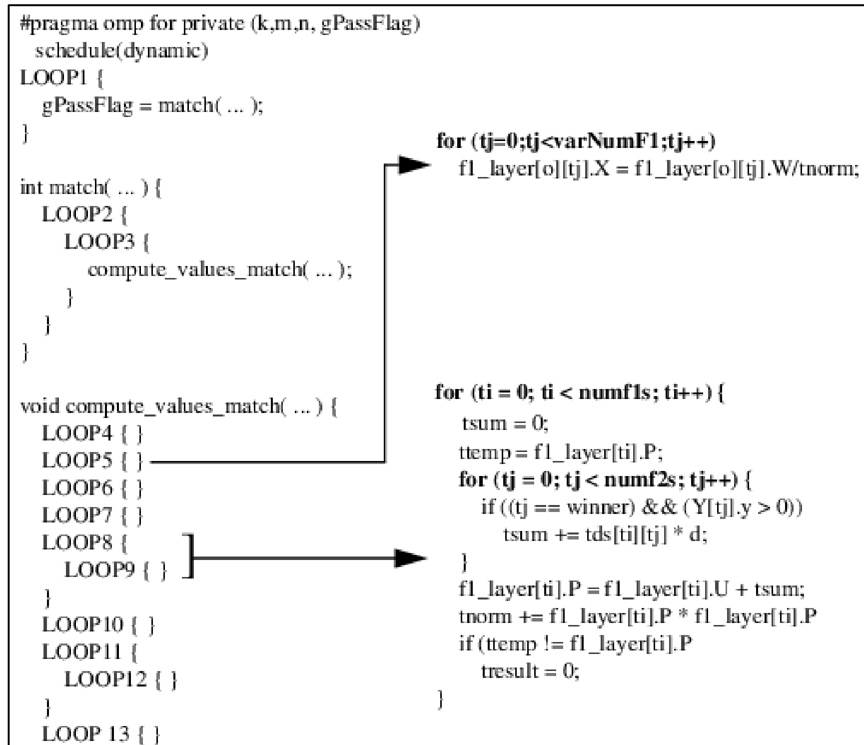


*Figure 11 Nested Region of Interest in ART Benchmark*

ART is a quadruple nested loop structure, with the outer loop exhibiting a low iteration count and very irregular. The compute_values_match(…) function which is called in the innermost loop, contains 10 loops, 2 of which are nested within 8 outer loops. These 8 outer loops in the compute_values_match(function) are much simpler and more computational in nature than the outer LOOP1 OpenMP loop.

The inner loops were converted to CUDA, run on the GPU and their resulting runtime (with respect to the same loop running on the CPU) are shown below.

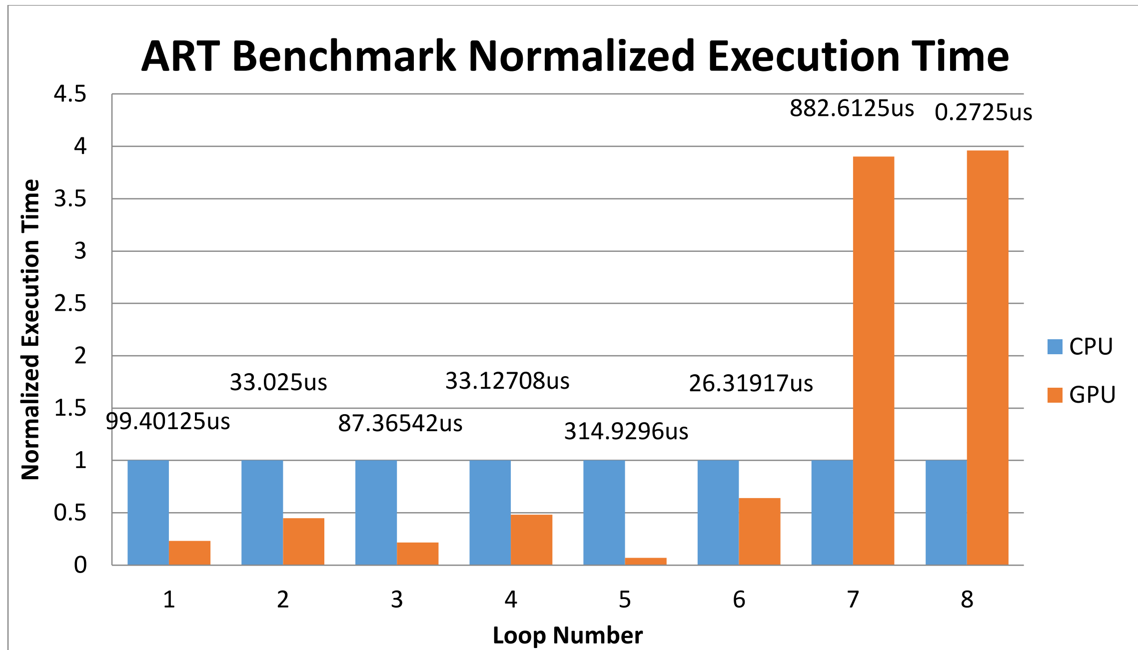


*Figure 12 ART Benchmark Normalized Execution Time*

| Art Loop Number | GPU Cycles | CPU Cycles |
|---:|---:|---:|
| 1 | 22916 | 238563 |
| 2 | 14839 | 79260 |
| 3 | 18981 | 209677 |
| 4 | 16005 | 79505 |
| 5 | 22097 | 755831 |
| 6 | 16891 | 63166 |
| 7 | 3444995 | 2118270 |
| 8 | 1079 | 654 |

*Table 1 ART Benchmark Execution Time in Cycles*

From the results, five of the loops ran better on the GPU (loops 1, 2, 3, 4, 5 and 6). The reminder of the loops ran better on the CPU. Deeper investigation of the two worst loops, 7 and 8, makes the cause of the slowdown clear. Loop 7 and 8 have a very low iteration count on the loop, either two or three in the SPEC provided input sets. Loop 8 also has divergent behavior due to the statement "if (Y[o][ti].y > Y[o][winner[o][0]].y)," which greatly inhibits possible GPU performance.

MD

MD is from the OpenMP Source Code Repository. It consists of one nested region of interest, shown below.

```
void compute(int np, int nd, vnd_t *pos, vnd_t *vel,
        double mass, vnd_t *f, double *pot_p, double *kin_p) {
  int i, j, k;
  vnd_t rij;
  double  d;
  double pot, kin;

  pot = 0.0;
  kin = 0.0;
  /* The computation of forces and energies is fully parallel. */
#pragma omp parallel for default(shared) private(i, j, k, rij, d) reduction(+ : pot, kin)
  for (i = 0; i < np; i++) {
    /* compute potential energy and forces */
    for (j = 0; j < nd; j++)
      f[i][j] = 0.0;
    for (j = 0; j < np; j++) {
      if (i != j) {
        d = dist(nd, pos[i], pos[j], rij);
        /* attribute half of the potential energy to particle 'j' */
        pot = pot + 0.5 * v(d);
        for (k = 0; k < nd; k++) {
          f[i][k] = f[i][k] - rij[k]* dv(d) /d;
        }
      }
    }
    /* compute kinetic energy */
    kin = kin + dot_prod(nd, vel[i], vel[j]);
  }
  kin = kin * 0.5 * mass;
  *pot_p = pot;
  *kin_p = kin;
}
```

*Figure 13 MD Nested Region of Interest*

MD has two inner loops. The first is an initialization loop setting every element of f to 0. The second loop, is more interesting. This loop contains some divergence due to the if statement along with a reduction of pot. The functions dist, v, and dv do not contain any obvious hazards. The outer loop forks iterations of the inner loop and then computes a reduction that is the sum of the dot product in each of the dimensions of interest. Because j is always np on exit from the inner loop and vel is independent of the inner loop, the work on the outer loop can be run fully in parallel with the inner loop allowing work to be overlapped.

This region was converted to CUDA and run on the GPU along with Simple Scalar for architectural comparisons. The resulting runtimes are shown below.

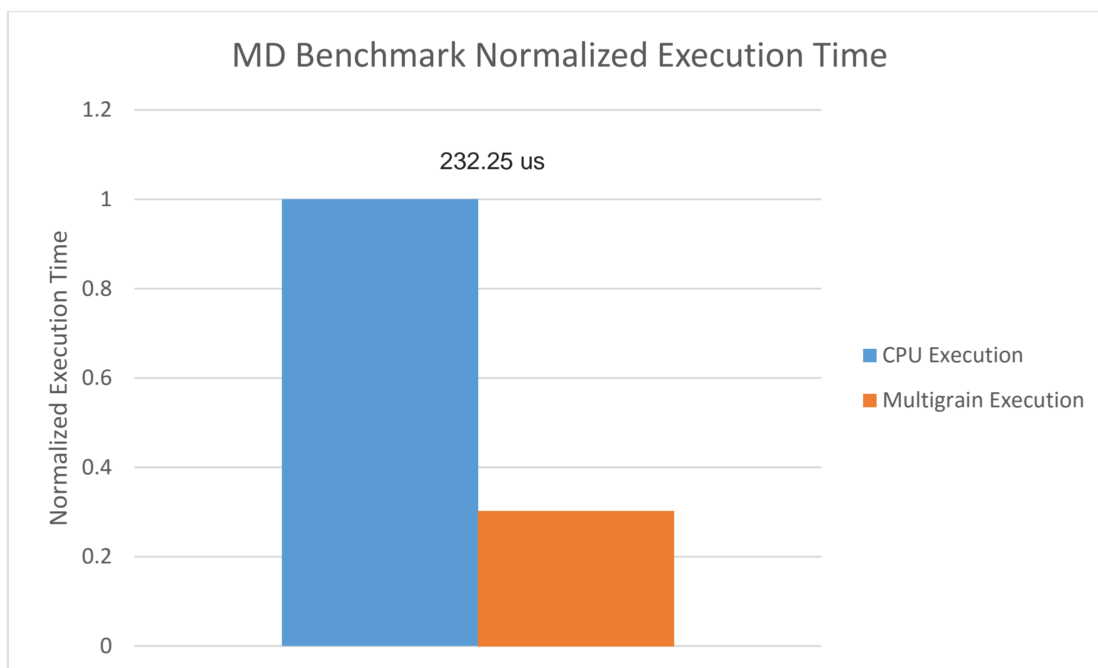


*Figure 14 MD Benchmark Normalized Execution Time*

| Molecular Dynamics | GPU Cycles | CPU Cycles |
|---|---|---|
| | 84295 | 557403 |

*Table 2 MD Benchmark Execution Time in Cycles*

The MD Loop, shown below, ran almost 3 times as fast on the GPU as compared to the CPU.

<u>FFT6</u>

FFT6 is also a benchmark from the OpenMP Source Code Repository. Three nested regions of interest were found in this benchmark. The resulting run times are shown below.

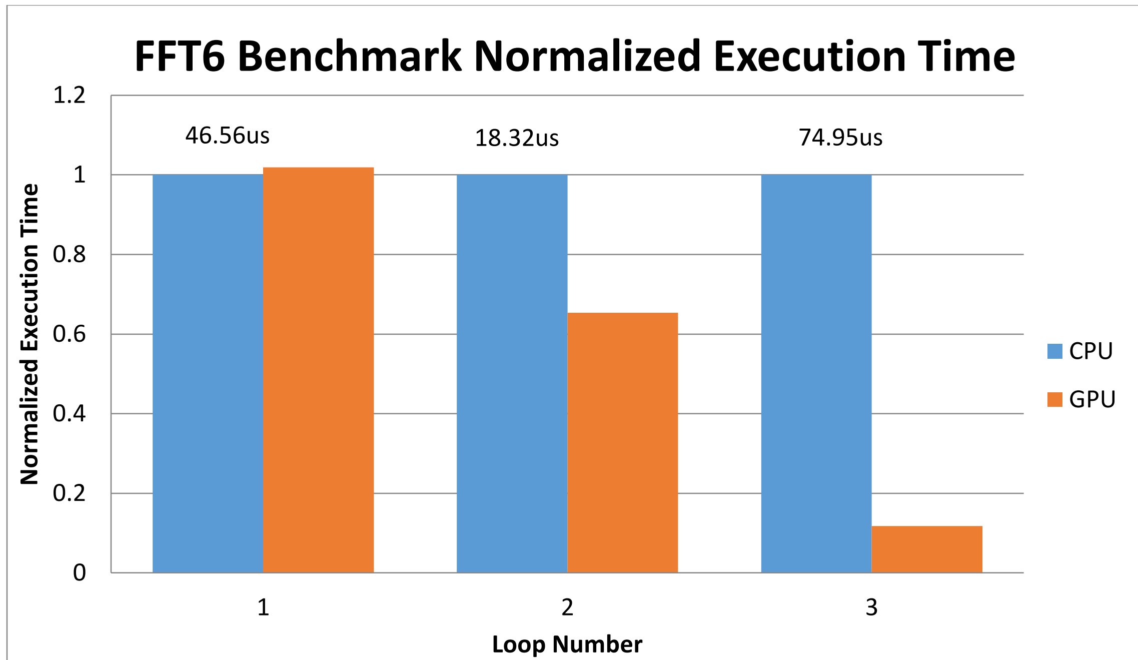


*Figure 15 FFT6 Benchmark Normalized Execution Time*

| FFT6 | GPU Cycles | CPU Cycles |
|---|---|---|
| Loop1 | 47419 | 111743 |
| Loop2 | 11973 | 43967 |
| Loop3 | 8836 | 179868 |

*Table 3 FFT6 Benchmark Execution Time in Cycles*

The first nested region, which runs with nearly the same speed on both the CPU and GPU is slowed on the GPU due to divergence. In this loop, shown below, the if statement splits the iterations into two separate control flows significantly delaying the execution time.

```
/* bit reverse step */
for (i = 0; i < n; ++i) {
  j = brt[i];
  if (i < (j-1)) {
    temp = a[j - 1];
    a[j - 1] = a[i];
    a[i] = temp;
  }
}
```

*Figure 16 FFT6 Loop 1*

The remainder of the loops showed improvement. The second loop is limited due to its size. This loop, called the butterfly computation, starts with the largest amount of work in the outer loop, then declining by log(n) each time. The decreased available work limits execution time gains over the CPU which excels at serial code.

```
/* butterfly computations */
ijDiff = 1;
stride = 2;
spowerOfW = ndv2;
for (stage = 0; stage < logn; ++stage) {
  /* Invariant: stride = 2 ** stage
     Invariant: ijDiff = 2 ** (stage - 1) */
  first = 0;
  for (powerOfW = 0; powerOfW < ndv2; powerOfW += spowerOfW) {
        pw = w[powerOfW];
    /* Invariant: pwr + sqrt(-1)*pwi = W**(powerOfW - 1) */
    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;
    }
    ++first;
  }
  ijDiff = stride;
  stride <<= 1;
  spowerOfW /= 2;
}
```

*Figure 17 FFT6 Loop 2*

The final region runs over 4x as fast on the GPU. This is due to the fine grain and high iteration work available in the inner loop.

```
for (i = 0; i < n; ++i) {
  for (j = 0; j < n; ++j) {
        index = i * n + j;
    aa = a[index];
        vv = v[index];
        a[index].re = aa.re * vv.re - aa.im * vv.im;
        a[index].im = aa.re * vv.im + aa.im * vv.re;
  }
}
```

*Figure 18 FFT6 Loop 3*

*Conclusions*

For these several simulated workloads, we see a mixed set of results. The majority of the workloads (6 ART kernels, MD, and 2 FFT6 kernels) showed decreased execution time when executed by the GPU, however three of the kernels did not improve. Accurately exploiting multigrain parallelism would allow for modest execution time improvement for two thirds of the kernels tested, however without an accurate heuristic to determine which kernels to execute on which core type, there could be a net negative effect, such as in the case of ART. It is clear some sort of scheduling algorithm will need to be developed if multigrain parallelism is to be effectively exploited. However, it appears there is the potential for significant runtime improvement through exploiting these constructs.

## Concurrency Analysis

To more fully analyze the efficacy of the single and dual parallel execution models, a custom analysis that modeled the heterogeneous architecture is necessary. Currently there are few joint CPU/GPU simulators, such as GEM5 and multi2sim, but neither of these are heterogeneous simulators, nor would they support the execution models we propose without significant alterations [28, 34]. Without existing simulator support, we designed a concurrent execution analysis model where our execution models could be tested. We then applied this analysis to three multigrain loops to begin to analyze the execution models. A description of the analysis we performed, followed by the results of our experiments, are given below.

*Concurrency Analysis*

We analyzed the full heterogeneous system by combining the results from Simple Scalar and GPGPU-sim to generate a model of a heterogeneous system. This analysis was developed to emulate a heterogeneous system, which combined 8 GTX480 style streaming multiprocessors, with a 4 core aggressively out of order CPU. The GPU simulator and CPU simulator from before were reused in the heterogeneous analysis infrastructure. For a more in depth description of the cores used, refer back to the CPU simulator and GPU simulator sections of the thesis.

*Assumptions Made*

We selected 4 CPU cores based on experimentation with our server. On running each of the selected regions, we found that openMP did not exceed 4 threads on the CPU side to execute the selected parallel portions. We assumed linear speedup for these 4 cores, which is an optimistic assumption, however after running the openMP regions on our servers ranging between 1 and 4 threads, we found that a nearly linear speedup factor (>.95) was seen for each of these parallel regions, making the assumption fairly realistic. We also assumed that the highest level of memory shared is the DRAM. This is pessimistic as most proposed and common heterogeneous cores have some sort of shared memory structure prior to DRAM (i.e. L2 cache is shared). This adds significant memory latency to the execution time.

In order to maintain cache coherency, we also took the most pessimistic assumption, assuming no available cache coherency mechanism and entirely flushing the cache upon a transition from GPU to CPU or vice versa. This requires the CPU and GPU to entirely repopulate the cache every time program control is reestablished.

*Functionality*

The designed simulator operated by breaking the program down into sections by the possibility of forwarding code on each section line to either the CPU or GPU. The simulator then determined the IPC at all points for these sections on both the CPU and GPU. For example, ART is fragmented into a section around each of the 8 identified inner loops, along with a section before the target openMP region, one after the target openMP region until the multigrain loops, one after the inner loops until the end of the target openMP region, and one region after the target openMP region. This produced 12 regions, 8 of which could be targeted at the GPU. For these regions, the Simple Scalar simulator described earlier was used to simulate each region, followed by GPGPU-sim if the region could be GPU targeted. To make the GPU simulations accurate with the possibility of 4 CPU cores forwarding kernels at any time, GPGPU-sim simulated all kernel arrival combinations in all forwarding orders and used results from the actual forwarding scenario simulated by the concurrency simulator.

The arrival order has a significant effect on execution time in Nvidia GPUs due to the first come, first serve scheduler onboard. Their scheduler gives the first arriving kernel the maximum usable resources. Kernels forwarded at a later time can only access the remaining resources unused by the previously forwarded kernels. There is no adjustment for underutilization of resources or effective allocation by the Nvidia scheduler.

With the generated IPC and instruction information for each of the code segments, the simulator steps through cycle by cycle and updates the current segment of code being executed as necessary.

Although the IPC number used is simply an average over the section, because section granularity is used and the CPUs are independent, the CPU IPC should remain identical to cycle by cycle simulator performance over each section. For the GPU, due to the multiple kernels being executed, the IPC numbers incorporate error due to different kernels being in different stages of execution when other kernels are launched on the GPU. We believe the error introduced by this is minimal due to the short kernels with often only a single computation being performed making the position in the kernel irrelevant with respect to GPU IPC.

*Determining Which Core Runs Code*

For these experiments, code is forwarded to the CPU or GPU based on static runtime results. This scheduling heuristic guarantees the optimal scheduling of code sections and is therefore an optimistic assumption in the current dynamically scheduled world. Applying some simple heuristics, such as setting a loop iteration cutoff at 500, setting a minimum loop instruction count, or simply not forwarding loops to the GPU with divergent instructions and behavior, all yield the same segment scheduling as the ideal case. As we are not making any comments on the scheduler involved in heterogeneous architectures and all obvious scheduling heuristics yield nearly ideal results for the selected regions, this assumption does not significantly skew the results.

*Concurrency Experiment*

The concurrency simulator was designed to get full benchmark execution time estimates for multigrain parallel benchmarks. Three of the benchmarks were

simulated using the concurrency simulator, Art, MD, and FFT6. For each of these benchmarks, we ran 4 separate tests, numbered in the results as #1, #2, #3, and #4 with each number corresponding to a unique execution model. These tests correspond as follows:

1. Sequential execution time. All code is run sequentially on the CPU only.
2. Sequential CPU, parallel GPU execution time. All code is run sequentially on the CPU, but with a single GPU that nested regions can be forwarded to. This is the mono parallel execution model discussed earlier.
3. Linear speedup CPU, individual GPU. All code is run in parallel on the CPU, assuming linear speedup when adding cores. Each core has access to a GPU which is not shared.
4. Linear speedup CPU, parallel GPU (with contention). All code is run in parallel on the CPU, assuming linear speedup when adding cores. Each core has access to a single shared GPU, introducing contention between the cores. This is the dual parallel execution model discussed earlier.

We evaluated each of these results for each benchmark. The results are discussed below.

### ART

Art was profiled using the concurrency simulator. The results are normalized to serial execution time and shown in the figure and table below.

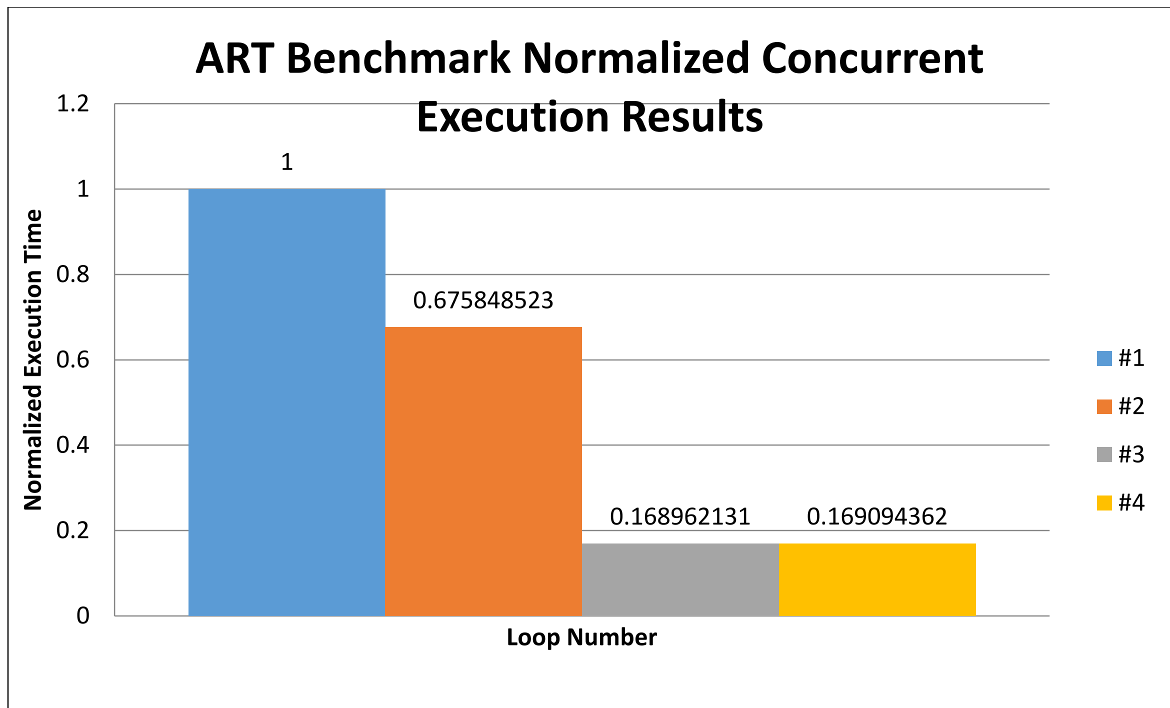


*Figure 19 ART Benchmark Normalized Concurrent Execution Results*

| Execution Model | Execution Time (Cycles) | Normalized Execution Time |
|---|---|---|
| #1 | 14179704 | 1 |
| #2 | 9583332 | 0.675848523 |
| #3 | 2395833 | 0.168962131 |
| #4 | 2397708 | 0.169094362 |

*Table 4 ART Benchmark Concurrent Execution Results*

From the results, you see a speed up of roughly 73% for the dual execution model over a serial execution model. You also find very little contention for the GPU in the ART benchmark. Contention is measured as the difference between #3 and #4, or the difference between all cores sharing a GPU versus having their own individual GPU. From the results, you see a less than 2000 cycle difference between #3 and #4 implying that GPU contention accounts for very little delay in execution time.

MD

MD was profiled using the concurrency simulator. The results are shown in the figure and table below.

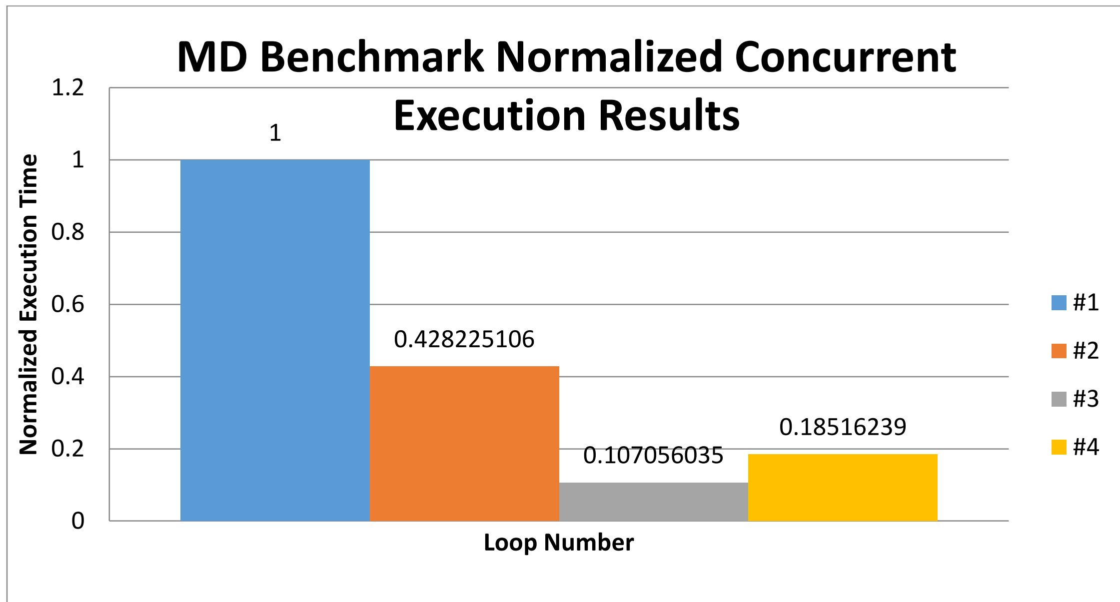


*Figure 20 MD Benchmark Normalized Concurrent Execution Results*

| Execution Model | Execution Time (Cycles) | Normalized Execution Time |
|---|---|---|
| #1 | 2071560 | 1 |
| #2 | 887094 | 0.428225106 |
| #3 | 221773 | 0.107056035 |
| #4 | 383575 | 0.18516239 |

*Table 5 MD Benchmark Concurrent Execution Results*

MD shows a 5x speedup with a dual execution model over a serial execution model. However, if each core had its own dedicated GPU a 10x speedup would be possible. This indicates that the GPU was overscheduled with all CPU cores forwarding to a single GPU because contention was a significant factor in execution time. Execution time would be decreased by a factor of nearly 2x with a larger GPU.

FFT6

FFT6 was profiled using the concurrency simulator. The results are shown in the figure and table below.

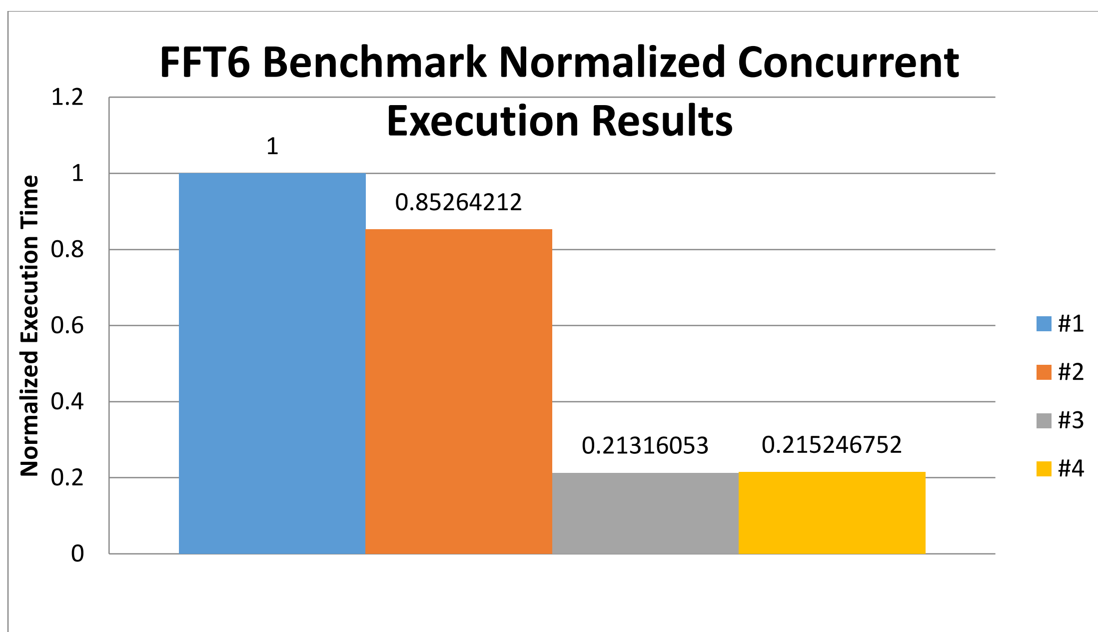


*Figure 21 FFT6 Benchmark Normalized Concurrent Execution Results*

| Execution Model | Execution Time (Cycles) | Normalized Execution Time |
|---|---|---|
| #1 | 778920 | 1 |
| #2 | 664140 | 0.85264212 |
| #3 | 166035 | 0.21316053 |
| #4 | 167660 | 0.215246752 |

*Table 6 FFT6 Benchmark Concurrent Execution Results*

FFT6 showed roughly 5x speedup using the dual execution model over sequential execution. There also was very little GPU contention, with less than a 2000 cycle difference between individual and shard GPUs.

## Conclusions

We initially began analysis by estimating the performance gain by offloading loops individually, shown as bar 2 in the graphs above. All bars normalized to the sequential execution bars, as shown in the figure below, provide speedup of 2.34x, 1.17x, and 2.17x for MD, FFT6 and ART respectively. The average execution time improvement is 1.89x. The speed-ups exhibited in these benchmarks are quite disappointing considering the speed-ups often seen when executing parallel code on a

GPU. However, as described before, these code segments are more complex. There is a mixture of SIMT and non-SIMT loops, making only a relatively small portion of each region targetable on a GPU. In the figure below, we display the normalized execution time as a function of the loops that can execute on the GPU. From viewing these figures, it is clear that there is a significant amount of code that runs more efficiently on the CPU, rather than the GPU. This creates a troubling instance of Amdahl's Law in which the GPU is only able to speed up a small portion of overall execution time. On top of this, some of the loops gainfully executed on the GPU do not exhibit massive speed-ups. These GPU targeted loops range from a 14.3x speedup, to a small 1.53x speedup. This effect is due to the complex nature of the loops, specifically small numbers of iterations (Loop 10), control flow divergence (Loop2), and non-unit stride memory accesses (Loop10, Loop14, Loop16, and Loop18). This further decreases the speed-ups seen in the #2 bars.

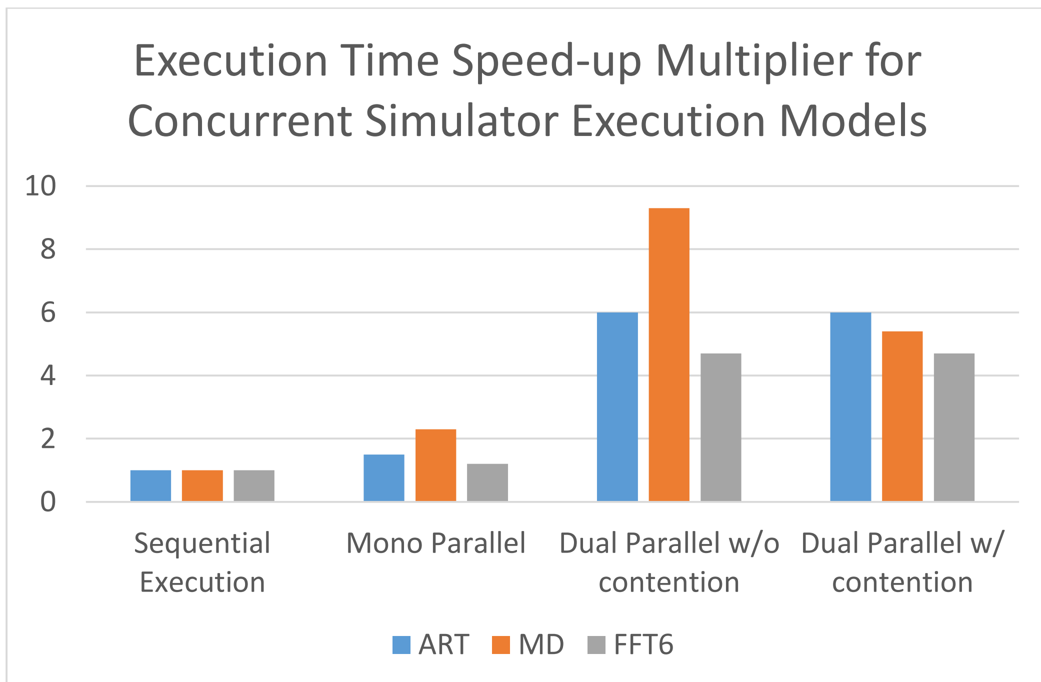


*Figure 22 : Execution Time Speed-up Multiplier for Concurrent Simulator Execution Models*

The next set of bars (#3 and #4 from concurrency analysis results) demonstrate a parallel trace, in which the openMP pragmas run on 4 CPU cores. Within the traces, we alternate between CPU cores and GPU cores, with #3 having a dedicated GPU, and #4 utilizing a shared GPU. As shown in the "Results" section of the paper, dual parallelism significantly outperforms the mono parallelism described above. Dual parallelism provided a speedup of 5.40x, 4.65x and 8.67x for MD, FFT6, and ART, respectively. By running these CPU cores in parallel, the Amdahl's law restriction encountered in the mono parallel execution is lessened where the GPU is significantly under-utilized. By making the CPU also operate in parallel, we are able to utilize the GPU to a greater degree. To compare the utilization of the GPU between mono and dual parallelism, we can compare the results in bar #3 in #4. Bar #3 gives each CPU thread a dedicated GPU in simulation, therefore demonstrating the ideal speedup of the parallel system. Bar #4 however does not provide the dedicated GPU to each CPU core. In Bar #4, contention will occur if the GPU is over-utilized. For FFT6 and ART, Bar #3 and #4 are nearly identical, indicating almost no contention. For MD, the ideal speedup is 1.73x faster, indicating that there is a GPU overutilization and therefore a contention issue. Even in the case where contention causes a problem, dual parallelism still out-performs mono parallelism by more than 2x.

## Related Works

Heterogeneous microprocessors have been commercially released by Intel since the Sandy Bridge architecture [19] along with AMD since the release of their fused accelerated processing unit (APU) architecture [5]. A fair amount of research

has been done into heterogeneous microprocessors and these cores. Daga et al [9] and Spafford et al [31] evaluated AMD's Fusion architecture [5], examining communication bottlenecks faced by discrete systems [15]. They however only considered traditional GPU workloads from the SHOC [10] and HPC Challenge [11] benchmark suites. They did not investigate new workloads enabled by the architectural differences introduced by these heterogeneous systems. Building on that, Arora et al [1] included several SPEC CPU 2000/2006 workloads [32] in their research along with the traditional GPU workloads, namely Rodinia [8]. The SPEC CPU benchmarks contain more irregular, traditionally CPU-targeted workloads, similar to those involved in our research; however, Arora focused on the effect of heterogeneity on the CPU cores, not the GPU cores as we focused on in our work. Arora showed that CPUs will execute more serial code as parallel loops are off-loaded to the GPU. Our research intends to find the balance between CPU and GPU execution by developing new parallelization techniques and exploiting the CPU-GPU communication interfaces more effectively.

The scheduling of code between the CPU and GPU is another related area. GPUs are specialized processors that do not perform well on all types of code; therefore, segments of our research focus on the scheduling of loops between the GPU and the CPU. Research groups have created performance models [2, 16, 17] for these heterogeneous processors and off-line profiling [22, 30] to statically schedule loops. Dynamic scheduling has also been investigated, such as [20] where each loop is scheduled on both the CPU and GPU cores the first time the loop is encountered. Subsequent executions of the loop are then delegated based on initial performance.

Dynamic work stealing [29, 30] was also investigated to schedule workloads on heterogeneous systems. The primary difference between these techniques and our proposed dual parallel technique is that they only schedule work from a single loop, thereby exploiting only homogenous parallelism. Dual parallelism exploits both types of parallelism from both loops, known as heterogeneous parallelism. This technique is much more effective when the parallelism in each loop is limited.

This multigrain parallelism we exploit is related to dynamic thread-block launch (DTBL) [35], which provides hardware support in the GPU such that a loop running on the GPU can initiate another instance of itself. This is effective in workloads with an initially indeterminate size, such as BFS. DTBL allows the GPU to create work dynamically as each new node is visited. This is similar to multigrain parallelism because DTBL enables multiple instances of a loop to execute simultaneously on a CPU. This increases the chip utilization. The difference, however, is that DTBL is not targeted for heterogeneous microprocessors. It only creates parallelism for the GPU; therefore, it is homogenous parallelism as opposed to the multigrain parallelism we employ which forks onto both CPU and GPU cores. DTBL can also only exploit SIMT parallelism, whereas multigrain parallelism can handle non-SIMT parallelism also.

## Future Work

We hope to continue this research in two separate directions. First, we hope to develop a heterogeneous computing benchmark suite. Currently, GPU research utilizes suites developed for discrete GPU systems such as Rodinia, Parboil, and

SHOC [8, 10, 33]. Because they target discrete GPU systems, they include massively parallel computations for which existing parallelization techniques are effective. These benchmarks do not accurately represent the workloads that will be opened to parallelization on heterogeneous systems. In order to explore heterogeneous computing systems, new benchmarks are needed. We hope to find enough programs to form a benchmark suite which highlights the capabilities of heterogeneous microprocessors including both task-level parallel and data-level parallel programs.

Second, we hope to apply the benchmarks we have developed and the suite we hope to develop on heterogeneous architectural simulators. Currently, there are two open source simulators capable of modeling heterogeneous microprocessors that we have considered: gem5-gpu [28] and Multi2Sim [34]. Regardless of the simulator we use, nothing available allows us to simulate the proposed multigrain parallelism execution models we investigated in this research. Therefore, significant simulator modifications will be required to allow proper data movement and execution models. We hope to develop an architectural simulator capable of accurately modeling these execution models so that we may further investigate it as a parallelization technique.

# Appendices

```
sim-outorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support.  This simulator is a performance simulator, tracking the
latency of all pipeline operations.

# -config                     # load configuration from a file
# -dumpconfig                 # dump configuration to a file
# -h                    false # print help message
# -v                    false # verbose operation
# -d                    false # enable debug message
# -i                    false # start in Dlite debugger
-seed                       1 # random number generator seed (0 for timer seed)
# -q                    false # initialize and terminate immediately
# -chkpt               <null> # restore EIO trace execution from <fname>
# -redir:sim           <null> # redirect simulator output to file (non-interactive only)
# -redir:prog          <null> # redirect simulated program output to file
-nice                       0 # simulator scheduling priority
-max:inst           200000000 # maximum number of inst's to execute
-fastfwd                    0 # number of insts skipped before timing starts
# -ptrace              <null> # generate pipetrace, i.e., <fname|stdout|stderr> <range>
-fetch:ifqsize              4 # instruction fetch queue size (in insts)
-fetch:mplat                3 # extra branch mis-prediction latency
-fetch:speed                1 # speed of front-end of machine relative to execution core
-bpred                  bimod # branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod     2048 # bimodal predictor config (<table size>)
-bpred:2lev      1 8192 12 1 # 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:comb      8192 # combining predictor config (<meta_table_size>)
-bpred:ras                 64 # return address stack size (0 for no return stack)
-bpred:btb       512 4 # BTB config (<num_sets> <associativity>)
# -bpred:spec_update       <null> # speculative predictors update in {ID|WB} (default non-spec)
-decode:width               4 # instruction decode B/W (insts/cycle)
-issue:width                4 # instruction issue B/W (insts/cycle)
-issue:inorder          false # run pipeline with in-order issue
-issue:wrongpath         true # issue instructions down wrong execution paths
-commit:width               4 # instruction commit B/W (insts/cycle)
-ruu:size                 128 # register update unit (RUU) size
-lsq:size                  64 # load/store queue (LSQ) size
-cache:dl1       dl1:128:32:4:l # l1 data cache config, i.e., {<config>|none}
-cache:dl1lat               1 # l1 data cache hit latency (in cycles)
-cache:dl2       ul2:1024:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:dl2lat              20 # l2 data cache hit latency (in cycles)
-cache:il1       il1:512:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1lat               1 # l1 instruction cache hit latency (in cycles)
-cache:il2                dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2lat              20 # l2 instruction cache hit latency (in cycles)
-cache:flush            false # flush caches on system calls
-cache:icompress        false # convert 64-bit inst addresses to 32-bit inst equivalents
-mem:lat         18 2 # memory access latency (<first_chunk> <inter_chunk>)
-mem:width                  8 # memory access bus width (in bytes)
-tlb:itlb        itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb        dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-tlb:lat                   30 # inst/data TLB miss latency (in cycles)
-res:ialu                   4 # total number of integer ALU's available
-res:imult                  2 # total number of integer multiplier/dividers available
-res:memport                4 # total number of memory system ports available (to CPU)
-res:fpalu                  2 # total number of floating point ALU's available
-res:fpmult                 2 # total number of floating point multiplier/dividers available
# -pcstat              <null> # profile stat(s) against text addr's (mult uses ok)
-bugcompat              false # operate in backward-compatible bugs mode (for testing only)
```

*Figure 23 Simple Scalar Out of Order Configuration Parameters*

```
# functional simulator specification
-gpgpu_ptx_instruction_classification 1
-gpgpu_ptx_sim_mode 0
-gpgpu_ptx_force_max_capability 20


# SASS execution (only supported with CUDA >= 4.0)
-gpgpu_ptx_convert_to_ptxplus 0
-gpgpu_ptx_save_converted_ptxplus 0

# high level architecture configuration
-gpgpu_n_clusters 2
-gpgpu_n_cores_per_cluster 1
-gpgpu_n_mem 6
-gpgpu_n_sub_partition_per_mchannel 2

#-gpgpu_clock_domains <Core Clock>:<Interconnect Clock>:<L2 Clock>:<DRAM Clock>
# In Fermi, each pipeline has 16 execution units, so the Core clock needs to be divided
# by 2. (GPGPU-Sim simulates a warp (32 threads) in a single cycle). 1400/2 = 700
-gpgpu_clock_domains 700.0:700.0:700.0:924.0

# shader core pipeline config
-gpgpu_shader_registers 32768

# This implies a maximum of 48 warps/SM
-gpgpu_shader_core_pipeline 1536:32
#-gpgpu_shader_core_pipeline 1104:23
-gpgpu_shader_cta 8
-gpgpu_simd_model 1

# Pipeline widths and number of FUs
# ID_OC_SP,ID_OC_SFU,ID_OC_MEM,OC_EX_SP,OC_EX_SFU,OC_EX_MEM,EX_WB
-gpgpu_pipeline_widths 2,1,1,2,1,1,2
-gpgpu_num_sp_units 2
-gpgpu_num_sfu_units 1

# Instruction latencies and initiation intervals
# "ADD,MAX,MUL,MAD,DIV"
-ptx_opcode_latency_int 4,13,4,5,145
-ptx_opcode_initiation_int 1,2,2,1,8
-ptx_opcode_latency_fp 4,13,4,5,39
-ptx_opcode_initiation_fp 1,2,1,1,4
-ptx_opcode_latency_dp 8,19,8,8,330
-ptx_opcode_initiation_dp 8,16,8,8,130


# In Fermi, the cache and shared memory can be configured to 16kb:48kb(default) or 48kb:16kb
#
<nsets>:<bsize>:<assoc>,<rep>:<wr>:<alloc>:<wr_alloc>,<mshr>:<N>:<merge>,<mq>:**<fifo_e
ntry>
# ** Optional parameter - Required when mshr_type==Texture Fifo
-gpgpu_cache:dl1  32:128:4,L:L:m:N,A:32:8,8
-gpgpu_shmem_size 49152

# The alternative configuration for fermi in case cudaFuncCachePreferL1 is selected
#-gpgpu_cache:dl1  64:128:6,L:L:m:N,A:32:8,8
```

```
#-gpgpu_shmem_size 16384

# 64 sets, each 128 bytes 8-way for each memory sub partition. This gives 786KB L2 cache
-gpgpu_cache:dl2 64:128:8,L:B:m:W,A:32:4,4:0,32
-gpgpu_cache:dl2_texture_only 0

-gpgpu_cache:il1 4:128:4,L:R:f:N,A:2:32,4
-gpgpu_tex_cache:l1 4:128:24,L:R:m:N,F:128:4,128:2
-gpgpu_const_cache:l1 64:64:2,L:R:f:N,A:2:32,4

# enable operand collector
-gpgpu_operand_collector_num_units_sp 6
-gpgpu_operand_collector_num_units_sfu 8
-gpgpu_operand_collector_num_in_ports_sp 2
-gpgpu_operand_collector_num_out_ports_sp 2
-gpgpu_num_reg_banks 16

# shared memory bankconflict detection
-gpgpu_shmem_num_banks 32
-gpgpu_shmem_limited_broadcast 0
-gpgpu_shmem_warp_parts 1

-gpgpu_max_insn_issue_per_warp 1

# interconnection
-network_mode 1
-inter_config_file config_fermi_islip.icnt

# memory partition latency config
-rop_latency 120
-dram_latency 100

# dram model config
-gpgpu_dram_scheduler 1
# The DRAM return queue and the scheduler queue together should provide buffer
# to sustain the memory level parallelism to tolerate DRAM latency
# To allow 100% DRAM utility, there should at least be enough buffer to sustain
# the minimum DRAM latency (100 core cycles).  I.e.
#   Total buffer space required = 100 x 924MHz / 700MHz = 132
-gpgpu_frfcfs_dram_sched_queue_size 16
-gpgpu_dram_return_queue_size 116

# for Fermi, bus width is 384bits, this is 8 bytes (4 bytes at each DRAM chip) per memory partition
-gpgpu_n_mem_per_ctrlr 2
-gpgpu_dram_buswidth 4
-gpgpu_dram_burst_length 8
-dram_data_command_freq_ratio 4  # GDDR5 is QDR
-gpgpu_mem_address_mask 1
-gpgpu_mem_addr_mapping
dramid@8;00000000.00000000.00000000.00000000.0000RRRR.RRRRRRRR.BBBCCCCB.CCSS
SSSS

# GDDR5 timing from hynix H5GQ1H24AFR
# to disable bank groups, set nbkgrp to 1 and tCCDL and tRTPL to 0
-gpgpu_dram_timing_opt "nbk=16:CCD=2:RRD=6:RCD=12:RAS=28:RP=12:RC=40:
                  CL=12:WL=4:CDLR=5:WR=12:nbkgrp=4:CCDL=3:RTPL=2"
```

```
# Fermi has two schedulers per core
-gpgpu_num_sched_per_core 2
# Two Level Scheduler with active and pending pools
#-gpgpu_scheduler two_level_active:6:0:1
# Loose round robbin scheduler
#-gpgpu_scheduler lrr
# Greedy then oldest scheduler
-gpgpu_scheduler gto

# stat collection
-gpgpu_memlatency_stat 14
-gpgpu_runtime_stat 500
-enable_ptx_file_line_stats 1
-visualizer_enabled 1

# power model configs
-power_simulation_enabled 1
-gpuwattch_xml_file gpuwattch_gtx480.xml

# tracing functionality
-trace_enabled 0
-trace_components WARP_SCHEDULER,SCOREBOARD
-trace_sampling_core 0
```

*Figure 24 GPGPU-Sim Configuration File*

# Bibliography

[1] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE MICRO*, pages 4–16, November/December 2012.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3]Bailey, David H., et al. "The NAS parallel benchmarks." *International Journal of High Performance Computing Applications* 5.3 (1991): 63-73.

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

[5] Bakhoda, Ali, et al. "Analyzing CUDA workloads using a detailed GPU simulator." *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009.

[6]Baskaran, Muthu Manikandan, Jj Ramanujam, and P. Sadayappan. "Automatic C-to-CUDA code generation for affine programs." *Compiler Construction*. Springer Berlin Heidelberg, 2010.

[7]Blelloch, Guy E., et al. "Implementation of a portable nested data-parallel language." *Journal of parallel and distributed computing* 21.1 (1994): 4-14.

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

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

[7] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

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

[8] Che, Shuai, et al. "A performance study of general-purpose applications on graphics processors using CUDA." *Journal of parallel and distributed computing* 68.10 (2008): 1370-1380.

[9] M. Daga, A. M. Aji, and W. chun Feng. 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*, 2011.

[10] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, Pittsburgh, PA, March 2010.

[11] J. Dongarra and P. Luszczek. Introduction to the HPC Challenge Benchmark Suite. Technical report, University of Tennessee-Knoxville, 2005.

[12]Duran, Alejandro, Marc Gonzàlez, and Julita Corbalán. "Automatic thread distribution for nested parallelism in OpenMP." *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005.

[13] I. Gelado, J. Cabezas, N. Navarro, J. E. Stone, S. Patel, and W. mei Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating System*, Pittsburgh, PA, March 2010.

[14] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dy SER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro*, 32(5), September-October 2012.

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

[16] D. Grewe, Z. Wang, and M. F. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, Shenzhen, China, February 2013.

[17] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. *SIGARCH Computer Architecture News*, 38(3):280–289, June 2010.

[18]Hummel, Susan Flynn, and Edith Schonberg. "Low-overhead scheduling of nested parallelism." *IBM Journal of Research and Development* 35.5.6 (1991): 743-765.

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

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

[21] G. Kyriazis. Heterogeneous System Architecture: A Technical Review. August 2012.

[22] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[23]Lee, Seyong, Seung-Jai Min, and Rudolf Eigenmann. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization." *ACM Sigplan Notices* 44.4 (2009): 101-110.

[24] A. Moshovos. Region Scout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

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

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

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

[28] J. Power, J. Hestness, M. Orr, M. D. Hill, and D. A. Wood. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *Computer Architecture Letters*, 13(1), January 2014.

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

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

[31] K. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter. The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures. In *Proceedings of the ACM International Conference on Computing Frontiers*, Cagliari, Italy, May 2012.

[32] Standard Performance Evaluation Corporation. *http://www.spec.org/benchmarks.html.* 2015.

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

[34] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, September 2012.

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