

Heterogeneous Task Scheduling for Accelerated OpenMP

Thomas R. W. Scogland* Barry Rountree† Wu-chun Feng* Bronis R. de Supinski†

* *Department of Computer Science, Virginia Tech, Blacksburg, VA 24060 USA*

† *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551 USA*
tom.scogland@vt.edu rountree@llnl.gov feng@vt.edu bronis@llnl.gov

Abstract—Heterogeneous systems with CPUs and computational accelerators such as GPUs, FPGAs or the upcoming Intel MIC are becoming mainstream. In these systems, peak performance includes the performance of not just the CPUs but also all available accelerators. In spite of this fact, the majority of programming models for heterogeneous computing focus on only one of these. With the development of Accelerated OpenMP for GPUs, both from PGI and Cray, we have a clear path to extend traditional OpenMP applications incrementally to use GPUs. The extensions are geared toward switching from CPU parallelism to GPU parallelism. However they do not preserve the former while adding the latter. Thus computational potential is wasted since either the CPU cores or the GPU cores are left idle. Our goal is to create a runtime system that can intelligently divide an accelerated OpenMP region across all available resources automatically. This paper presents our proof-of-concept runtime system for dynamic task scheduling across CPUs and GPUs. Further, we motivate the addition of this system into the proposed *OpenMP for Accelerators* standard. Finally, we show that this option can produce as much as a two-fold performance improvement over using either the CPU or GPU alone.

Keywords—GPGPU; OpenMP; Programming models;

I. INTRODUCTION

Multicore processors are ubiquitous. Nearly everyone has the equivalent of a small cluster on their desk, or even in their phone. Further, in the race to exascale, hardware vendors are offering a myriad of heterogeneous computational devices and systems that use them as accelerators. Many applications written for these accelerators, or ported to them, use them to the exclusion of the CPU cores. Alternatively most parallel codes are well parallelized on the CPU but ignore accelerators. Many of these could benefit from using the combined potential of both CPU and GPU resources together. As we move towards exascale, extracting maximum performance from all resources in a given node will be crucial to maintaining strong scaling, rather than just continuing weak scaling as we add more nodes.

One can certainly spread an algorithm across both CPU and GPU using CUDA, OpenCL, or the OpenMP accelerator directives to send work to the GPU, and pthreads, OpenMP or OpenCL for the CPU cores. However, this approach

currently requires a programmer either to program in at least two different parallel programming models, or to use one of the two that support both GPUs and CPUs. Multiple models however require code replication, and maintaining two completely distinct implementations of a computational kernel is a difficult and error-prone proposition. That leaves us with using either OpenCL or accelerated OpenMP to complete the task.

OpenCL's greatest strength lies in its broad hardware support. In a way, though, that is also its greatest weakness. To enable one to program this disparate hardware efficiently, the language is very low level, comes with a steep learning curve and many pitfalls related to performance across hardware as well as an almost complete lack of bindings for languages not directly based on C. Given an existing OpenCL application, dividing an application across the devices in a system should be simple: divide the inputs and accumulate the outputs. Unfortunately, managing the data transfers, multiple CPU threads, and ensuring that the code functions correctly and runs quickly on different hardware remains a daunting task.

Accelerated OpenMP in contrast is designed to allow a user familiar with basic OpenMP programming to port their code to accelerators with relative ease. It offers a more digestible and familiar syntax, especially for Fortran programmers, while remaining capable of significant performance gains. When it comes to using both CPU and accelerator however, the current state-of-the-art implementations offer little support for splitting a workload across multiple compute units concurrently. We propose the addition of new options to accelerated OpenMP, designed to split accelerated regions across available devices automatically.

Our ultimate goal is to enable OpenMP programmers to experiment with coscheduling, combining CPUs with GPUs, without having to re-create the work necessary to split and to load balance their computation. This approach requires the compiler and runtime system to (1) split regular OpenMP accelerator loop regions across compute devices and (2) manage the distribution of inputs and outputs while preserving the semantics of the original region transparently. We investigate the creation of such a runtime system and the requirements to automate the process. Specifically we present a case study that uses a development version of

Cray’s GPU accelerated OpenMP. For the purpose of this paper, we use accelerator and GPU interchangeably, although we could apply our approach to any platform that offers similar OpenMP accelerator extensions.

We make the following contributions:

- Extensions to OpenMP accelerator directives to support co-scheduling;
- Four novel scheduling policies for splitting work across CPUs and accelerators;
- An implementation of those extensions that is built on top of the OpenMP runtime and, thus, applicable to any implementation of the OpenMP accelerator directives;
- An evaluation that demonstrates our extensions significantly improve performance over only using an accelerator.

Our results for four programs that use the OpenMP accelerator directives demonstrates that our approach can produce as much as a two-fold performance improvement over using either the CPU or GPU alone.

The rest of the paper is arranged as follows. Section II provides a background in accelerator programming and the OpenMP Accelerator Directives. Section III describes the design of our proposed heterogeneous scheduling extension. Details of our proof-of-concept implementation follow in Section IV. We present results and discussion in Section V.

II. BACKGROUND

This section provides background material with a particular focus on three items. First, we review accelerator programming in general, with a focus on GPUs. A description of OpenMP for accelerators follows. Finally, we describe the previously proposed version of region splitting and task scheduling for this system.

A. Accelerator Programming

To understand the extensions in accelerated OpenMP, one needs a basic background in accelerator programming. Many types of accelerators exist, from very CPU-like accelerators that share the cache hierarchy, to completely separate devices that must be treated as though they were compute nodes unto themselves. Arguably the most popular type of accelerator is the GPU, which is highly divergent from the standard SMP programming model assumed by OpenMP.

Using terminology coined for NVIDIA’s CUDA architecture, GPUs consist of several multiprocessors, each of which contains several cores, 8 to 32 depending on the generation up to this point. An NVIDIA Tesla C2050, for example, consists of 14 multiprocessors, each with 32 cores. These cores follow the Single Instruction Multiple Thread (SIMT) execution model in which all cores on a multiprocessor must run the same instruction each cycle although the hardware hides the details of computing each branch independently from the software if threads diverge.

In essence, this mechanism allows a SIMD processor to be programmed as though it were MIMD.

Unlike CPU cores, GPU cores do not have direct access to main memory but rather access a hierarchy of dedicated, on-board memory modules. Each multiprocessor has a set of caches and *shared memory*, a local memory space that only threads run on that multiprocessor can access. The only memory space that multiprocessors share is *global memory*, some parts of which can also be used as read-only texture memory. Our example architecture, the Tesla C2050, contains three gigabytes of global memory. While all cores share the global memory, similarly to how CPU cores share main memory, the GPU memory system does not offer coherence guarantees. Changes to global memory are only guaranteed to be globally visible after either the termination of a kernel (i.e., a GPU function), or a call to the `thread_fence()` GPU function.

Programming a GPU effectively requires exploitation of many of these architectural details. The most significant issue is the separate address space from the CPUs, which means it cannot be treated as a shared-memory entity. Instead, we must program it as a distributed memory system. Essentially, we must send and receive messages to load data to the GPU, to retrieve results, or to synchronize. Much like transfers between nodes in a cluster, these transfers are costly and require care to ensure that no more is transferred than necessary. OpenMP for accelerators, which was designed to address these issues, strikes a balance between expressiveness and familiarity of syntax.

B. Accelerator Extensions

The OpenMP accelerator directives are proposed extensions to OpenMP that parallelize OpenMP regions across GPUs or other accelerators. For the purpose of this work, we use the version proposed by James Beyer et al. [1]. Another set of directives with similar goals are available from PGI as the PGI accelerator directives [2]. Although our work builds directly on a working prototype of the former from Cray, our method is generally applicable and should perform similarly with the PGI version.

The extensions add three main concepts to OpenMP: accelerator regions, input/output identifiers, and array shaping. An accelerator region generates a team of threads on an accelerator to process the region, analogous to a parallel region. Input/Output identifiers specify how to transfer data in and out of a region with greater specificity than shared and private. Accelerators, such as GPUs, with on-board discrete memory require explicit memory movement, as discussed in Section II-A. The identifiers support specification of that movement explicitly. Array shaping specifies the dimensions, range and stride of arrays. These shapes are passed with pointers to the input/output clauses to bound unbounded types in C or to transfer only the necessary part of arrays.

```

#pragma omp parallel for          \
        shared(in1,in2,out,pow)
for (i=0; i<end; i++){
    out[i] = in1[i]*in2[i];
    pow[i] = pow[i]*pow[i];
}

```

(a) Standard OpenMP

```

#pragma acc_region_loop          \
        acc_copyin(in1[0:end],in2[0:end]) \
        acc_copyout(out[0:end])          \
        acc_copy(pow[0:end])            \
        hetero(<cond>,                 \
               <iterations for CPU>)
for (i=0; i<end; i++){
    out[i] = in1[i] * in2[i];
    pow[i] = pow[i]*pow[i];
}

```

(c) Accelerated with hetero clause

```

#pragma acc_region_loop          \
        acc_copyin(in1[0:end],in2[0:end]) \
        acc_copyout(out[0:end])          \
        acc_copy(pow[0:end])
for (i=0; i<end; i++){
    out[i] = in1[i] * in2[i];
    pow[i] = pow[i]*pow[i];
}

```

(b) Accelerated OpenMP

```

#pragma acc_region_loop          \
        acc_copyin(in1[0:end],in2[0:end]) \
        acc_copyout(out[0:end])          \
        acc_copy(pow[0:end])            \
        hetero(<cond>[,<scheduler>[,<ratio>\
                  [,<div>]]])
for (i=0; i<end; i++){
    out[i] = in1[i] * in2[i];
    pow[i] = pow[i]*pow[i];
}

```

(d) Proposed hetero clause

Figure 1: OpenMP accelerator directive comparison

Figure 1a shows a loop parallelized for the CPU with OpenMP. Figure 1b is the same loop parallelized across a GPU with an `acc_region_loop` directive. We also add the `acc_copy()`, `acc_copyin()` and `acc_copyout()` clauses, which specify that values must be copied in and out, just in, or just out. Each clause accepts a list of variables or shaped arrays of the form `array[<start>:<end>:<stride>]`. These extensions preserve the clarity and syntax of OpenMP while allowing the use of local distributed memory accelerators.

The third code segment in Figure 1c includes a clause that was part of the draft standard for the OpenMP accelerator directives. This clause is of the form `hetero(<cond>, <width>)` where `cond` is a boolean expression, true to split, false to use only the accelerator, and `width` is the number of iterations to assign to the CPU. It does not provide for scheduling options however, and assumes that the application programmer will explicitly specify the number of loop iterations to run on the CPU, the others to be run on a single accelerator. Further, a more recent draft no longer includes the option. We expect that the option will be useful with some adjustment and increased runtime support; we propose our version in Section III.

III. DESIGN

This section presents the abstract design of our proposed system and its schedulers. First we describe the overall structure and then discuss the three classes of schedulers and their overall merits. The first class is `static`, analogous to but distinct from the `hetero` clause. The second supports dynamic scheduling with an intentional deviation from the traditional OpenMP dynamic scheduler inputs. Our third type of scheduler is two special case scheduling policies that combine the static and dynamic policies to handle common behavior of accelerated applications.

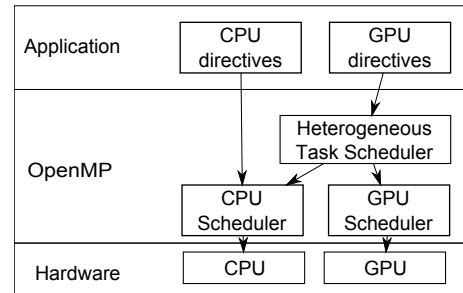


Figure 2: Our proposed software stack

A. Overview

We do not propose to replace part of the existing software stack but rather to add a new one. As Figure 2 shows, we intend our work to be a new layer between the OpenMP accelerator directives and the existing CPU and GPU schedulers, which leverages those existing schedulers to handle the details of each device. We focus on assigning tasks to a compute device, which we define as an entity that can be targeted by a `parallel` or `acc region`, i.e., a single GPU or all CPU cores rather than an individual core. Since we work at the region layer, our design applies to any architecture for which an implementation of the accelerator directives is available.

Since we target heterogeneous resources, compute devices may have completely disparate performance characteristics. Standard OpenMP schedulers use the size of a chunk to split the work across cores. For example, given a loop of 500,000 iterations one might add `schedule(dynamic, 500)` to their parallel loop, which would cause each thread to receive 500 iterations, compute those, and check for another chunk of 500 to compute. However, the optimal chunk size depends on the performance of the underlying devices and the

cost to distribute new work to them. Given a CPU based system, chunks of size 500 may perform well, but assigning 500 iterations to an entire GPU will usually take so little time to execute that overhead dominates, wasting potential computation time. Conversely, a chunk large enough for the GPU can run so long on a CPU as to dominate the program execution time before it finishes the first chunk.

Our solution does not use chunks. Instead we specify a ratio that captures the amount of work that a CPU can complete in the time it takes for a GPU to finish 100 units. This schedule is essentially a form of unbalanced static scheduling, like those proposed by Ayguade et al. [3]. For example, if the CPU device (i.e., all CPU cores) completes 100 iterations in the time it takes for the GPU to complete 500 the ratio would be 17%. Alternatively, if the CPU is more suitable for a particular problem and completes 200 iterations in the time that it takes the GPU to handle 100 the ratio would be 67%. Thus, we specify the relationship between the compute units, rather than trying to find a single sensible unit of work to assign to both.

Our scheduler operates at the boundaries of a region rather than within it, except in special cases, much like the DVFS decisions made in Adagio [4], which provided the inspiration for this type of interpass dynamic approach. This choice is another concession to the overhead of GPU kernel launches: by making scheduling decisions only once each pass through a region we generate only one thread team per compute unit rather than having to recreate them repeatedly. It also allows us to synchronize memory at the beginning and end of the region and not between, in turn saving synchronization time. The user expects that all memory is consistent at the end of the region. The most basic example is that the output arrays on the CPU specified by the `acc_copy` and `acc_copyout` must be consistent with the output of running the full problem set on the GPU, but updates must also be pushed to GPU memory, failure to do either can cause unexpected side-effects.

B. Static Splitting

Static, which is the default scheduler, divides tasks at the beginning of the region. Each entry into the region runs one CPU team and one GPU team, using the underlying static schedulers for each. As noted above, we split based on a ratio. The CPU receives $i_c = i_t * r$ iterations where i_t is the total number of iterations requested and r is the ratio. the GPU receives the remainder $i_g = i_t - i_c$.

The ratio argument is optional; its default value is a non-trivial problem. We compute the default ratio at runtime based on the compute resources found to be available, making the assumption that the workload is floating point computation bound. The goal is for the ratio to express the percentage of the total floating point work that the CPU device can perform in a unit of time. Unfortunately, most compute hardware does not expose a software API to query

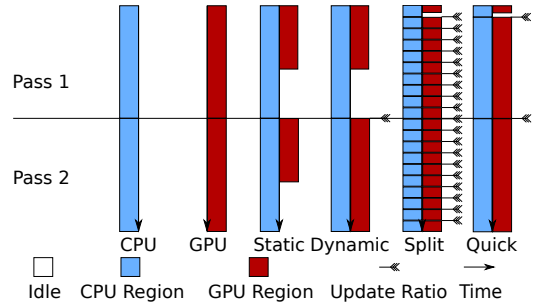


Figure 3: Scheduler behavior over time

its peak flops directly, so we must approximate based on something more accessible.

We essentially need to know how many floating point operations each compute device can evaluate in a given unit of time. Further, we know that on a current generation GPU, each core can compute one floating point instruction per cycle, for now assuming single precision. The CPU is a more complicated. Each core on a CPU can compute anywhere from one to its SIMD width floating point instructions per cycle. We assume that floating point operations in the region are mostly vectorizable so the CPU can retire its full SIMD width in each cycle, which overestimates the CPU somewhat. This overestimation helps to balance another assumption: both the CPU and GPU operate on the same frequency. The final equation is $r = c_c * 4 / (c_c + c_g)$ where c_c is the number of CPU cores and c_g is the number of GPU cores. This default is portable since we can detect the compute resources available on any given system and adjust to them. We find that this simple model performs well for compute-bound floating point intensive applications, but not for memory bound ones, or highly conditional applications, as we discuss further in Section V.

C. Dynamic Splitting

Similarly to our static scheduler, our dynamic scheduler deviates from the original OpenMP dynamic schedule policy. We make scheduling decisions only at the boundaries of accelerated regions. Thus, the dynamic scheduler assumes that the code will execute the parallel region several times. The first time, our approach executes the region as the static scheduler would. We measure the time taken to complete the assigned iterations on each compute unit. On all subsequent instances of the parallel region, we update the ratio based on these measurements.

Since we split at region boundaries rather than using a queue, we are subject to blocking time, during which one compute unit is waiting on the other to finish before they can pass the barrier at the end of the region. In order to minimize blocking time, we attempt to compute the ratio that causes the compute units to finish in as close to the same amount of time as possible. In order to predict the time for

I = total iterations in next pass (int)
 i_j = iterations on compute unit j in next pass (int)
 p_j = time/iteration for compute unit j from last pass
 n = number of compute devices
 t_j^+ = time over equal
 t_j^- = time under equal

Table I: Variable definitions for the linear program

$$\begin{aligned}
 i'_c * p_c &= i'_g * p_g \\
 i'_c * p_c &= (i'_t - i'_c) * p_g \\
 i'_c &= ((i'_t - i'_c) * p_g) / p_c \\
 i'_c &= ((i'_t - i'_c) * p_g) / p_c \\
 i'_c + (i'_c * p_g) / p_c &= (i'_t * p_g) / p_c \\
 (i'_c * p_c) / p_c + (i'_c * p_g) / p_c &= (i'_t * p_g) / p_c \\
 i'_c * (p_g + p_c) &= i'_t * p_g \\
 i'_c &= (i'_t * p_g) / (p_g + p_c) \quad (6)
 \end{aligned}$$

the next iteration, we assume that iterations take the same amount of time on average from one pass to the next. For the general case with an arbitrary number of compute units, we use a linear program for which Table I lists the necessary variables. Equation 1 represents the objective function, with the accompanying constraints in Equations 2 through 5.

$$\min\left(\sum_{j=1}^{n-1} t_1^+ + t_1^- \cdots + t_{n-1}^+ + t_{n-1}^-\right) \quad (1)$$

$$\sum_{j=0}^n i_j = I \quad (2)$$

$$i_2 * p_2 - i_1 * p_1 = t_1^+ - t_1^- \quad (3)$$

$$i_3 * p_3 - i_1 * p_1 = t_2^+ - t_2^- \quad (4)$$

⋮

$$i_n * p_n - i_1 * p_1 = t_{n-1}^+ - t_{n-1}^- \quad (5)$$

Expressed in words, the linear program calculates the iteration counts with predicted times that are as close as possible to identical between all devices. The constraints specify that the sum of all assigned iterations must equal the total number of iterations and that all iteration counts must be integers. Since we often have exactly two compute devices, we also use a reduced form that is only accurate for two devices but can be solved more efficiently. The new ratio is computed such that $t'_c = t'_g$ where t'_c is the predicted new time for the CPU portion to finish and t'_g is the predicted time to finish the GPU portion. When expanded we eventually get Equation 6, which can be solved in only a few instructions and produces a result within one iteration of the linear program for the common case.

D. Special Purpose

Our design and testing indicated that neither of the schedulers above is entirely appropriate in certain cases. Thus, we created two other schedulers: *split* and *quick*.

1) *Split*: Our dynamic scheduling requires multiple executions of the parallel region to compute an improved ratio, which works well for applications that make multiple passes. However, some parallel regions are executed only a few times, or even just once. Split scheduling addresses these regions. Each pass through the region begins a loop that iterates *div* times with each iteration executing *totaltasks/div* tasks. Thus, the runtime can adjust the ratio more frequently, and earlier, than with dynamic scheduling. More importantly, it can adjust the ratio in cases that dynamic scheduling cannot. The split schedule is analogous to the original OpenMP dynamic schedule since it specifies *totaliterations/chunksize* instead of chunk size directly. It remains distinct however in that while it runs a number of chunks, they can be independently subdivided to avoid overloading, or underloading, a compute device. Increasing the number of passes however, and thus the number of synchronization steps, increases overhead, which is especially problematic with short regions and those unsuitable for GPU computation so it is unsuitable as a definitive replacement for dynamic.

2) *Quick*: Quick is a hybrid of the split and dynamic schedules. It executes a small section of the first pass of size *iterations/div* just as split does, but the rest of that pass in one step of size *iterations - iterations/div*. It then switches to using the dynamic schedule for the rest of the run. It targets efficiently scheduling of applications with long running parallel regions that can be dominated by the first pass of the dynamic schedule when given a poorly chosen initial ratio. Quick is especially useful when such regions are executed repeatedly, making the split scheduler impractical due to its added overhead.

E. Schedules

We have alluded to the types of application that each schedule targets. Figure 4 shows the computational patterns, in terms of OpenMP regions, of three applications that we evaluate in Section V. Each of our three dynamic schedulers

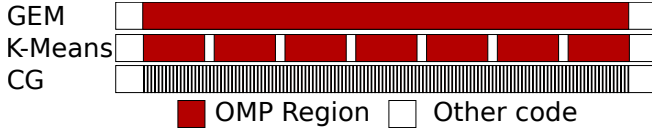


Figure 4: Computation patterns of evaluated benchmarks

targets one of these three cases, not just for these applications but as a general pattern of use. The first application type, with its single huge region, is a clear choice for the split schedule. The quick schedule targets the second, which has slightly smaller sections in which the segments are too long to allow an entire pass with a bad static mapping but do not require splitting every region to achieve load balance. Finally, we have applications that use fine grained regions. Any overhead dominate these regions, which do best with either quick or dynamic. Of course, we could use static for any of these cases as well, especially if we want to fine-tune the ratio manually.

IV. IMPLEMENTATION

We implement our concurrent heterogeneous support as a library that uses the OpenMP accelerator directives to motivate its addition in lower levels. This library encapsulates our scheduling functionality. We manually translate applications with minimal effort to function as if our proposed clause and schedules were used. Our implementation would be easy to integrate into a compiler, which represents the design in Section III. In the only significant difference from our design, our implementation currently only supports two devices at a time since our testing environment only offers two. We will implement the general case in future work. In addition to the implementation of the library, we also investigate what such loop splitting requires without underlying support.

A. The Splitter Library

In order to keep the implementation as general as possible, we design the library to be independent of the implementation of accelerated OpenMP. The library does not use accelerated regions or accelerated OpenMP functions or constructs, with the single exception of `omp_get_thread_limit()`, which we use to determine the number of available threads and to calculate the default static ratio. In the current version, we read the number of GPU cores from an environment variable, or assume it is 448, which is the number of cores in the NVIDIA Tesla C2050 GPU. While we would prefer to read the value from the underlying system, the OpenMP accelerator extensions do not include a function for this purpose.

The interface has six functions and a structure, as Figure 5 shows. The `split_init()` function initializes the library for a new region. It takes the arguments that would be given

```

splitter * split_init(int size, split_type sched,
                    double *rat, int *div)
splitter * split_next(splitter * s, int size,
                    int iteration)
void      split_cpu_start(splitter *s);
void      split_cpu_end(splitter *s);
void      split_gpu_start(splitter *s);
void      split_gpu_end(splitter *s);
typedef struct splitter{
    int cts; //CPU start iteration
    int cte; //CPU end iteration
    int gts; //GPU start iteration
    int gte; //GPU end iteration
    int d_end; //div
    int d_ccs; //start of CPU output
    int d_cce; //end of CPU output
    int d_gcs; //start of GPU output
    int d_gce; //end of GPU output
}splitter;

```

Figure 5: Splitter API for basic, CPU and single GPU, case

to the `hetero()` clause as well as the number of tasks to expect and it returns a structure to use with the splitter functions. After that, `split_next()` is evaluated at least once, populating the structure with the assignments for each device. Each pass through the region, `split_next()` restarts these counters, unless invoked with monotonically increasing iteration values, which is used to implement the split scheduler as we discuss shortly. The other four functions are timing calls that inform the library of the beginning and end of each split region.

In order to avoid repeated data transfers to and from the GPU in a pass, the library sends the entire data set for the region to the GPU and retrieves the entire output whether or not it is all used. Although this choice is inefficient, it is more efficient than copying piecemeal as the split between CPU and GPU is adjusted. Lower level APIs in future could make this choice unnecessary. Because we copy back the entire region, we also must use a temporary array to receive either the output from the CPU or GPU, and merge that into the main output array after both have finished. Otherwise consistency could not be assured. Use of this array could also be avoided at a lower level in future work.

Since we must merge the data, we attempt to merge as efficiently as possible. We accomplish this goal by having each compute device work from opposite ends of the iteration space toward the center. Thus, we divide the output only into two pieces regardless of any adjustments made during the run, unlike a simpler implementation that assigns chunks moving from one end to the other.

B. Using Splitter

We present an example to illustrate the use of the library. The code in Figure 6 shows the manually translated version of the code of a k-means kernel in Figure 7, which includes a `hetero` clause so it would correspond to the manually transformed version. This kernel is part of the code for the k-means implementation that we evaluate in Section V.

```

splitter * s = split_init(no, SPLIT_DYNAMIC, NULL, NULL);
int *m_c = (int*)malloc(sizeof(int)*no);
for(int d_it=0; d_it < s->d_end; d_it++)
{
    s = split_next(no, d_it);

#pragma omp parallel num_threads(2)
    {
        if(omp_get_thread_num()>0)
            //CPU OpenMP code
            split_cpu_start(s);
#pragma omp parallel shared(fo,fc,m_c,s) \
                    num_threads(omp_get_thread_limit()-1) \
                    firstprivate(no,ncl,nc) private(i)
            {
                #pragma omp for
                for (i=s->cts; i<s->cte; i++) {
                    m_c[i] = findc(no,ncl,nc,fo,fc,i);
                }
            }
            split_cpu_end(s);
        }else{//GPU OpenMP code
            split_gpu_start(s);
            int gts = s->gts, gte = s->gte;
#pragma omp acc_region_loop private(i) \
                    firstprivate(nc,nc,ncl,gts,gte) \
                    acc_copyin(fc[0:ncl*nc]) \
                    acc_copyout(m[0:no]) \
                    present(fo) default(none)
            for (i=gts; i<gte; i++) {
                m[i] = findc(no,ncl,nc,fo,fc,i);
            }
            split_gpu_end(s);
        }
    }
}
memcpy(m+s->d_ccs,m_c+s->d_ccs,
       (s->d_cce-s->d_ccs)*sizeof(int));
free(m_c);

```

Figure 6: Manually transformed k-means kernel

```

#pragma omp acc_region_loop private(i) \
                    firstprivate(nc,nc,ncl) default(none) \
                    acc_copyin(fc[0:ncl*nc]) present(fo) \
                    acc_copyout(m[0:no])
// hetero(1,dynamic)
for (i=0; i<no; i++) {
    m[i] = findc(no,ncl,nc,fo,fc,i);
}

```

Figure 7: Accelerated OpenMP k-means region

We divide the translated code in two ways. First, we divide the region into a CPU region and a GPU region, each in one branch of a conditional, with each given one thread from an outer parallel region. Thus, the master thread runs the GPU region, a requirement as all other threads crash when encountering GPU regions in this version of the compiler, and the other thread starts a new team that uses the remaining compute resources on the CPU. We transform the computational loop in each of the two regions to use the start and end values specified by the splitter library, and bound it on either side by calls used for timing.

The other way in which we split the code is through the outer loop. That loop allows the scheduler to split the region into multiple sub-regions, by specifying how many iterations

it will pass, and thus how many times `split_next()` will be evaluated. At the end of the outer loop, the merging step is a simple `memcpy()` of the CPU calculated values from the extra array to the original output array.

While the manual transformation is conceptually simple, it is verbose. The original kernel is only 12 lines of code, including line wrapping, while the transformed version is 43. It also has two copies of the inner loop, forcing any change made in one to be replicated. For both of these reasons, even using a library such as ours to split a region at this level is tedious and error prone. Even so, being able to use all the compute resources available in a system is worthwhile.

V. EVALUATION

This section presents an evaluation of our proof-of-concept runtime library. All codes were compiled with a development version of the Cray Compiling Environment (CCE) compiler version 8.0 using optimization flag `-O3`. All tests were run on a single node containing a 2.2Ghz 12-core AMD Opteron 6174 processor and one NVIDIA Tesla C2050 GPU. No other processes, aside from the standard daemons, were allowed to run during the tests. All CPU results, unless otherwise specified, use all 12 available cores, in the case of runs using both the CPU and the GPU concurrently one core is reserved to manage the GPU, the other eleven are assigned to computing. Parameters to the scheduler are defaults unless otherwise specified; ratio is as defined in Section III-B and div is 10.

To evaluate our proof-of-concept, we have implemented OpenMP Accelerator directives versions of four applications, GEM [5], [6], [7], k-means, CG [8] and helmholtz which will be described in greater detail below. Each of the four presents a different pattern of execution, and different levels of fitness for GPU computing. In each case, the minimum transformation necessary to accelerate the code was used. In GEM, k-means, and helmholtz the transformation entailed exactly two lines of code. CG required more changes because of some undefined behavior when mixing regular OpenMP threads with GPU regions in Fortran, which we solved by porting the computational kernel of CG to C before accelerating it. Optimizations could certainly be applied, and the performance of each benchmark would benefit, but we are evaluating the framework, not the specific benchmark, and so leave this for future work. In addition to the OpenMP accelerator directive modifications, we applied the transformations necessary to hook into our region splitting library as we described in Section IV

For each application we collect computation time, as defined by the time to compute a result excluding problem setup and I/O. All transfers to and from the GPU are included, as is scheduling time and extra work necessary to split and to reassemble data to preserve the memory semantics of the region. In addition to computation time, we collect the number of iterations scheduled on the CPU and

GPU on each pass through the accelerated region, as well as the amount of time each spent running those iterations. From this we calculate the blocking time on a given pass as the time one compute unit must wait for another to finish, or $\max(\text{time}_{gpu}, \text{time}_{cpu}) - \min(\text{time}_{gpu}, \text{time}_{cpu})$. Finally, we track the application’s performance ratio, as we defined in Section III-B.

Each of our four benchmarks represents one of three types of computation patterns that Figure 4 illustrates. Our scheduling system works at the boundaries of OpenMP regions. Thus, how these regions are distributed, can have significant impact on the schedulers. Additionally, the length of each pass is quite important, as it determines how much overhead the runtime can afford to incur in launching each region. We characterize these issues by measuring three factors: the number of passes through the accelerated region in a run; the average time to complete a pass; and the native ratio, i.e., the ratio of CPU to GPU computation that produces the most balanced workload for the application. While the ratio may vary by input, all ratios for a given application tend to cluster into a relatively small region due to their suitability, or lack thereof, for GPU computation. Table 8b depicts these values for the default input set. Length is based on the time to run one pass while runtime measures the entire computation, both on only the GPU.

A. K-Means

K-Means is a popular clustering algorithm that uses an iterative method. Each iteration has two stages; the first calculates the nearest cluster for all data points and the second moves each cluster to the center of the data points that identified it as nearest earlier that time step. As a converging algorithm, it does not have a set number of passes as an application, but a given problem does. In our case, the number of passes is generally low in relation to the kernel execution time, but varying the dataset can vary these parameters. The dataset used for our tests consists of 1,210,000 points each with nine observations and groups those points into 500 clusters, requiring seven iterations to converge to a solution.

Figure 8a shows the results for our k-means test. For this input set and implementation the CPU and GPU are relatively evenly matched. K-means, is bound by one of two operations, memory accesses as part of computing the distance to every cluster, or the conditionals that check whether that is the nearest cluster. Consequently, it is not compute bound on the GPU, and the default ratio of 0.098, which effectively gives the CPU 9.8% of the work, is far off from its native ratio of 0.396. Thus, the default assigns excess work to the GPU and the static scheduler performs particularly poorly. The dynamic scheduler performs better, but suffers from the first pass being run entirely with that significantly poor ratio. Split, despite the added overhead, outperforms dynamic by reducing the time that we run

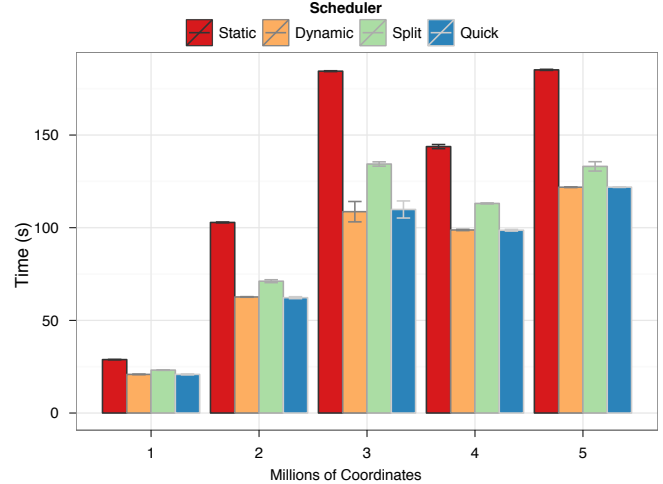


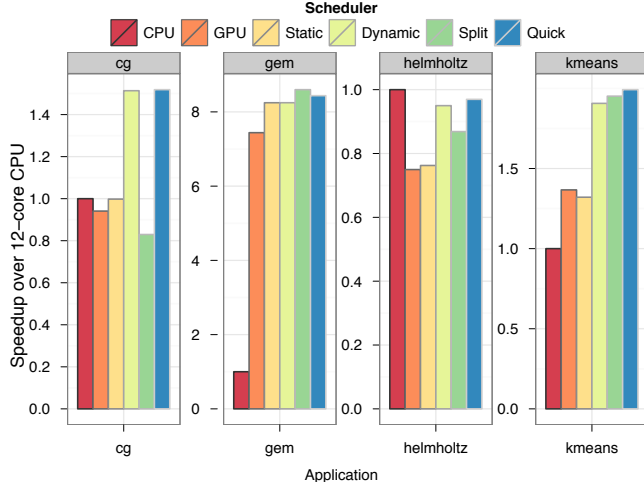
Figure 9: k-means performance on data sets in one million point increments

at the default ratio. This observation motivated our quick scheduler, which quickly makes the first adjustment while reducing overhead for the remainder of the run. In fact, quick produces an extra 10% performance improvement over split for this case.

K-means is unique among our four benchmarks in that its pattern of computation that Figure 4 shows can change significantly based on the input dataset. To study the effect of this change on the scheduler, we ran a variety of data sizes in Figure 9. These tests use varying size subsets of a 5 million point dataset with 10 observations per point and groups them into 100 clusters. The number of passes necessary for convergence, and length of a pass both vary as the data size varies. Despite the variability in the number and nature of the passes, the three dynamic schedulers show consistent behavior across all five reference sizes, although as the number of iterations for convergence grows, peaking in the three million point dataset, the split scheduler gets progressively worse, leaving the dynamic and quick schedulers as the clear options for k-means with quick being the most consistently fast across all tests.

B. CG

CG is the NAS Parallel Benchmarks implementation of the conjugate gradient method. This method is used to solve systems of linear equations by iterative refinement. While CG requires a relatively small number of steps to converge, each full iteration consists of a set of smaller steps internally. Thus, the number of passes of the accelerated region is large, and each pass is short. We use the C class for all our tests, which requires 75 iterations to converge, and runs 1900 passes through the accelerated region. Figure 8a shows that, switching to the GPU alone does not speed up CG over using the 12 CPU cores. This result is expected since



(a) Speedup over 12-core CPU for each application across schedulers

Program	Passes	Length (s)	Runtime (s)	Ratio
CG	1900	0.038	84.90	0.554
GEM	1	138.005	138.04	0.117
helmholtz	100	0.24	26.5	0.999
k-means	7	0.486	3.71	0.396

(b) Program characteristics as measured for the default dataset

Device/pass	1	2	3	4	5	6
CPU	392	2793	3637	3891	4000	4000
GPU	3608	1207	363	109	0	0

(c) Outer-loop iterations assigned to CPU and GPU per pass in helmholtz

Figure 8: Application performance and behavior for our four benchmarks

the data copying and kernel launch overhead inherent in the simple acceleration are significant. Even so, it almost matches 12 CPU cores, which in terms of performance per dollar is still worth it if both can be used together. The static scheduler sends insufficient work to the CPU, much as it does with k-means, although in this case, the performance is slightly improved regardless. Due to the very small pass size, the added overhead for split completely destroys the performance of the application, and the small extra overhead and early misprediction in quick cause it to be, on average, slightly lower performing than the dynamic scheduler.

C. GEM

GEM is a molecular modeling code developed by Fenley et al. [5] to study the electrostatic potential along the surface of biomolecules. It is a single step non-bonded force interaction problem that has a computational complexity of $a*v$ where a is the number of atoms in the biomolecule and v is the number of vertices, or points in the surface grid, to be computed. Since GEM is a single step problem, it runs large data sets through a single iteration of one region, as Figure 4 shows. For all GEM tests, we use the 2eu1 input, which consists of 109,802 atoms and 898,584 vertices.

The first striking feature of the GEM data that Figure 8a shows is that the GPU is significantly faster, about $7.5\times$, than all 12 CPU cores. This gain is consistent with prior studies [7]. However, including the 12 CPU cores, which are normally left idle, improves performance by approximately 10% over the GPU version. Also, the static scheduler performs well for GEM. The default ratio balances floating point performance and, thus, produces favorable results for floating point computation bound applications like GEM.

This application was the original impetus for the development of the split scheduler. Since it has only one very long

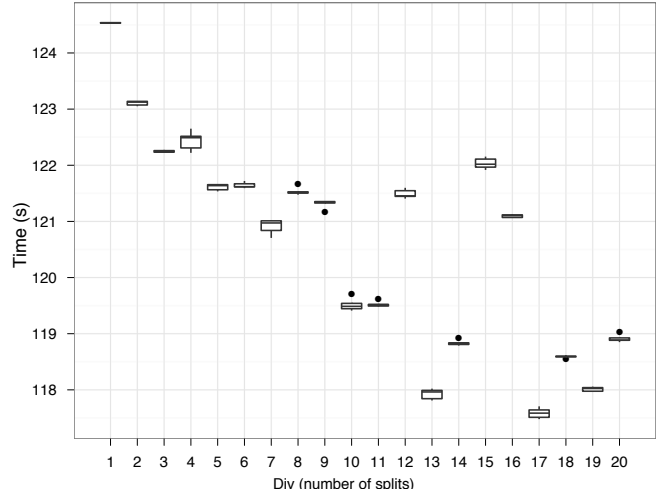


Figure 10: GEM performance at varying divs

running region, as Figure 4 depicts, the standard dynamic scheduler has no opportunity to adjust the ratio during the course of a run. We expected that the split scheduler would produce an improvement, but that the quick scheduler would provide even greater benefit due to its lower overhead and reduced synchronization. However, the split scheduler performs best, possibly because the quick scheduler runs a small chunk of a data and then extrapolates to the rest of the dataset, which could make it overly sensitive to overhead in thread team creation and kernel launching. In other applications, dynamic adjusts for to this cost after the second pass to hide it. However, GEM's single iteration provides no chance to fix the imbalance.

We also vary div with GEM to characterize its optimal split size. Figure 10 shows that performance increases until

a div of ten. Three specific larger divs cause performance degradation. We initially suspected that these tests exhibited system noise, but the results are repeatable. The degradations may be due to sub-optimal kernel launch decisions made by the underlying system similar to what happens when a poor block size decision is made directly in CUDA. In any event, the optimal choice for div is non-trivial, and presents an opportunity for future tuning.

D. Helmholtz

Helmholtz implements a solver for a discrete finite difference version of the Helmholtz equation, using the Jacobi iterative method. The implementation is similar in design to a map reduce in two dimensions that calculates the equation at all $n * m$ points in the space and reduces on the error residuals to test for convergence. The application has a single OpenMP region with a reduce clause, so the map and reduce are combined into one phase. The results for helmholtz in Figure 8a are materially different from our other applications. Communication overhead completely dominates the computation whenever a region is offloaded to the GPU. Even with a significant problem size, none of the GPU enabled versions could keep up with using only the CPU. The dynamic schedulers detect the issue and effectively turn off the GPU by not sending it any work after the first few iterations – an average of 5 iterations. Table 8c shows the number of outer-loop iterations assigned to CPU and GPU in the first six iterations of a run with the dynamic scheduler. With the advent of accelerators that require little to no copying overhead, such as AMD Fusion, we expect that the accelerator could compete with the CPU, and thus it may be useful to enable this feature even for this application when they become available. We leave optimizing the number of iterations for the scheduler to determine that the GPU is detrimental for future work.

E. Overall

The optimal scheduler depends on the application and, in some cases, the size of the dataset. K-means benefits most from the quick scheduler, GEM the split, and CG the basic dynamic. Since each scheduler benefits at least one pattern, these four schedulers offer a good starting point for dynamic task splitting in heterogeneous systems. Alternatively, a user could compute the ratio for a given application or save the ratio that a dynamic scheduler computes to derive a low overhead static schedule that achieves accurate splits as Figure 11 shows. In a case with regions small enough to be highly sensitive to overhead, as in the case of CG, tuned static scheduling can be highly effective. However, for GEM and k-means, the dynamic policy performs better, implying that the ratio is adjusted during the run based on current conditions to produce a better ratio. In addition to the ability to compute a better scheduler at runtime, computing the ratios for all input data sets and applications, not to

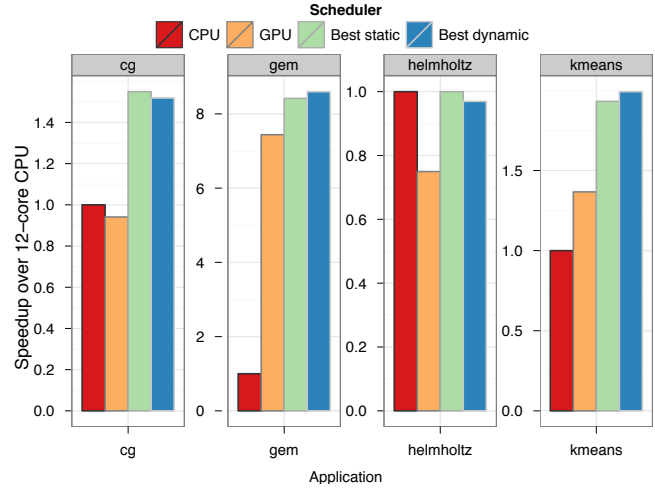


Figure 11: Speedup of best static and dynamic options over 12-core CPU

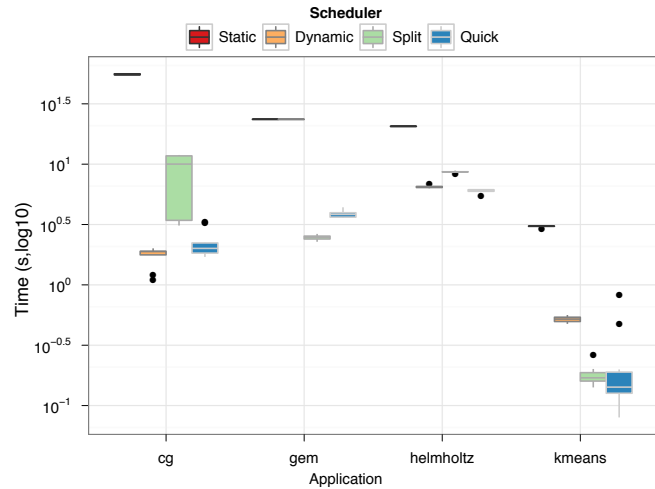


Figure 12: Total blocking time observed with each scheduler for all applications

mention hardware platforms, will not always be feasible, so we foresee a continued need for runtime dynamic splitting.

In addition to the overall performance of each application, we collect the length of time spent blocking waiting on either accelerator. As we discussed in Section III, our dynamic scheduler, and those that build on it, assign iterations in order to minimize blocking time. Figure 12 presents the total blocking time in each application scheduler combination. Overall these results follow our expectations: the blocking time usually corresponds to overall execution time behavior. CG presents a notable exception in that the static scheduler, while not performing well, performs significantly better than the split scheduler. However the blocking time of the split scheduler makes it appear more efficient possibly due to

an increase in the percentage of the accelerated region that transfers data and performs synchronization. As the regions become smaller, that overhead begins to dominate the execution time, slowing computation despite a balanced workload, as seen in the time per iteration on each device under the two schedulers. Under static, iterations average $0.39\mu s$ on the CPU and $0.26\mu s$ on the GPU. Under split with the default div of 10, iterations on the CPU average approximately the same time ($0.37\mu s$) but GPU iterations take much longer ($1.12\mu s$). The extra overhead increases the per iteration runtime by more than $4\times$. Thus, alternative metrics for the dynamic schedule may produce better results for this application.

VI. RELATED WORK

Automatic task scheduling is becoming increasingly popular for parallel computing. Traditionally the domain of shared memory parallel systems like OpenMP [9], Intel TBB [10] or even languages like Erlang, task scheduling has begun to move outside that realm with the rise of frameworks for cluster and cloud task based computing. Frameworks like Charm++ [11] for clusters, and MapReduce [12] for the cloud offer many of the same productivity benefits of their shared memory ancestors, and provide interesting testbeds for advances in scheduling and work partitioning. MapReduce in particular has seen a significant amount of work to become more aware of and amenable to heterogeneous computing resources. CellMR [13] presents a MapReduce framework for asymmetric cell based clusters. In response to a different kind of heterogeneity, in this case background tasks and node loss in a work stealing environment, Moon [14] provides a heterogeneous grid of work stealing resources with a small number of high-uptime cluster-like resources for improved performance and reliability.

Along with these platforms, a wide range of work explores task scheduling policies and their effects on performance and scalability. Ayguadé et al. question the need for a schedule clause in OpenMP, using knowledge of previous passes through a parallel region to influence the schedule of future passes to positive results, but showing in the end that none of the solutions is always optimal [3]. Their work influenced the design of our initial heterogeneous schedulers as it offers low overhead while remaining effective.

Accelerator-based heterogeneous computing, including GPU platforms from NVIDIA and AMD, is also on the rise. As these systems become more common, automatic task scheduling in heterogeneous systems has also become more common. StarPU [15] is designed to be a platform for heterogeneous task scheduling. Along with StarPU, Qilin [16], Scout [17], the dynamic load balancing system created by Chen et al. [18], and the work by Jiménez et al. [19] forms a solid foundation for both the need and the capability for a heterogeneous task scheduler. These solutions, however, require the user to reimplement their application – in a new

programming language in the case of StarPU or Scout; a new API in Qilin – or manually to create multiple copies of a function for multiple platforms to provide to the scheduler. In contrast, we bring heterogeneous task scheduling into an extension of OpenMP, which eases the translation of legacy scientific codes and makes the advances available immediately to users.

OmpSs, proposed by Duran et al. [20], attempts to offer an alternative to Accelerated OpenMP that also provides a coscheduling mechanism. The major difference is that in general one must still write GPU kernel code, CUDA or OpenCL, to use OmpSs. Interestingly, in terms of comparison with our results, coscheduling was consistently found to be slower than using two CPUs in OmpSs.

Other relevant work has come in the form of various studies that show code written and optimized for one GPU cannot be trusted to run equally well on other GPUs. Du et al. [21] for example conclude that performance across GPU architectures cannot be assumed to be portable and offer some methods to make it more portable. Even within a given GPU architecture and vendor, Archuleta et al. [22] show that different GPUs react differently to algorithmic and mapping changes. Each case calls the portability of accelerator performance into question. Our work also attempts to address this issue through a mechanism by which computation that does not run well on the GPU on a system is remapped automatically to use the CPU.

VII. CONCLUSIONS

In this paper we have presented our design for an automatic heterogeneous task scheduler for Accelerated OpenMP. We make four major contributions: the design of the extension; four scheduling policies to handle a variety of application behaviors; our case study implementation in the splitter library; and our evaluation across four representative scientific codes. Despite certain drawbacks inherent in our library implementation approach, we have shown speedups of as much as $2\times$ over using the CPU or GPU alone. We clearly demonstrate the utility of a `hetero()` clause in Accelerated OpenMP.

In future work we will implement our runtime and policies at a lower level, such as the compiler or potentially an extension. This implementation not only will make the system easier to use but it also will allow us to reduce memory transfer costs and to explore heterogeneous task scheduling further. We will investigate the ratio as a metric for the suitability of a given compute unit to a given application, and extend it to a more general model for scheduling work across a variety of accelerator platforms.

REFERENCES

- [1] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, "OpenMP for Accelerators," in *OpenMP in the Petascale Era*, B. M. Chapman, W. D. Gropp, K. Kumaran, and M. S. Miller,

- Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6665, pp. 108–121.
- [2] M. Wolfe, “Implementing the PGI Accelerator Model,” in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM Press, 2010, p. 43.
- [3] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera, “Is the Schedule Clause Really Necessary in OpenMP?” in *WOMPAT’03: Proceedings of the Workshop on OpenMP Applications and Tools 2003*. Springer-Verlag, Jun. 2003.
- [4] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, “Adagio: Making DVS Practical for Complex HPC Applications,” in *ICS ’09: Proceedings of the 23rd International Conference on Supercomputing*. ACM, Jun. 2009.
- [5] A. T. Fenley, J. C. Gordon, and A. Onufriev, “An Analytical Approach to Computing Biomolecular Electrostatic Potential. I. Derivation and Analysis,” *The Journal of Chemical Physics*, vol. 129, 2008.
- [6] J. C. Gordon, A. T. Fenley, and A. Onufriev, “An Analytical Approach to Computing Biomolecular Electrostatic Potential. II. Validation and Applications,” *The Journal of Chemical Physics*, vol. 129, p. 075102, 2008.
- [7] R. Anandkrishnan, T. R. Scogland, A. T. Fenley, J. C. Gordon, W.-c. Feng, and A. V. Onufriev, “Accelerating Electrostatic Surface Potential Calculation with Multi-Scale Approximation on Graphics Processing Units,” *Journal of Molecular Graphics and Modelling*, vol. 28, no. 8, pp. 904–910.
- [8] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, “The NAS Parallel Benchmarks,” *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.
- [9] L. Dagum and R. Menon, “OpenMP: An Industry Standard API for Shared-Memory Programming,” *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, Mar. 1998.
- [10] J. Reinders, “Intel Threading Building Blocks,” 2007.
- [11] L. V. Kale and S. Krishnan, “CHARM++,” in *OOPSLA ’93 Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1993, pp. 91–108.
- [12] J. Dean and S. Ghemawat, “MapReduce,” *Communications of the ACM*, vol. 51, p. 107, Jan. 2008.
- [13] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, “CellMR: A Framework for Supporting MapReduce on Asymmetric Cell-Based Clusters,” in *IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009*. IEEE, May 2009, pp. 1–12.
- [14] H. Lin, X. Ma, and W.-c. Feng, “Reliable MapReduce Computing on Opportunistic Resources,” *Cluster Computing*, Feb. 2011.
- [15] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures,” in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5704, pp. 863–874.
- [16] C. Luk, S. Hong, and H. Kim, “Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM Press, 2009, p. 45.
- [17] P. McCormick, J. Inman, J. Ahrens, J. Mohdyusof, G. Roth, and S. Cummins, “Scout: A Data-Parallel Programming Language for Graphics Processors,” *Parallel Computing*, Sep. 2007.
- [18] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, “Dynamic Load Balancing on Single- and Multi-GPU Systems,” *2010 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pp. 1–12, 2010.
- [19] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, “Predictive Runtime Code Scheduling for Heterogeneous Architectures,” in *High Performance Embedded Architectures and Compilers*, A. Sezec, J. Emer, M. OBoyle, M. Martonosi, and T. Ungerer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, vol. 5409, pp. 19–33.
- [20] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures,” *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.
- [21] P. Dua, R. Webera, P. Luszczeka, S. Tomova, G. Petersona, and J. Dongarraa, “From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming,” Technical Report CS-10-656, Electrical Engineering and Computer Science Department, University of Tennessee, Tech. Rep.
- [22] J. Archuleta, Y. Cao, T. Scogland, and W.-c. Feng, “Multi-Dimensional Characterization of Temporal Data Mining on Graphics Processors,” in *Parallel and Distributed Processing Symposium, International*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 1–12.