LLNL-CONF-502012

Heterogeneous Task Scheduling for Accelerated OpenMP

T. R. W. Scogland, B. Rountree, W. Feng, B. R. de Supinski

September 29, 2011

IEEE International Parallel & Distributed Processing Symposium (IPDPS)
Shanghai, China
May 21, 2012 through May 25, 2012

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 incrementally extend traditional OpenMP applications 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 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 to either 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.

This work was supported in part by a DoD National Defense Science & Engineering Graduate Fellowship (NDSEG). LLNL-CONF-502012

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 a near 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 and combinations of CPUs, and even potentially multiple GPUs, without having to recreate the work necessary to split and to load balance their computation. This 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. Our solution also must offer performance improvements while being easy to use. We investigate the creation of such a runtime system and the requirements of automating 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, such as Intel's MIC, that includes 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 which 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 of 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, as Figure 1 shows. 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.

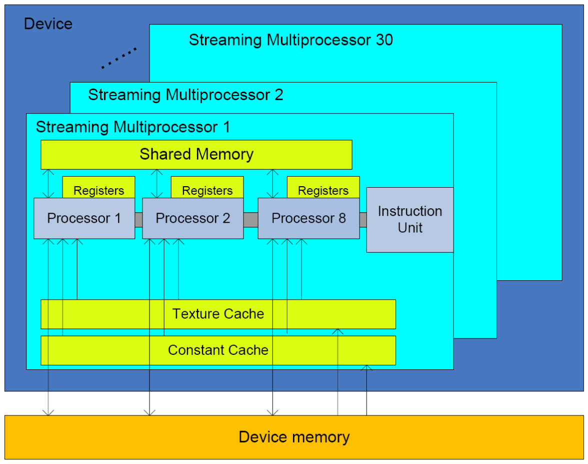


Figure 1: NVIDIA GPU architecture

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 coherency 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, with most significant issue being 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

```
#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])
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>,                      \
               <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])                  \
        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 2: OpenMP accelerator directive comparison

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.

Figure 2a shows a loop parallelized for the CPU with OpenMP. Figure 2b 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 2c 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. 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 schedulers are our two special case scheduling policies that combine aspects of the static and dynamic policies, each designed to handle a common behavior in accelerated applications. Finally, we discuss the overall merits of schedulers.

A. Overview

First, we are not proposing to replace part of the existing software stack but rather to add a new one. As Figure 3 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 should apply 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.

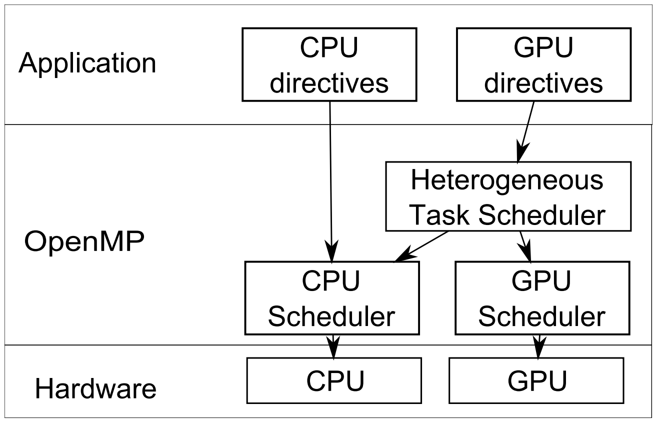


Figure 3: Our proposed software stack

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 as to be dominated by overhead, 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 a CPU can complete in the time it takes for a GPU to finish 100 units. 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 it takes the GPU to handle 100 the ratio would be 67%. In this way, 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. 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

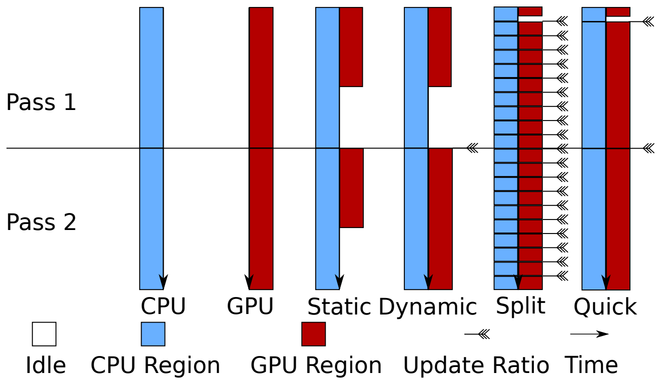


Figure 4: Scheduler behavior over time

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 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

$I =$	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

compute-bound floating point intensive applications, but not for memory bound ones, or highly conditional applications, 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 executes as it would have been by the static scheduler. 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 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 Figure I lists the necessary variables. Equation 1 represents the objective function, with the accompanying constraints in Equations 2 through 5.

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

$$\sum_{j=0}^{n} i_j = I \tag{2}$$

$$i_2 * p_2 - i_1 * p_1 = t_1^+ - t_1^- \tag{3}$$

$$i_3 * p_3 - i_1 * p_1 = t_2^+ - t_2^- \tag{4}$$

$$\vdots$$

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

Expressed in words, the linear program calculates the iteration counts with predicted times that are as close to identical between all devices as possible. 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.

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

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 more at least multiple executions of the parallel region to compute an improved ratio, which works well for applications which make multiple passes. However, some parallel regions are executed only a few times, or even just once. Split scheduling addresses these regions. In this case, 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

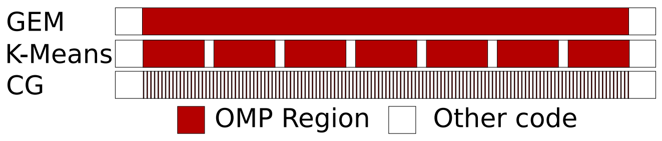


Figure 5: Computation patterns of evaluated benchmarks

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 5 shows the computational patterns, in terms of OpenMP regions, of three applications that we evaluate in Section V. Each of our three dynamic schedulers 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 which does 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 it as a library on top of the OpenMP accelerator directives, which motivates the addition of this functionality to lower levels. This library encapsulates all of our scheduling functionality. We manually translate applications with minimal effort to function as though our proposed clause and schedules were used. Our implementation is designed to be easy to integrate into a compiler and to represent the design in Section III. The only significant difference from our design is that our implementation currently only supports two devices at a time as a result of our testing environment, which 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

```
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 6: Splitter API for basic, CPU and single GPU, case

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 6 shows. The `split_init()` function initializes the library for a new region, taking the arguments that would be given to the `hetero()` clause and in addition the number of tasks to expect, returning 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 used to 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

```
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,nco) private(i)
            {
                #pragma omp for
                for (i=s->cts; i<s->cte; i++) {
                    m_c[i] = findc(no,ncl,nco,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(nco,no,ncl,gts,gte)\
                    acc_copyin(fc[0:ncl*nco])       \
                    acc_copyout(m[0:no])            \
                    present(fo)    default(none)
            for (i=gts; i<gte; i++) {
                m[i] = findc(no,ncl,nco,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 7: Manually transformed k-means kernel

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

Figure 8: Accelerated OpenMP k-means region

the run, unlike a simpler implementation that assigns from chunks moving from one end to the other.

B. Using Splitter

To illustrate the use of the library, we present an example. The code in Figures 7 shows the manually translated version of the code of a k-means kernel in Figure 8, which includes `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.

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, 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 [3], [4], [5], k-means, CG [6] 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 because of some undefined behavior when mixing regular OpenMP threads with GPU regions in Fortran, requiring us to port 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 directives modifications, we applied the transformations described in Section IV necessary to hook into our region splitting library.

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 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 $max(time_{gpu}, time_{cpu}) - min(time_{gpu}, time_{cpu})$. Lastly we track the application's performance ratio, as defined in Section III-B.

Each of our four benchmarks, represents one of three types of computation patterns as illustrated earlier in Figure 5. 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 this by measuring three factors, the number of passes through the accelerated region in a run, the average time to complete a pass and their native ratio, or the ratio of CPU to GPU computation which 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. For the default input set, these values are depicted in Table 9b, Length is based on the time to run one pass, and runtime the entire computation, both on only the GPU.

A. *K-Means*

K-Means is a popular clustering algorithm which uses an iterative method. There are two stages to each iteration, the first calculates the nearest cluster for all data points, the second moves each cluster to the center of the data points which 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 on a solution.

The results for our k-means test can be found in Figure 9a. For this input set and implementation the CPU and GPU are relatively evenly matched. K-means, is bounded by one of two operations, memory accesses as part of computing the distance to every cluster, or the conditionals checking whether that is the nearest cluster. Consequently, it is not compute bound on the GPU, and the default ratio of 0.098,

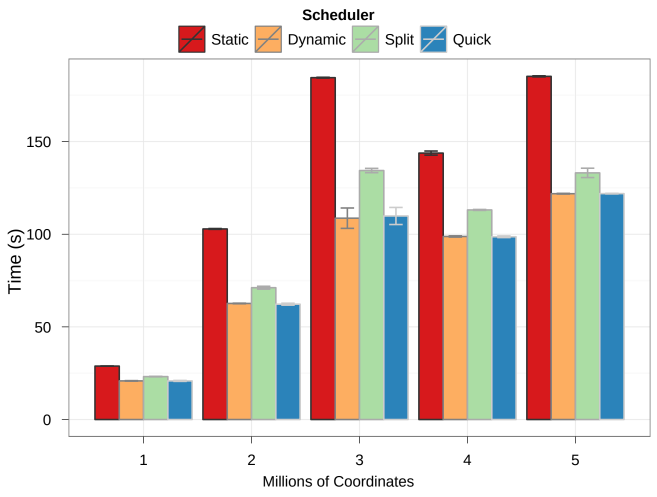


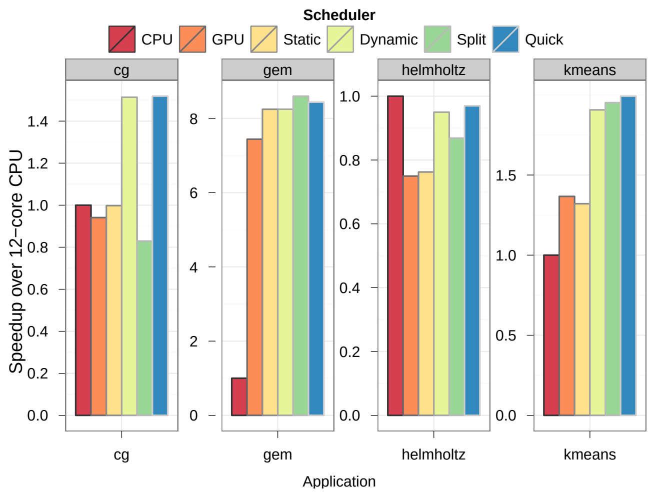
Figure 10: k-means performance on data sets in one million point increments

which effectively gives the CPU 9.8% of the work, is far off from its native ratio of 0.396, assigning excess work to the GPU. As a result, the static scheduler performs particularly poorly. The dynamic scheduler performs better, but suffers from the first pass being run entirely with that significantly off ratio. Split, despite the added overhead, outperforms dynamic by shortening the time running at the default ratio. It was this that motivated the creation of the quick scheduler, which allows the first adjustment to be made quickly, while reducing overhead for the remainder of the run, and in fact 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, as portrayed in Figure 5 can change significantly based on the input dataset. To study what effect this has on the scheduler, we ran a variety of data sizes in Figure 10. These tests used 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 has a relatively small number of steps to convergence,



(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 9: Application performance and behavior for our four benchmarks

each full iteration is actually made up of a set of smaller steps internally. As a result, the number of passes on the accelerated region is very large, and each is quite short. We use the C class for all our tests, which requires 75 iterations to converge, and runs 1900 passes through the accelerated region. As you can see from Figure 9a CG does not, by switching to GPU alone, see any speedup over the 12 CPU cores. That is somewhat to be expected, as 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 mis-prediction 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. [3] to study the electrostatic potential along the surface of biomolecules. It is a single step non-bonded force interaction problem, and 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. As a single step problem, GEM runs large data sets through a single iteration, and thus only a single region as depicted in Figure 5. For all GEM tests, we use a large input known as 2eu1, which consists of 109,802 atoms and 898,584 vertices.

The first striking feature of GEM's data, as shown in Figure 9a, is that the GPU is significantly faster than even all 12 CPU cores for this problem at about 7.5x. This is expected, and was in fact studied in [5], but even so including the extra 12 cores which are normally left idle improves performance by approximately 10% over the GPU version. Note also that the static scheduler performs well in this case, the default ratio is meant to balance floating point performance, and thus for floating point computation bound applications like GEM produces favorable results. Further, this application was the original impetus for the development of the split scheduler. Since it has only one very long running region, as depicted in Figure 5, the standard dynamic scheduler is denied the 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 overtake it due to its lower overhead and reduced synchronization. The results do not bear this out however, and the split scheduler continues to perform best. We believe this is a result of the fact that the quick scheduler, in running such a small chunk of the data and then extrapolating to the rest of the dataset, is overly sensitive to overhead in thread team creation and kernel launching. In other applications this is hidden by the fact that dynamic adjusts for it after the second pass, but in GEM there is no third, so the imbalance is never fixed.

We also took the opportunity to test GEM with a variety of div values in an attempt to characterize the optimal split size for this application. These results are presented in Figure 11. Overall, the performance increases until a div of ten, after that though, three specific divs cause performance degradation. Initially we suspected that these tests might be issues with system noise, but after repeating the test the same three remained anomalous. We believe these are caused by sub-optimal kernel launch decisions made by the underlying system similar to what happens when a poor block size

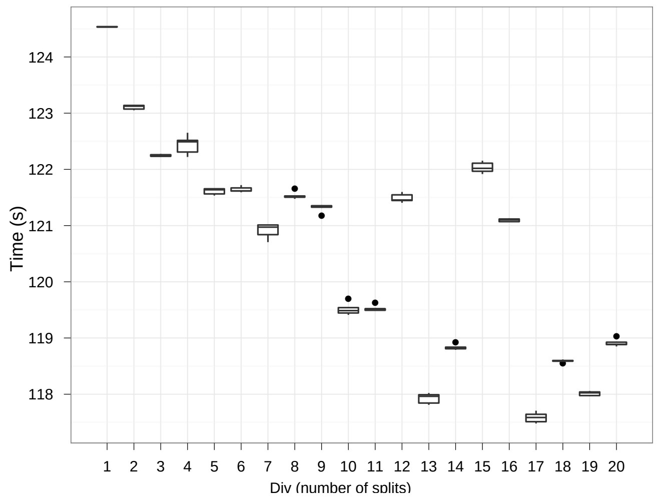


Figure 11: GEM performance at varying divs

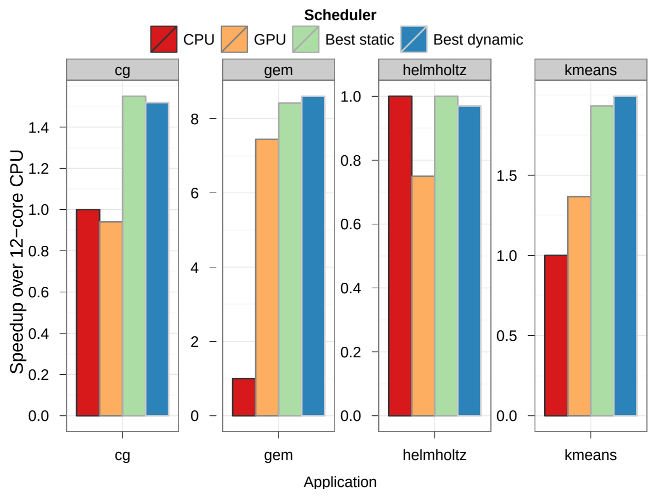


Figure 12: Speedup of best static and dynamic options over 12-core CPU

decision is made directly in CUDA. That aside, it is clear that the decision of what div to use 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 extremely similar in design to a map reduce in two dimensions, calculating the equation at all $n * m$ points in the space and reducing on the error residuals to test for convergence. All of this is implemented as a single OpenMP region with a reduce clause, so the map and reduce are combined into one phase. The results for helmholtz in Figure 9a are materially different from the other applications we tested, in this case, the computation is completely dwarfed by communication overhead whenever offloading a region to the GPU. Even with a significant problem size, none of the GPU enabled versions could keep up with using only the CPU. As a result, the dynamic schedulers detected the issue, and effectively turned off the GPU by not sending it any work after the first few iterations, an average of 5 iterations to be precise. The number of outer-loop iterations assigned to CPU and GPU in the first six iterations of a run with the dynamic scheduler are depicted in Table 9c. 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 do believe the scheduler should converge on the decision to turn off the GPU faster but leave this for future work.

E. Overall

Depending on which application is in use, and in some cases the size of the dataset, the optimal scheduler changes. K-means benefits most from the quick scheduler, GEM the split, and CG the basic dynamic. Since each is seen to be worthwhile for at least one pattern, we believe that these four schedulers offer a good starting point for dynamic task splitting in heterogeneous systems. It is worth noting as well, that if a user is so inclined, and computes all the ratios for a given application, or saves the ratio computed by one of the dynamic schedulers, that can be fed into static for low-overhead and still accurate splits as depicted in Figure 12. As the results show, in a case with regions small enough to be highly sensitive to overhead, as in the case of CG, well tuned static scheduling can be highly effective. In the case of both GEM and k-means however, the dynamic policy performs better, implying that the ratio is adjusted during the run based on current conditions, producing 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 mention hardware platforms, will not always be feasible, so we foresee a continued need for runtime dynamic splitting moving forward.

In addition to the overall performance of each application, we collect the length of time spent blocking waiting on either accelerator. As discussed earlier in Section III, our dynamic scheduler, and those which depend on it, operate by assigning iterations such that they will produce the least possible blocking time. The total time spent blocking in each application scheduler combination is presented in Figure 13. Overall these follow our expectations, comparing the blocking time to the overall execution time behavior of the benchmarks matches well in most cases. There is a notable exception however. In the CG benchmark the static scheduler, while it does not perform well, performs significantly better than the split scheduler. When evaluated by blocking time however, the split scheduler would seem

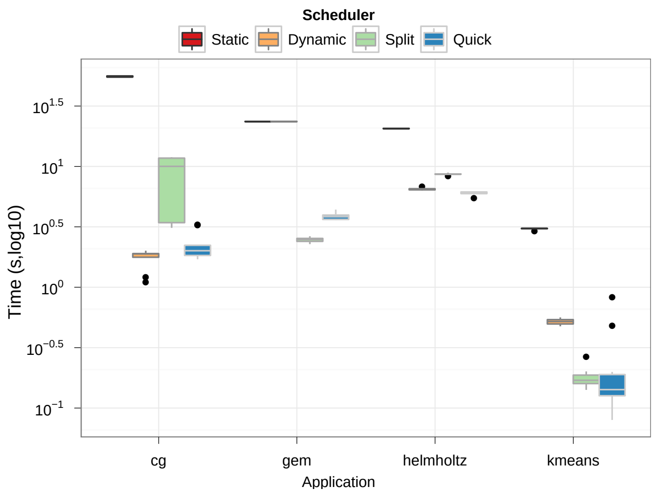


Figure 13: Total blocking time observed with each scheduler for all applications

to be the more efficient option. This suggests that alternative metrics for the dynamic schedule may produce better results for this application, but that is outside the scope of this paper. The reason as far as we can tell is due to the increase in the percentage of the accelerated region spent transferring data and in synchronization with respect to work. As the regions get smaller that overhead begins to dominate the execution time of the region, slowing computation despite the fact that the workload is well balanced. This is supported by the time it takes to compute an iteration on each device under the two schedulers. In the case of static, an iteration is calculated on the CPU every $0.39\mu s$ and on the GPU every $0.26\mu s$. The same case run with split however, with the default div of 10, runs a CPU iteration in approximately the same time at $0.37\mu s$ but GPU iterations in $1.12\mu s$. The extra overhead increasing the per iteration runtime by more than four times.

VI. Related Work

Automatic task scheduling has become increasingly popular for parallel computing recently. Traditionally the domain of shared memory parallel systems like OpenMP [7], Intel TBB [8] or even languages like Erlang, task scheduling has begun to move outside that realm with the rise of framworks for cluster and cloud task based computing. Frameworks like Charm++ [9] for clusters, and MapReduce [10] 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 [11] 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 [12] provides a heterogeneous grid of work stealing resources with a small number of high-uptime cluster-like resources for improved performance and reliability.

Accelerator-based heterogeneous computing, including IBM's CellBE and GPU plaforms 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 [13] is designed to be a platform for heterogeneous task scheduling. Along with StarPU, Qilin [14], Scout [15] and the work by Jiménez et al. [16] forms a solid foundation for both the desire and the capability for a heterogeneous task scheduler. All of 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 ease the translation of legacy scientific codes and make some of the advances available now.

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. [17] 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. [18] 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 may be remapped automatically to use the CPU as well, and as such at least to mitigate the issue.

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 2x over using the CPU or GPU alone. We clearly demonstrate that accelerated OpenMP should include a `hetero()` clause.

In future work we will implement our runtime and policies at a lower level, such as the compiler or potentially an extension. This implementation will not only make the system easily available to end users, but also allow us to reduce the memory transfer cost, and to provide a platform for heterogeneous task scheduling research. We further propose to 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. Mller, 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] 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.

[4] 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.

[5] R. Anandakrishnan, 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.

[6] 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.

[7] 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.

[8] J. Reinders, "Intel Threading Building Blocks," 2007.

[9] 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.

[10] J. Dean and S. Ghemawat, "MapReduce," *Communications of the ACM*, vol. 51, p. 107, Jan. 2008.

[11] 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.

[12] H. Lin, X. Ma, and W.-c. Feng, "Reliable MapReduce Computing on Opportunistic Resources," *Cluster Computing*, Feb. 2011.

[13] 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.

[14] 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.

[15] 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.

[16] 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. Seznec, J. Emer, M. OBoyle, M. Martonosi, and T. Ungerer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, vol. 5409, pp. 19–33.

[17] 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.

[18] 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.

Prepared by LLNL under Contract DE-AC52-07NA27344.