CoreTSAR: Task Scheduling for Accelerator-aware Runtimes

Thomas R. W. Scogland Wu-chun Feng
Department of Computer Science, Virginia Tech,
Blacksburg, VA 24060 USA
{tom.scogland,wfeng}@vt.edu

Barry Rountree Bronis R. de Supinski
Center for Applied Scientific Computing, Lawrence
Livermore National Laboratory,
Livermore, CA 94551 USA
{rountree,bronis}@llnl.gov

Abstract

Heterogeneous supercomputers that incorporate computational accelerators such as GPUs are increasingly popular due to their high peak performance, energy efficiency and comparatively low cost. Unfortunately, the programming models and frameworks designed to extract performance from all computational units still lack the flexibility of their CPU-only counterparts. Accelerated OpenMP improves this situation by supporting natural migration of OpenMP code from CPUs to a GPU. However, these implementations currently lose one of OpenMP's best features, its flexibility: typical OpenMP applications can run on any number of CPUs. GPU implementations do not transparently employ multiple GPUs on a node or a mix of GPUs and CPUs. To address these shortcomings, we present CoreTSAR, our runtime library for dynamically scheduling tasks across heterogeneous resources, and propose straightforward extensions that incorporate this functionality into Accelerated OpenMP. We show that our approach can provide nearly linear speedup to four GPUs over only using CPUs or one GPU while increasing the overall flexibility of Accelerated OpenMP.

1. Introduction

While heterogeneous, especially GPU-accelerated, large-scale systems are becoming more popular, their programming models deter many potential users. Unlike adding more or faster CPUs, which at least work without code changes, programs must be explicitly updated to use GPUs, frequently with unfamiliar programming models and syntax. Rather than grapple with the issue, users often simply run their legacy CPU-only code on accelerated resources, which can leave a significant portion of the computing resources idle. A more familiar and consistent programming model that handles both accelerators and CPUs efficiently would realize significantly more benefits of heterogeneous systems.

Accelerated OpenMP [6] offers the desired programming model with familiar OpenMP-style syntax. This syntax will facilitate the adoption of accelerators in scientific computing, especially for legacy OpenMP applications, which can be ported to the GPU with native syntax. However, Accelerated OpenMP is not a panacea: the model's current design helps one *move* their computation to a *single* GPU with straightforward adjustments to OpenMP source code. The extensions do not support transparent use of more than one GPU. They also do not support use of both CPU and GPU resources within the same parallel region. Thus, users must manually split work between available resources and manage the complex rules that ensure OpenMP and Accelerated OpenMP cooperate safely.

Our work creates a scheduling system, along with a set of proposed Accelerated OpenMP clauses, that allow users to exploit all computational resources through a consistent interface. With our approach, a user does not have to divide their problem manually across devices. Our system, the CoreTSAR (Task Scheduler for Accelerator-aware Runtimes) library, automates the scheduling, load balancing, and cross-device data management. This paper presents our implementation of CoreTSAR as a library for use with Accelerated OpenMP or any accelerator-aware runtime as well as a proposal to integrate its functionality into Accelerated OpenMP. CoreTSAR has been designed so the compiler/runtime could transparently include it. Specifically we make these contributions:

- Extensions to Accelerated OpenMP to support low-overhead stable co-scheduling of parallel loop regions across an arbitrary number of CPUs and GPUs without code replication;
- An implementation of our extensions on top of the OpenMP runtime which, thus, is applicable to any Accelerated OpenMP implementation (our evaluation uses PGI Accelerator);
- An evaluation which demonstrates that our multiple GPU extensions significantly improve performance over only using one GPU or using CPU cores and one GPU, and that the choice of scheduler is application-centric and, thus, portable.

The paper is composed as follows. Section 2 offers background on GPUs, GPU programming models, and the extensions made to OpenMP to support these architectures. Section 3 describes the design of CoreTSAR including our task management concept, scheduling mechanisms, and memory management. Details on our implementation follow in Section 4. We present results in Section 5.

2. Background

This section discusses GPU programming in general, related terminology, and Accelerated OpenMP.

2.1 GPU Programming Models

GPUs are the most commonly available computational accelerator. We use the terminology from NVIDIA's CUDA architecture to describe their design and components. An NVIDIA Tesla C2070 GPU contains 14 multiprocessors, each of which is a 32 wide SIMD processor. CUDA supports using these as Multiple Instruction, Multiple Data (MIMD) units by serializing conflicting instructions.

The memory model is the most important difference between programming a GPU and a set of CPUs. While GPUs are node-

```
#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];
}
```

```
#pragma acc region for              \
        copyin(in1[0:end],in2[0:end])\
        copyout(out[0:end])         \
        copy(pow[0:end])
     /* hetero(<cond>[,<scheduler>[,<ratio>\
           [,<div>][,<devices>]]]) \
        partial(out,pow)*/ /* proposed extension */
for (i=0; i<end; i++){
    out[i] = in1[i]*in2[i];
    pow[i] = pow[i]*pow[i];
}
```

Figure 1: OpenMP (top) and Accelerated OpenMP (bottom).

local resources, they use a separate memory hierarchy that consists of a small scratch memory local to each multiprocessor called *shared memory*, L1 and L2 caches, and *global memory*, which is memory that all multiprocessors can access. Unlike system main memory, cache coherence of global memory is not maintained except for atomic instructions. This split between CPU and GPU accessible resources complicates the inclusion of GPUs in traditional shared memory programming models such as OpenMP.

2.2 OpenMP Accelerator Directives

Current options for OpenMP-like accelerator directives include HMPP [12], OpenACC [7], OpenMP for accelerators [6], and PGI Accelerator directives [27]. Our work refers to an OpenMP Language Committee proposal that combines and extends these options as Accelerated OpenMP. OpenACC is directly based on that proposal, and our statements are equally applicable to it. In this paper, we use the PGI Accelerator directives implementation for our examples and evaluation.

Unlike standard OpenMP, accelerator directives must support distributed memory. Figure 1 presents a simple example that illustrates the necessary modifications to an OpenMP program. The OpenMP version parallelizes the loop so that all threads share the input and output arrays. The accelerator version is similar, but more specific about memory movement. First, instead of simply listing the arrays as shared, each array is marked as `copyin()`, `copyout()` or `copy()`, which causes the runtime to allocate memory on a GPU and copy the CPU array into the device but not out, out but not in, or both in and out, respectively. The subscripts in these clauses support array shaping using the form `array[<start>:<end>:<stride>]`, which defines the array's size (which the compiler may not be able to infer), and can request that only a subset of the array be copied into or out of a region.

Figure 1 includes a comment that shows our proposed interface for invoking CoreTSAR. The `hetero()` clause specifies that the region should use CoreTSAR for coscheduling. We discuss its arguments in Section 3. We also propose another clause, `partial()`, to specify that only parts of certain arrays need to be copied to execute a given loop iteration, which allows the compiler and runtime to copy only the data that the scheduled work needs.

3. Design

First and foremost, CoreTSAR is a (co)scheduling entity that takes in a list of related tasks and distributes them across a set of (heterogeneous) resources. CoreTSAR is not a programming model, nor does it parallelize serial code. It *does* extend an existing programming model. One could use CoreTSAR with raw CUDA and C, or any combination of programming models, and in principle for any combination of devices. Our case study uses Accelerated OpenMP, specifically the PGI Accelerator implementation, to compile code for GPUs and OpenMP to generate threaded CPU code. Our extensions to Accelerated OpenMP would allow CoreTSAR to become a completely transparent part of the model. In order to schedule across devices that may, or may not, share memory, we also include a memory management interface that allows CoreTSAR to optimize data transfers. Specifically, it reduces transfer times by allowing data to persist across regions and copying only those elements currently necessary to complete the work assigned to the target device.

3.1 Assigning Tasks

Evenly dividing homogeneous iterations across homogeneous resources, as with the OpenMP static schedule, yields high performance. Since iterations in programs can vary in runtime, OpenMP supports additional types of scheduling (dynamic and guided) to improve load balancing. These schedules for heterogeneous iterations still target homogeneous resources so they work well on homogeneous, shared-memory resources with low concurrency control costs. However, they are less appropriate for heterogeneous resources due to varying costs and synchronization requirements. Since CoreTSAR targets heterogeneous resources with distributed memories, we provide different schedules.

Our adaptive scheduler assigns iterations at the boundaries of parallel regions, or sub-regions, and re-evaluates at their ends. This approach reduces locking overhead but does not balance load dynamically like OpenMP dynamic or guided. To provide balanced schedules, CoreTSAR predicts the time to compute an iteration on each resource in the next pass based on previous executions.

We base our schedules around assigning each device enough work to finish in the same amount of time as all others. CoreTSAR tracks the average time to complete an iteration on any given device, which it uses to predict the amount of work that each device can complete in the next pass. For example, assuming we have a system with two CPU cores and one GPU, with one CPU core controlling the GPU, if the CPU core completes 10 iterations in the same time that the GPU takes to copy in data, to complete 40 iterations, and to copy back the results, then the CPU should be assigned 20% of the operations in the following pass. Alternatively, if those numbers are reversed, the CPU would be assigned 80%. We thus determine the relationship between compute units and can compute the amount of work to provide each device to balance their loads. However, we must extend this simple approach to more than two devices and to choose an initial split.

3.2 Applying Ratios

We use a linear program to extend our approach to arbitrary device counts. The linear program computes the iterations to assign to each device based on their time per iteration. Figure 2 lists its variables (Equation 1), objective function (Equation 2) and accompanying constraints (Equations 3-6). The program minimizes the difference between the predicted runtime for each device, such that all should finish at the same time. We assume that performance of an average iteration does not change across instances of the region. The time for a device to finish its work in the next pass equals the time per iteration from the previous pass multiplied by the number of iterations that it is assigned. In practice this assumption holds well: although iterations may have different computational cost, the same iteration across two passes often has similar performance, rendering accuracy within a few percent for our test applications.

$$
\begin{aligned}
I &= \text{total iterations available} \\
i_j &= \text{iterations for compute unit j} \\
f_j &= \text{fraction of iterations for compute unit j} \\
p_j &= \text{recent time/iteration for compute unit j} \\
n &= \text{number of compute devices} \\
t_j^+ \text{(or } t_j^-) &= \text{time over (or under) equal}
\end{aligned} \tag{1}
$$

$$min(\sum_{j=1}^{n-1} t_1^+ + t_1^- \cdots + t_{n-1}^+ + t_{n-1}^-) \tag{2}$$

$$\sum_{j=0}^{n} i_j = I \tag{3}$$

$$i_2 * p_2 - i_1 * p_1 = t_1^+ - t_1^- \tag{4}$$

$$i_3 * p_3 - i_1 * p_1 = t_2^+ - t_2^- \tag{5}$$

$$\vdots$$

$$i_n * p_n - i_1 * p_1 = t_{n-1}^+ - t_{n-1}^- \tag{6}$$

Figure 2: Linear program variables, objective and constraints

3.3 Static Scheduling

Our static schedule uses the linear program to assign iterations based on either an input value or our computed default division. To increase portability, we compute a default division at runtime rather than using a precomputed static value. We assume that one instruction cycle on any GPU takes the same time as four on a CPU, which models a CPU with a SIMD unit and higher clock frequency. While this assumption does not hold in general, we can portably compute an initial time per iteration for each device. We compute the time per iteration for a GPU as $t_g i = 1/c_g/4$ and for CPU cores as $1 - t_g i$ (where c_g is the number of multiprocessors on a GPU, in the case of multiple distinct GPUs, the largest value is used for the CPU cores), which scales based on the compute resources available. For applications that are not dominated by floating-point computation, we have considered models that include several other factors, including memory bandwidth and integer performance. However, these models have proven less successful; the selection of their appropriate static model is future work.

3.4 Adaptive Scheduling

Our adaptive schedules (*Adaptive*, *Split* and *Quick*) use the static schedule as an initial training step. From the time that each device takes to complete its iterations in this static pass, we use our linear program to compute the appropriate division of work for subsequent passes. Our design intentionally includes all recurring data transfer and similar overheads required to execute an iteration on a particular device. Thus, we incorporate those overheads into the cost of the iteration and naturally account for them. The *Adaptive* schedule trains on the first instance of the region and then each subsequent instance. The *Split* schedule accommodates regions that may only run once or that may benefit from scheduling more often. It breaks each region instance into several evenly split sub-regions, based on the `div` input. Each time a sub-region completes, we use the linear program to split the next. This schedule can provide better load balance at the cost of increased scheduling and kernel launch overhead. Thus, it is impractical for short regions and overhead sensitive applications. The *Quick* schedule balances between the *Split* and *Adaptive* schedules by executing a small sub-region for its first training phase, similarly to *Split*. It then immediately schedules all remaining iterations of the first region instance and uses the *Adaptive* schedule for any subsequent instances. This schedule suits applications that cannot tolerate a full instance using the static schedule or the overhead of extra scheduling steps in every pass.

3.5 Memory Management

A key to efficient region execution across multiple memory spaces is accurately determining the minimum amount of data movement necessary to complete the computation. CoreTSAR is designed to reduce the data movement requirements of applications over vanilla Accelerated OpenMP in two key ways. First by the ability to specify that a data block should be persistently allocated on a device and remain resident until it is explicitly removed. The Accelerated OpenMP directives currently do not offer this, instead any given block of data is associated with a region, explicitly or implicitly, and in the case of PGI Accelerator and OpenACC can only be used on more than one GPU by having multiple threads running at the point where that region begins. As a result of these limitations, a user may be forced to repeatedly copy data, which is known to be static, to avoid major code restructuring. The addition of persistent copies removes this potential obstacle. Second, we allow the user to specify the data that each task requires individually as a part of the input and output data blocks. We have provided a simple memory management interface that takes the start of an array or matrix, the size of the element that each task will use, and the number of those elements each task should receive. Combining that with the iteration variable produces the necessary data for a given task. This approach supports simple determination of the regions necessary for runs of successive tasks. While this simple pattern clearly will not work for all applications, we have found that many applications can use this interface without modification. Either it naturally associates tasks to data, which is common since it is best for cache-locality, or tasks require the entire data region, as with random or unpredictable accesses. Future work will develop a more comprehensive specification methodology.

4. Implementation

We implement CoreTSAR as a library on top of Accelerated OpenMP. We have tested it with PGI Accelerator as well as Cray's Accelerated OpenMP prototype. Our evaluation in this paper focuses on PGI Accelerator, so we use its directive format in our examples. This section discusses our implementation including its portability, its API and our automatic memory manager as well as some necessary deviations from the abstract design discussed in Section 3.

4.1 Shifts from Design

Several issues lead to pragmatic shifts from our high-level design. The underlying Accelerated OpenMP implementations assume all threads of a team participate in barriers. Some applications do not suit GPU computing while others have fine-grained parallel regions. We now discuss how we accommodate these issues.

4.1.1 Scheduler Overhead

Since our schedules repeatedly solve a linear program, that overhead could be an issue, especially for applications with many short regions. We use the lp_solve [5] library, a highly optimized linear program solver that can use the previous solution's tableau as a partial result. This incremental approach greatly reduces overhead.

Figure 3 represents the time spent in CoreTSAR over 1,900 passes and 19,000 scheduling iterations with the split scheduler. Unfortunately, we find that our original linear model has exponential time complexity as the number of devices increases. In the worst case the split schedule with four GPUs takes nearly 3×

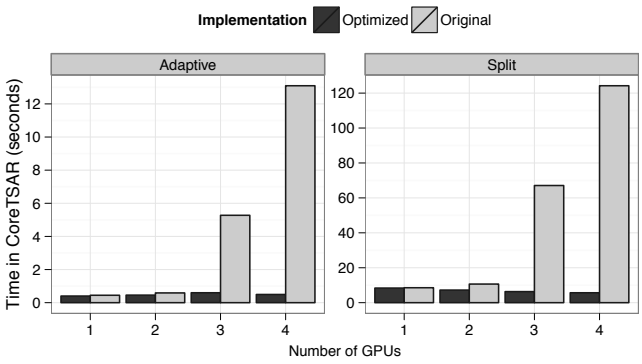


Figure 3: Time spent in CoreTSAR during 1,900 passes.

$$min(\sum_{j=1}^{n-1} t_1^+ + t_1^- \cdots + t_{n-1}^+ + t_{n-1}^-) \tag{7}$$

$$\sum_{j=0}^{n} f_j = 1 \tag{8}$$

$$f_2 * p_2 - f_1 * p_1 = t_1^+ - t_1^- \tag{9}$$

$$f_3 * p_3 - f_1 * p_1 = t_2^+ - t_2^- \tag{10}$$

$$\vdots$$

$$f_n * p_n - f_1 * p_1 = t_{n-1}^+ - t_{n-1}^- \tag{11}$$

Figure 4: Modified objective and constraints

longer than the 40-second compute phase (over 90% in the lp_solve routine).

Two issues reduce the solver's performance. First, its input has widely distributed values, which leads to poor numerical stability that slows convergence and frequent floating-point error corrections. Second, about half of the matrix values are integers, which significantly increases computational complexity.

To alleviate these issues, we remove integer output requirements and keep values near the range of 0 to 1. Figure 4 shows the new objective function (Equation 7) and constraints (Equations 8-11). We now compute the floating-point fraction of total iterations f_j all of which must sum to 1. These changes produce the optimized results that Figure 3 shows. Thus, the time in CoreTSAR can actually decrease as the number of GPUs increases because of GPU performance consistency. Despite the larger matrix, the solution converges faster since it deviates less from the previous solution.

The runtime multiplies each f_j by I and assigns the nearest lower integer. This approximation can fail to assign up to n iterations optimally. We assign these iterations round-robin to devices so each device is at most 1 iteration above or below optimal. Since most applications execute thousands of iterations, the small deviation is within the error threshold of our measurements.

4.1.2 GPU-Averse Applications

CoreTSAR handles applications that run more efficiently using one CPU core than using that core to control a GPU. Some applications cannot run a GPU pass quickly enough to match the CPU execution time regardless of the number of iterations. We can identify when a given device cannot match the performance of other devices and take action since we track the time per iteration of each device.

```
ctsar * ctsar_init(int size, ctsar_type sched,
                   ctsar_dev_type allowed_devs,
                   double *rat,int *div);
ctsar * ctsar_next(ctsar * c, int size);
int     ctsar_loop(ctsar * c);
void  * ctsar_reg_mem(ctsar * c, void * ptr,
                  size_t item_size, size_t count
                  ctsar_mem flags);
void ctsar_unreg_mem(ctsar * c, void * ptr);
void ctsar_swap_mem(ctsar * c, void * ptr, void * ptr);

typedef struct ctsar{
  ctsar_type   type;     //scheduler type in use
  ctsar_device devices[];//devices by thread id
  void * internal;       //opaque internal values
}ctsar;

typedef struct ctsar_device{
  ctsar_dev_type type;   //type of device
  size_t start;          //start of device iterations
  size_t end;            //end of device iterations
  void * internal;       //opaque internal values
}ctsar_device;
```

Figure 5: CoreTSAR API

Thus, CoreTSAR converts a GPU offload thread into a CPU thread when the GPU has a higher time per iteration than the slowest CPU core for a configurable number (default is two) of iterations. We discuss the effects of this extension further in Section 5.

4.2 CoreTSAR API and Usage

The CoreTSAR API consists of six functions and two structures (and associated values), as Figure 5 shows. Figure 6 presents an example transformation for a real-world example. We use the manual code in our k-means benchmark evaluation (variable names shortened for space). A comment in the original version shows our proposed clause to invoke CoreTSAR's functionality through directives.

In the manually translated version, after initialization with the adaptive scheduler, the two registration functions register a persistent input and an output array that only needs to be partially copied back. In subsequent iterations the associated device array is retrieved by matching the input pointer's address and returned without reallocation, CPUs simply receive the input pointer back immediately. The bottom of the example uses a swap to inform CoreTSAR that the host buffer to use for copies has changed, which allows host-side double buffering without re-registration.

We encapsulate the main kernel in a parallel region that uses all cores. Using OpenMP thread numbers as IDs, we obtain the device structures from CoreTSAR. The inner loop of the parallel region supports use of the split and quick schedules. The `ctsar_loop()` function tracks the status of the work and continues or exits as needed. Inside the loop, the `ctsar_next()` function takes the number of iterations that have not yet been completed and distributes iterations to devices.

Note that there is no replication of code, each thread is assigned iterations to compute in their associated device structure and select whether they are using the GPU for acceleration or not using the Accelerated OpenMP `if()` clause. In the event that the device is a CPU, the loop is run serially on the associated core completing its assigned iterations. If, on the other hand, it is a GPU-controlling thread, the `deviceptr()` clause passes in memory regions provided by CoreTSAR rather than specifying explicit array slices in the directive. In this code, the library handles timing and all memory allocation and movement, except that of fc, which greatly simplifies the design compared to our previous work. Even with the semantic improvements, the CoreTSAR version is more verbose

```
ctsar * c = ctsar_init(no, CTSAR_ADAPTIVE,
            CTSAR_DEV_CPU | CTSAR_DEV_GPU,
            NULL, NULL);
#pragma omp parallel shared(fo,fc) private(i,j,k)\
                    firstprivate(no,ncl,nco)
{
  int tid = omp_get_thread_num();
  ctsar_device * dev = c->devices[tid];
  float * cfo,cm;
  cfo = (float *)ctsar_reg_mem(c, fo, sizeof(int),
        no*nco, CTSAR_MEM_PERSIST | CTSAR_MEM_INPUT);
  cm  = (float *)ctsar_reg_mem(c,  m, sizeof(int),
            no, CTSAR_MEM_PARTIAL | CTSAR_MEM_OUTPUT);
  do{
      c = ctsar_next(no);
      int gts = dev->start, gte = dev->end;
#pragma acc region for independent private(i)\
          deviceptr(cm,cfo) copyin(fc[0:ncl*nco])\
          copyin(nco,no,ncl,gts,gte) \
          if(dev->type == CTSAR_DEV_GPU)
      for (i=gts; i<gte; i++) {
          cm[i] = findc(no,ncl, nco, cfo, fc,i);
      }
  }while(ctsar_loop());
  ctsar_swap_mem(c,m,m2);
}
swap(&m,&m2);
```

```
#pragma acc region for independent private(i) copyout(m)\
            copyin(fc[0:ncl*nco],fo[0:no*nco])          \
            copyin(nco,no,ncl,gts,gte)\
            hetero(1,quick) \
            persist(fo) \
            partial(m) /* proposed extension */
for (i=0; i<no; i++) {
  m[i] = findc(no,ncl,nco,fo,fc,i);
}
swap(&m,&m2);
```

Figure 6: Manually transformed k-means kernel (top) and proposed extensions for automatic transformation of the k-means kernel (bottom).

than the original, but it can use all CPU cores and all GPUs without code replication.

4.3 Memory Management

Our previous heterogeneous scheduler design, handled memory by sending the entire inputs to the GPU at the beginning of a region, and copying the entire output arrays back at the end. Some benchmarks use Accelerated OpenMP data regions to manage persistent memory at a higher level that allows the input copy phase to be skipped when using one GPU. Nonetheless, far more data than necessary is copied. It used that design because existing Accelerated OpenMP implementations did not support partial array copies. With this support now available, we use it to reduce memory transfer overhead and to eliminate the need to merge multiple copies of output arrays in a separate step. Further, we found that we could not emulate a data region for *persistent* data to be placed on multiple GPUs. The model supports doing so with one GPU but the required nesting order with multiple GPUs is complicated and frequently infeasible.

Our solution uses the `deviceptr()` clause to pass pointers to memory that are allocated on the GPU and managed by CoreTSAR. We offer a straightforward syntax that supports automatic management of partial array copies and data regions across devices. CoreTSAR currently takes a pointer to CPU memory, the size of an element of that array, the number of array elements, and a flag option that allows the user to control copy behavior. Flags control whether memory is copied in or out or both. CoreTSAR also supports two special cases: persistent memory and partial copies. The manually translated code in Figure 6 shows examples. While we control memory copies instead of the compiler, and thus lose any compiler optimizations, the resulting applications provide equal or better performance than those produced by allowing the PGI Accelerator compiler to handle copies, and the added flexibility of offering persistent memory regions across multiple GPUs improves on that even more by avoiding unnecessary copies of static data.

Regardless of the flags, `ctsar_reg_mem()` allocates an appropriate size buffer on the device associated with the calling thread. If the region is set to persistent, we copy the data from the CPU array into the newly allocated memory, where it resides until it is explicitly removed with a call to `ctsar_unreg_mem()`. Given a copy direction (in or out) but no special flags, we copy the region to the GPU in `ctsar_next()` and from it in `ctsar_loop()` when the return value will cause the thread to exit the outer loop. Regions that are marked partial indicate that the region only needs part of that memory to execute the compute kernel.

Currently, CoreTSAR only handles simple partial copies, as described in Section 3.5, in which it copies the elements of the array that correspond to the iterations assigned to each thread, to or from its memory space. This basic support only covers a small subset of the total possible cases but covers most cases that arise in practice, partly because OpenMP codes tend to work on sequentially allocated memory locations in a given thread. Given an item size, we can use this system either to copy a range of items, such as floats, that directly correspond to loop iterations, or to copy a run of 100 floats that correspond to each loop iteration. For example, the Helmholtz benchmark uses a two dimensional matrix, allocated as an array, in which each thread computes a set of rows in that matrix. While each thread computes more than one output cell, since they are sequential, we can implement the necessary partial copies by specifying the item size as the size of a single element times the length of a row. Thus, we do not require special handling beyond what CoreTSAR offers.

The one special-purpose treatment worth mentioning is that of reductions. Since the output of a reduction is not specific to a single thread, we keep one version in each memory space and merge the results at the end in a short reduction pass on the CPU cores.

4.4 Portability

Currently CoreTSAR assumes that OpenMP manages CPU threads, and that accelerators are CUDA-enabled GPUs. Thus, we use `omp_get_thread_limit()` to determine the number of CPU resources, `cudaMalloc()/cudaMemcopy()` to allocate and to manage accelerator memory and `cudaGetDeviceProperties()` to read the number of cores and capabilities of the GPU for initial performance estimation. Our design does not require these assumptions so we could easily add other targets or replace these interfaces.

5. Evaluation

This section evaluates the CoreTSAR library. We compiled all benchmarks with the PGI Accelerator compiler, pgcc, pgCC or pfortran as appropriate, version 12.5. Optimization flags are -O3, -mp=allcores, and enabling compilation for NVIDIA GPUs including compute capabilities 1.3 and 2.0. Table 1 lists our test platforms. Unless otherwise specified, we ran tests on escaflowne. All tests employ all CPU cores, in tests with GPUs enabled, one thread is used to control each selected GPU, and thus does not do computation. We use default scheduler parameters unless otherwise specified, with the initial split calculated at runtime based on the available resources and a div of 10. We include all scheduling overhead, GPU data transfer time, and synchronization time.

System name	CPU Model	CPU Cores/die	CPU Dies	CPU RAM (MB)	GPU Model	GPU Cards	GPU Cores	GPU RAM (MB)
amdlow3	E3300	2	1	2,012	Tesla C2050	1	448	3,071
armor1	E5405	4	2	3,964	GeForce GT 520	1	48	1,023
dna2	i5-2400	4	1	7,923	GeForce GTX 280	1	240	1,023
dna3	i5-2400	4	1	7,923	Tesla C2075	1	448	5,375
escaflowne	X5550	4	2	24,154	Tesla C2070	4	448	5,375

Table 1: Test system specifications, all CPUs and GPUs are made by Intel and NVIDIA respectively.

We evaluate CoreTSAR with five applications that we manually accelerated using the PGI Accelerator directives, and subsequently enhanced with the CoreTSAR library. These applications are GEM [1, 15, 16], k-means, CG [4], Helmholtz and CORR [17]. We chose these applications because they provide a range of application behaviors in terms of data sizes, region lengths, and GPU suitability as well as hitting many of the potential corner cases and difficult situations CoreTSAR would likely face in real-world use. As our primary concern is scheduling behavior and not computational kernel performance, we ported each from OpenMP (except CORR, which used HMPP originally) to PGI Accelerator with the minimum change possible. This decision provides a realistic use of Accelerated OpenMP, at least at first, to evaluate the suitability of GPUs, and is how we envision CoreTSAR being used if integrated directly into the directives. Thus, we obtain computational kernels that are not optimized for the GPU other than by the compiler. However, CoreTSAR supports users providing an OpenMP CPU code and a manually implemented and highly optimized CUDA kernel, as long as the inputs and outputs of the two are compatible. CoreTSAR would schedule across them and offer memory management with the same syntax.

During a given run, we collect various statistics and results. We define computation time as the time to complete all computations and all initialization and finalization necessary to run the computations. We include all time that is not required for the original OpenMP CPU code to function, such as library initialization, scheduling, and memory movement. We do not include application IO or problem setup that is shared between CPU, GPU and scheduled versions. We also record the time for each thread to complete its assigned iterations, from which we can compute the time that threads wait for others to complete, the time spent to calculate the split for the next pass and, as a subset of that, the time taken to update the linear model solution to the new values. Finally, we track the time per iteration for each thread, as described in Section 3.

5.1 Benchmarks

As mentioned above, we employ five benchmarks in our evaluation. CG is a direct port of the NAS parallel benchmarks, their implementation of the conjugate gradient method. CORR is a correlation code that computes an upper triangular 2D correlation matrix originally from the PolyBenchGPU benchmark suite. Each independent iteration is one row of that output matrix so each iteration has a different load, which produces an unbalanced execution profile. GEM is a molecular modeling application for the study of the electrostatic potential along the surface of a macromolecule whose profile has been well studied as a test case for GPU optimization [9]. Helmholtz is a discrete finite difference code that uses the Jacobi iterative method to solve the Helmholtz equation. Finally, k-means is a popular iterative clustering method. The five benchmarks can generally be characterized by the number of passes through the parallel region that they make, the length of each of these passes, and how suitable they are to run on the GPU. Table 2 presents values that represent each of these properties for our five benchmarks.

The table exhibits a wide range in number of passes through the parallel region – from 1 to 1900 passes. Our scheduler is designed to operate primarily at the boundaries of parallel regions, so this number can greatly affect how CoreTSAR interacts with an application. For example, in the GEM benchmark, the adaptive scheduler acts in a manner identical to the static scheduler because the training pass is the only pass in the application. Conversely, CG has many passes, which provides CoreTSAR with a wealth of opportunities to adjust and to correct scheduling decisions. That benefit comes at a cost. CG's passes are very short, which accentuates any scheduler overheads as well as data copy costs. CORR is also important in this respect; it has only one pass by default, but is a computational kernel that a real application would use repeatedly. For our evaluation, we use 10 iterations; 5 and 20 iterations yield similar relative results.

Benchmark	Passes	Time/ pass	CPU (static) runtime	GPU runtime	Ratio
CG	1900	0.02	50.56	273.04	0.85
CORR	10*	6.36	1540.43	70.97	0.45
GEM	1	1098.10	1098.35	107.43	0.06
Helmholtz	100	0.08	8.61	73.64	1.00
k-means	7	1.14	8.82	4.79	0.41

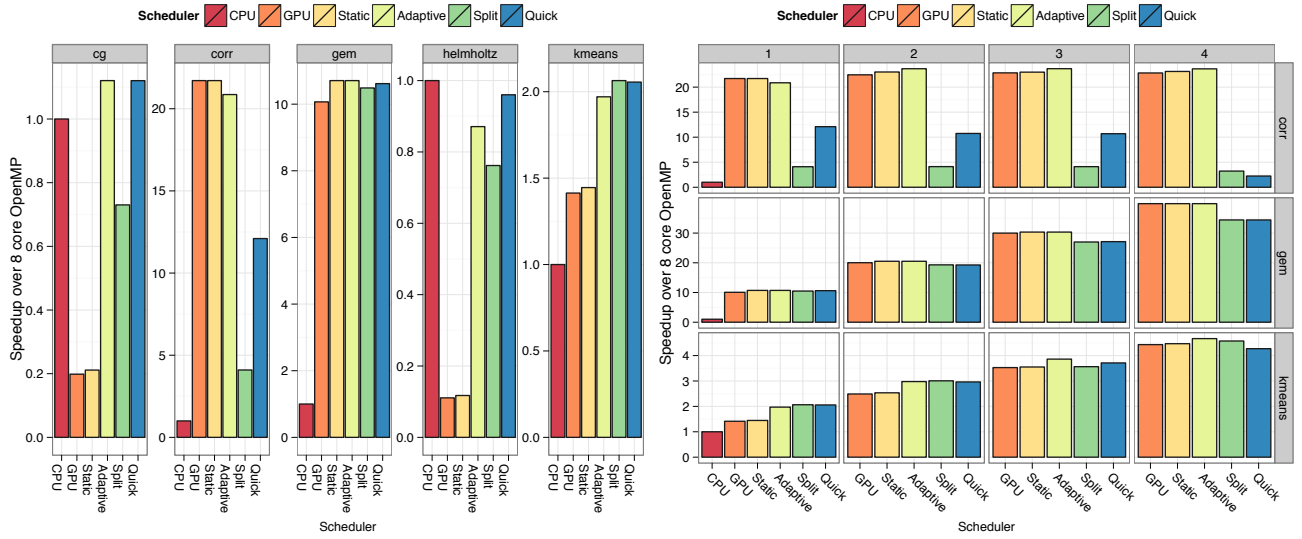
Table 2: Benchmark characteristics, times in seconds.

The table also shows a wide range of GPU suitability, which we define as the quality of an algorithm or in this case an implementation for running on a particular device. Running GEM on only one GPU finishes the problem more than $10\times$ faster than on eight server class Intel CPU cores. CORR also shows extreme suitability, largely a result of the static schedule employed in the CPU tests. Because the workload is imbalanced, each CPU core is given a different amount of actual work. The GPU test, because of the multiblock design of GPU kernels, handles this variation better. If we use the OpenMP dynamic schedule, CORR runs in approximately 150 seconds, $10\times$ faster than the static performance. The ratios of GEM and CORR reflect their high level of suitability, a ratio near zero allocates most work to the GPU, near 1 almost all on the CPU, as discussed in Section 3. K-means has a similarly high level of suitability, while Helmholtz is so unsuitable that it benefits from turning off the GPUs. Generally, the suitabilities match our expectations, with the exception of CG.

In our previous work, as well as that of others, CG has been found suitable for GPUs. Some of our experiments on other platforms showed a ratio of approximately 0.55. Here, the GPU version takes more than $5\times$ longer than the CPU version on our primary test system, partly because of the overhead of running so many small kernels, but mostly because PGI Accelerator's default assignment policy underutilizes the GPU multiprocessors. Optimization could likely overcome this issue, but we leave this for future work since we focus on scheduling properties rather than individual benchmark GPU performance.

5.2 CoreTSAR Performance

We begin our evaluation of CoreTSAR with an evaluation of the overall speedup achieved for benchmarks across schedulers on one GPU and all cores on escaflowne, as Figure 7a shows. For the applications that suit GPUs, all but Helmholtz, one or more of



(a) CoreTSAR speedup using one GPU across schedulers

(b) CoreTSAR speedup across schedulers and number of GPUs for GPU amenable benchmarks

Figure 7: CoreTSAR evaluation on escaflowne.

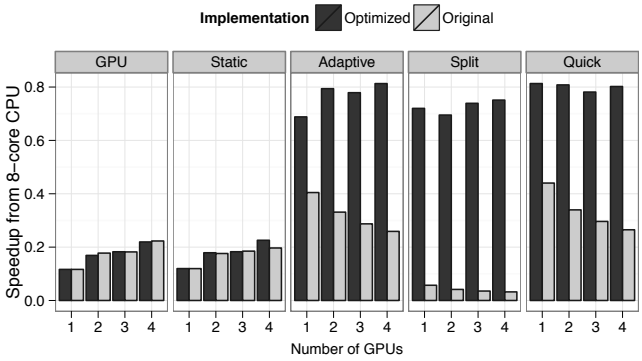


Figure 8: Helmholtz with and without GPU backoff support

the CoreTSAR modes, Static, Adaptive, Split and Quick, improve performance over using either the GPU or CPU alone. Performance of k-means improves by as much as 2×, GEM and CG by about 10%, and in the case of CORR by a tiny margin over GPU only.

5.2.1 GPU Averse Applications

As we discussed earlier, Helmholtz was selected specifically because the application is GPU-averse. In no circumstance that we have found does it perform better by using a GPU for any work at all. We include this benchmark because real applications are likely to include sections like this, and should not suffer much performance degradation as a result of using CoreTSAR. More importantly, some applications may perform well using a GPU in a particular configuration but not another, and users need consistent performance running on either hardware without being forced to rewrite their code. We discuss this topic further in Section 5.4.

For the single GPU case, one of the lower overhead schedulers, such as Adaptive or Quick, can reduce the performance loss of Helmholtz by quickly converting GPU threads to CPU threads. To evaluate the effect of our extension that converts GPU threads, Figure 8 displays the results for Helmholtz without GPU back-off in *Original* and with it in *Optimized*. While neither result matches the CPU performance, *Optimized* is as much as 24× faster than *Original*, and, having released the GPUs, uses fewer resources. Regardless of the number of GPUs, CoreTSAR decides to stop using all of them. While we do not achieve a speedup, we expect that some accelerators, especially upcoming ones that share the memory subsystem with the CPU, may be able to accelerate this application. In the meantime we can mitigate the loss not only for this application but for any application that runs poorly on a given accelerator. For example, CG benefits from it whenever we assign more than one GPU, by turning off all but one GPU to avoid detrimental data-movement overheads.

5.2.2 GPU Amenable Applications

The other three benchmarks (CORR, GEM and k-means) benefit from using more than one GPU, as Figure 7b shows. CORR does not behave like the others. Moving from one to two GPUs increases performance, but performance improves only slightly with more GPUs. The Quick scheduler even loses performance as we add more GPUs, due to an incorrect decision made based on the imbalanced load presented by CORR in the training pass. This result shows that Quick and Split are not appropriate for applications with highly variable in-pass workloads. Adaptive performance does not improve because the benchmark only has enough work to utilize one GPU fully and part of a second; adding more GPUs leaves them underutilized and does not further reduce computation time.

GEM and k-means on the other hand scale almost perfectly up to four GPUs and four CPU cores. Specifically, the runtime for the default dataset falls from 101 seconds for GEM or 3.29 seconds for k-means with one GPU and the Quick scheduler to 27 seconds and 1.4 seconds with four GPUs using Adaptive, which implies scaling values of 3.74× and 2.35× respectively. While the scaling of k-means may not appear linear, recall that we reduce the CPU resources as we add GPU resources. Thus, 4× is an unreasonable expectation. 8 CPU cores achieve as much performance as 80% of

Num. of GPUs Benchmark	1	2	3	4
CG persist	257.48	264.94	268.52	268.59
CG	438.58	664.81	892.59	1109.84
CORR persist	106.39	67.05	66.98	66.58
CORR	113.22	66.89	66.73	66.59
GEM persist	101.01	52.72	35.65	27.10
GEM	101.01	52.72	35.65	27.10
Helmholtz p.	73.22	48.53	50.45	48.75
Helmholtz	86.51	64.25	70.81	75.15
k-means p.	4.61	2.63	1.88	1.49
k-means	4.76	2.80	2.08	1.77

Table 3: Time to run each benchmark with the static schedule with and without persistent memory available

one GPU, so we are effectively scaling from 1.8 GPUs to approximately 4.4 GPUs, which corresponds to peak scaling of $2.4\times$, only very slightly more than achieved. With k-means, the most efficient scheduler changes as the number of GPUs increases. Including the GPU copy time in the iteration time causes the Quick scheduler to underestimate the performance of the GPUs in the initial short training phase, then underutilize them during the rest of the first pass. As the GPU count increases, the effect intensifies and the Adaptive scheduler performs progressively better despite the fact that they both make equivalent decisions in subsequent passes.

5.3 Memory Performance

We now evaluate our memory manager and some of its optimizations (Section 4.3). The partial array copy and persistent array options are of particular interest. We can not disable partial memory copy and maintain correctness in our current design, so we do not evaluate it directly. We can manipulate persistent memory across multiple GPUs. Without CoreTSAR, placing memory on a GPU, or especially multiple GPUs, and retaining that memory from the first entry into a region through to the last is prohibitively difficult. Each of our applications has at least a small data region that does not change, either a range of constants in the case of CORR, or the large list of observations in k-means. Enabling persistence on these data structures can greatly reduce the memory overhead in some cases as Table 3 shows.

We use the Static scheduler results for the purpose of this comparison because the Adaptive scheduler would in some cases detect the excess overhead and cloud the results by converting GPU threads to CPU threads. The most immediately striking difference is in CG, where the overhead caused by a lack of persistent memory exceeds $3\times$. CORR, due to low data copy overhead, and GEM, because data is only copied once, are largely unaffected. Helmholtz has an interesting pattern: the copy overhead decreases with more GPUs, because its copy overhead is mostly from the partial data that must be copied every iteration, which is increasingly parallelized with more GPUs. K-means initially shows little difference, but reaches approximately 15% overhead with four GPUs due to a lack of persistent memory. Despite its simplicity, existing Accelerated OpenMP do not include a comparable persistent memory option.

5.4 Adaptation Across Machines

As we mentioned in earlier sections, one of our primary goals is to create a system that offers the flexibility of OpenMP, automatically adapting to available resources portably, for accelerator-based codes and systems. In Section 5.2 we showed that CoreTSAR adapts well to different numbers of GPUs in a given system. We now evaluate how CoreTSAR performs across a set of highly distinct systems. We use a group of diverse systems that we boot with the same system image to reduce software differences, and test our benchmarks across them all. Table 1 shows the systems. Other than our main test system evaluated elsewhere (escaflowne), these systems have only one GPU. However, the GPUs are from several generations with varying amounts of memory (from an NVIDIA GT 520 at the low end up to an NVIDIA C2075). The CPUs are even more diverse, ranging from a single 2-core Intel Celeron to two quad-core Intel Xeon processors. Of particular interest are the GPU-centric system amdlow3, which contains a dual-core Intel Celeron processor and NVIDIA C2050 GPU, and the CPU-centric system armor1 with two quad-core Intel Xeon cores and a low power NVIDIA GT 520. For this evaluation, we use all benchmarks except Helmholtz and CORR, as the two smaller memory GPUs cannot hold our default problem sets for these benchmarks in memory. Figure 9a shows our results across these systems.

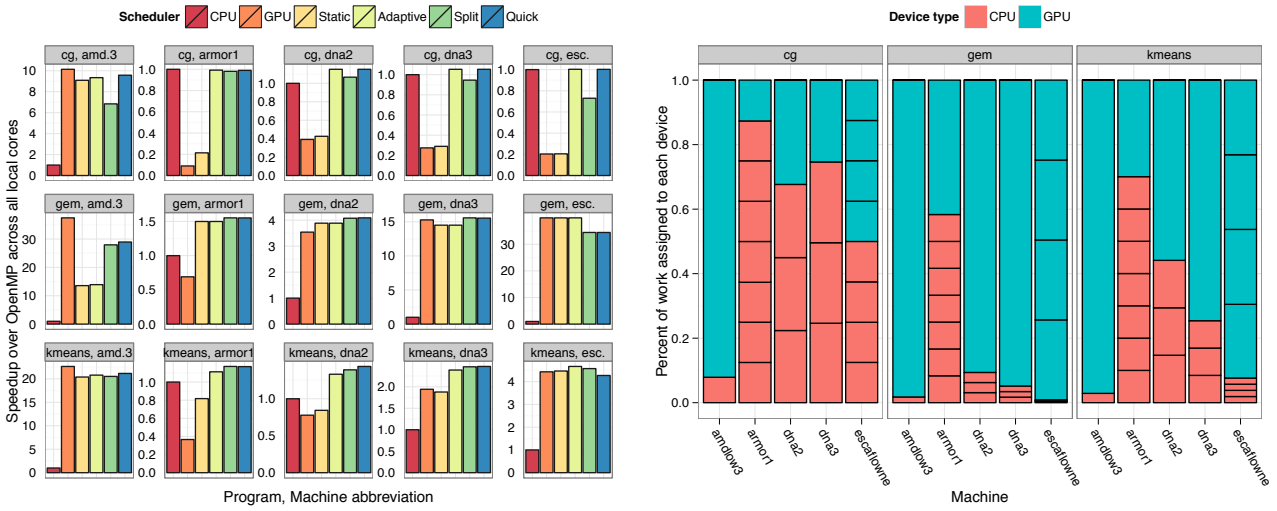
The most prominent feature of the results across systems is the significant change in overall speedup. In particular, amdlow3 exhibits consistently high speedups using the GPU for anything, although the Split scheduler suffers from high scheduling overhead due to the severe imbalance between its Intel Celeron processor and NVIDIA C2050 GPU. Even CG shows material speedups on the system, as much as $10\times$. More importantly, even though the speedup and overall performance shift across the various systems for each benchmark, the distribution of performance by scheduler is similar across them. Thus, finding the right scheduling algorithm to use with CoreTSAR is more about the application than the hardware. A user can spend the time to select the proper scheduling option once per region and then apply that choice across a wide array of machines. Further, these results show that the default adaptive scheduler is effective across hardware configurations, with only GEM on amdlow3 as an issue, as a result of its single iteration and being unable to recover. GEM's strong performance on the other devices also showcases the portability of our computed default division of work, which for that application is consistently near the optimal on systems with capable CPUs.

We also investigate the actual division of work between different processing elements in Figure 9b. Each benchmark has its own pattern, with CG using mostly CPU cores on most machines, except amdlow3, and GEM using the GPU almost entirely, except on armor1. While the distribution of overall performance in Figure 9a shows that the Adaptive scheduler often achieves high performance, the work division shows how differently each machine behaves, which CoreTSAR hides from the end user.

6. Related Work

Task scheduling as a mechanism for easing parallel programming has a long history. Traditional applications of the approach include dynamic loop parallelization in OpenMP [10] and Intel's TBB [25]. These mechanisms tend to offer simplified syntax for shared memory parallelism, but little to no support for heterogeneous architectures or distributed memory. With the rise of cluster and cloud computing platforms, frameworks like Charm++ [21] and MapReduce [11] perform a similar role, dividing tasks across distributed resources automatically. In addition to its role as a cloud computing model, MapReduce has been the target of various GPU acceleration projects including Mars [18], MapCG [19], and StreamMR [14]. While some GPU MapReduce implementations support multiple GPUs, they generally do not support coscheduling of CPUs and GPUs on the same problem.

Task scheduling policies, especially in OpenMP and Charm++, have also been the focus of significant research. Work by Ayguadé et al[3], directly influenced the design of CoreTSAR. They investigated the possibility of removing or extending the OpenMP scheduler clause by calculating the distribution of work in future passes



(a) CoreTSAR speedup across systems

(b) Work distribution across devices in each system, where a single device is a single black box, with the Adaptive scheduler

Figure 9: CoreTSAR evaluation across machines

through a region based on times seen for each core in previous ones. Their results showed the method was not always optimal but the solution was efficient and stable. Our ratio-based design works similarly, although with a different mechanism to determine the split between units, and explicit support for heterogeneous hardware.

With the proliferation of GPUs and other computational accelerators, task schedulers are being designed specifically for these environments. StarPU [2] schedules heterogeneous tasks and serves as the basis for various scheduling mechanisms. Other important work includes Qilin [22] and the work by Jiménez et al. [20]. Two major factors distinguish our work from these schedulers. First, they target overall compute bandwidth by scheduling at the granularity of function calls or kernels while we target compute latency by scheduling the work inside a single parallel region. Second, they require reimplementation of one's code in a new programming language (StarPU) or a new API (Qilin), or one must manually create a function for each architecture. While CoreTSAR currently requires one to split code manually, our Accelerated OpenMP extensions will allow the compiler to automate this mechanical process.

Accelerated OpenMP implementations are beginning to proliferate in industry, including HMPP [12], OpenACC [7], PGI Accelerator [27] and Cray Accelerated OpenMP [6]. Each one offers a method for a user to target multiple GPUs and CPUs by explicitly splitting the workload and targeting each device with a region or codelet. They do not offer coscheduling within a single region. As they all offer a different implementation of C or Fortran to GPU translation, and do not offer in-region coscheduling, they are orthogonal to our work. Our goal is to offer an extension to this model, using one of these as a platform for coscheduling rather than competing with them.

OmpSs [13] offers another implementation of Accelerated OpenMP in addition to those being slowly adopted by industry. This programming model includes a coscheduling mechanism that supports both CPUs and multiple GPUs. However, their evaluation concluded that two CPUs always beat any combination of CPU and one to two GPUs. Our approach yields significant speedups by using all available compute resources.

The dynamic load balancing scheme in Chen et al.'s [8] work provides an efficient work queue for load balancing and work distribution across multiple GPUs. Their approach can schedule individual blocks through the use of a persistent kernel on the GPU that receives work in block size chunks from the CPU. This model is somewhat orthogonal to our work. CoreTSAR could employ a work-queue design much like theirs inside of an Accelerated OpenMP implementation. That environment would offer greater control to the scheduler, while also allowing the compiler to preload all application kernels, thus avoiding the primary drawback of Chen et al.'s work by creating the offbeat structure that it requires.

The scheduling framework presented by Ravi et al. [23] provides an interesting counterpoint to our approach. Building on their generalized reduction framework and code generator [24], they present scheduling mechanisms for multicore systems with a single GPU. While we avoid the chunk scheduling scheme in order to avoid the additional transfer overheads, they present an alternative approach using chunk based scheduling while mitigating the overhead through runtime techniques.

Finally, HTS [26] is clearly the most closely related work. Therein the authors investigate extensions to Accelerated OpenMP to support coscheduling across CPUs and one GPU within a region. It evaluates similar schedules to those that we explore in this work, with the limitation that exactly two heterogeneous processors could be uniquely scheduled. CoreTSAR reworks this previous approach by extending the scheduling scheme to apply to an arbitrary number of devices allowing independent scheduling of individual cores as well as multiple GPUs. Most importantly, our new approach does not need to rely on static splitting or underlying schedulers between "homogeneous" devices, which allows us to handle some classes of imbalanced workloads directly and to manage NUMA effects within a class of device better.

7. Conclusion

We have presented the design and implementation of CoreTSAR, our automated task scheduler for accelerator-aware runtimes. We make three major contributions: the design of our scheduler for

adaptive scheduling across arbitrary numbers of heterogeneous devices; an implementation and optimization of that design; and our evaluation across five scientific codes, and five distinct systems. These contributions yield speedups as high as $3.74\times$ over the best achievable by using all cores and a single GPU in our test system. When compared to the original CPU performance on 8 cores, we achieve as much as $40\times$ for one benchmark. These results, along with the portability our schedulers displayed across systems, clearly motivate the addition of a co-scheduling interface, such as the `hetero()` clause that we propose, to Accelerated OpenMP.

As future work, we will investigate expanding CoreTSAR with a persistent kernel work-queue for finer grained GPU scheduling. In the main scheduler, CoreTSAR could automatically detect NUMA issues, and the association of GPUs, through PCI-E association, to CPUs and manage these automatically for greater performance. Finally, the memory management interface that we present is the first step towards a general interface for declaring the relationship between tasks and the portions of inputs and outputs that they require. Given that information, many schedulers, including ours, could automatically manage input and output, providing significant value especially as computers become more complex.

References

[1] Ramu Anandakrishnan, Tom R.W. Scogland, Andrew T. Fenley, John C. Gordon, Wu-chun Feng, and Alexey V. Onufriev. Accelerating Electrostatic Surface Potential Calculation with Multi-Scale Approximation on Graphics Processing Units. *Journal of Molecular Graphics and Modelling*, 28(8):904–910, 2009.

[2] Cdric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andr Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704, pages 863–874. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[3] Eduard Ayguadé, Bob Blainey, Alejandro Duran, Jesús Labarta, Francisco Martínez, Xavier Martorell, and Raúl Silvera. Is the Schedule Clause Really Necessary in OpenMP? In *WOMPAT'03: Proceedings of the Workshop on OpenMP Applications and Tools 2003*. Springer-Verlag, June 2003.

[4] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63 –73, 1991.

[5] M. Berkelaar, P. Notebaert, and K. Eikland. lp_solve:(mixed integer) linear programming problem solver. http://lpsolve.sourceforge.net/5.0/, 2003.

[6] James C. Beyer, Eric J. Stotzer, Alistair Hart, and Bronis R. de Supinski. OpenMP for Accelerators. In Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Mller, editors, *OpenMP in the Petascale Era*, volume 6665, pages 108–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[7] CAPS Enterprise, Cray Inc., NVIDIA and the Portland Group. The openacc application programming interface, v1.0. http://www.openacc-standard.org, Nov. 2011.

[8] Long Chen, O Villa, S Krishnamoorthy, and G.R Gao. Dynamic Load Balancing on Single- and Multi-GPU Systems. *2010 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12, 2010.

[9] M. Daga, T. Scogland, and W. Feng. Architecture-aware mapping and optimization on a 1600-core gpu. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 316–323. IEEE, 2011.

[10] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, March 1998.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51:107, January 2008.

[12] R. Dolbeau, S. Bihan, and F. Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[13] A. Duran, E. Ayguade, R.M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[14] M Elteir, Heshan Lin, Wu-chun Feng, and T Scogland. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 364–371. IEEE Computer Society, 2011.

[15] Andrew T. Fenley, John C. Gordon, and Alexey Onufriev. An Analytical Approach to Computing Biomolecular Electrostatic Potential. I. Derivation and Analysis. *The Journal of Chemical Physics*, 129, 2008.

[16] John C. Gordon, Andrew T. Fenley, and Alexey Onufriev. An Analytical Approach to Computing Biomolecular Electrostatic Potential. II. Validation and Applications. *The Journal of Chemical Physics*, 129:075102, 2008.

[17] S Grauer-Gray, L Xu, R Searles, and S Ayalasomayajula. Auto-tuning a High-Level Language Targeted to GPU Codes. *cis.udel.edu*.

[18] B He, W Fang, Q Luo, N K Govindaraju, and T Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM New York, NY, USA, 2008.

[19] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: writing parallel program portable between CPU and GPU. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM Request Permissions, September 2010.

[20] Vctor J. Jiménez, Llus Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In Andr Seznec, Joel Emer, Michael OBoyle, Margaret Martonosi, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 5409, pages 19–33. Springer Berlin Heidelberg, Berlin, Heidelberg.

[21] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++. In *OOPSLA '93 Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108. ACM Press, 1993.

[22] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 45. ACM Press, 2009.

[23] V T Ravi and G Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10, 2011.

[24] Vignesh T Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*. ACM Request Permissions, June 2010.

[25] J. Reinders. Intel Threading Building Blocks. 2007.

[26] Thomas R. W. Scogland, Barry Rountree, Wu-chun Feng, and Bronis R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *2012 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Shanghai, China.

[27] Michael Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, page 43. ACM Press, 2010.