CoreTSAR: Core Task-Size Adapting Runtime

Thomas R. W. Scogland* Wu-chun Feng* Barry Rountree† Bronis R. de Supinski†
* Department of Computer Science, Virginia Tech, Blacksburg, VA 24060
† Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551
tom.scogland@vt.edu wfeng@vt.edu rountree@llnl.gov bronis@llnl.gov

Abstract—Heterogeneity continues to increase at all levels of computing, with the rise of accelerators such as GPUs, FPGAs, and other co-processors into everything from desktops to supercomputers. As a consequence, efficiently managing such disparate resources has become increasingly complex. CoreTSAR seeks to reduce this complexity by adaptively worksharing parallel-loop regions across compute resources without requiring any transformation of the code within the loop. Our results show performance improvements of up to three-fold over a current state-of-the-art heterogeneous task scheduler as well as linear performance scaling from a single GPU to four GPUs for many codes. In addition, CoreTSAR demonstrates a robust ability to adapt to both a variety of workloads and underlying system configurations.

1 Introduction

While heterogeneous systems are becoming more popular, their programming models deter many potential users. Unlike adding more or faster CPUs, where existing programming models work without code changes, programs must be explicitly updated to use GPUs and other accelerators. Rather than grapple with unfamiliar programming models, users often run their CPU-only code on accelerated resources, leaving a significant portion of the computing resources idle. Accelerated OpenMP, our term for a class of directive-based programming models including OpenMP for Accelerators [9] and the PGI accelerator model [21] among others, can ease this transition by allowing users to target accelerators with a familiar OpenMP-style syntax. However, Accelerated OpenMP is not a panacea: current iterations help one *move* their computation to a *single* accelerator with straightforward syntax. Once moved however, there is no way to workshare a loop across multiple devices without *manually* targeting each device.

In order to target, for example, a GPU and four CPU cores, a user must manually split the work, run that work on each separate device, and manually merge each result. Any load balancing, coherency, or runtime adaptation of any kind must be reimplemented by every user. So, while Accelerated OpenMP can parallelize serial code via annotation, it lacks the ability to scale and to load-balance work transparently on the hardware found at runtime.

Our work enables safe and efficient worksharing across devices in Accelerated OpenMP. To do so, we must address two primary concerns. First, we manage memory input and output across multiple address spaces without requiring alterations to the associated parallel loop. Second, we divide work across devices with vastly different computational capabilities fairly and efficiently. In all, our CoreTSAR (Task-Size Adapting Runtime) library automates the scheduling, load balancing, and cross-device data management for safe and efficient worksharing. This paper presents the design and implementation of CoreTSAR and the extended Accelerated OpenMP syntax to integrate its functionality. Specifically, we make the following contributions:

- The design and syntax of a multi-target, worksharing construct for Accelerated OpenMP;
- The design, implementation, and optimization of our scheduling and memory management library, CoreTSAR, which can be used with any Accelerated OpenMP compiler/runtime or with CUDA and CPU OpenMP directly;
- Seven adaptive scheduling policies, spanning from a low-overhead but coarse-grained adaptive approach to a chunk-based, fine-grained scheduling approach for distributing work.
- A rigorous performance evaluation of CoreTSAR that demonstrates how runtime scheduling can significantly improve performance while maintaining programmability.

The rest of the paper is composed as follows. Section 2 offers motivation and background. 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. Section 5 present our results. Related work follows in Section 6 and conclusions in Section 7.

This work was supported in part by the Air Force Office of Scientific Research (AFOSR) Computational Mathematics Program via Grant No. FA9550-12-1-0442, NSF I/UCRC IIP-1266245 via the NSF Center for High-Performance Reconfigurable Computing (CHREC), and a DoD National Defense Science & Engineering Graduate Fellowship (NDSEG)

```
1  void runGemm(T **a_a, T **b_a, T **c_a) {
2    T *a = *a_a, *b = *b_a, *c = *c_a;
3  //OpenMP
4  #pragma omp parallel for
5  //Accelerated OpenMP
6  #pragma acc region for copy(c[0:N*N])    \
7          copyin(a[0:N*N],b[0:N*N])
8  //Accelerated OpenMP + extension
9  #pragma acc region for part_copy(c[1:N][0:N])\
10         copyin(b[0:N*N]) part_copyin(a[1:N][0:N])\
11         hetero(1, all, adaptive)
12   for (int i = 0; i < N; ++i) {
13     for (int j = 0; j < N; ++j) {
14       c[(i*N) + j] *= B;
15       for (int k = 0; k < N; ++k) {
16           c[(i*N)+j] += A * a[(i*N)+k] * b[(k*N)+j];
17 } } } }
```

```
1  __global__ void
2  cudag(T *a, T *b, T *c, T A, T B, int n) {
3    uint i = blockIdx.x * blockDim.x + threadIdx.x;
4    if(i < n) {
5      for (int j = 0; j < n; ++j) {
6        c[(i*N) + j] *= B;
7        for (int k = 0; k < n; ++k) {
8          c[(i*N)+j] += A * a[(i*N)+k] * b[(k*N)+j];
9  } } } }
10 void runGemm(T **a, T **b, T **c) {
11   T *ca, *cb, *cc; dim3 dB, dG;
12   size_t size = N*N*sizeof(T);
13   dB.x = 64; dB.y = dB.z = 1;
14   dG.x = (N/dB.x)+1; dG.y = dG.z = 1;
15   cudaMalloc(&ca, size);
16   cudaMalloc(&cb, size);
17   cudaMalloc(&cc, size);
18   cudaMemcpy(ca,*a,size,cudaMemcpyHostToDevice);
19   cudaMemcpy(cb,*b,size,cudaMemcpyHostToDevice);
20   cudaMemcpy(cc,*c,size,cudaMemcpyHostToDevice);
21   cudag<<<dG,dB>>>(a, b, c, A, B, N);
22   cudaMemcpy(*c,cc,size,cudaMemcpyDeviceToHost);
23 }
```

```
1  #pragma omp target device(smp,cuda)
2  void gemm(T *a, T *b, T *c, int i, T A, T B, int n);

3  #pragma omp target device(smp) copy_deps
4  #pragma omp task input ([n] a, [n*n] b) inout ([n] c)
5  void gemm(T *a, T *b, T *c, int i, T A, T B, int n) {
6    for (int j = 0; j < n; ++j) {
7      c[j] *= B;
8      for (int k = 0; k < n; ++k) {
9        c[j] += A * a[k] * b[(k*n)+j];
10 } } }

11 #pragma omp target device(cuda)
12 __global__ void
13 cudag(T *a, T *b, T *c, int i, T A, T B, int n)
14 { unsigned int j = blockIdx.x * blockDim.x + threadIdx.x;
15   if(j < n) {
16     c[j] *= B;
17     for (int k = 0; k < n; ++k) {
18       c[j] += A * a[k] * b[(k*n)+j];
19 } } }

20 #pragma omp target device(cuda) copy_deps implements(gemm)
21 #pragma omp task input ([n] a, [n*n] b) inout ([n] c)
22 void gemm_gpu_wrap(T *a, T *b, T *c, int i, T A, T B, int n)
23 {
24   __global__ void
25   cudag(T *a, T *b, T *c, int i, T A, T B, int n);

26   dim3 dB, dG;
27   dB.x = 64; dB.y = dB.z = 1;
28   dG.x = (n/dB.x)+1; dG.y = dG.z = 1;
29   cudag<<<dG,dB>>>(a, b, c, i, A, B, n);
30 }

31 void runGemm(T **a, T **b, T **c) {
32   for (int i = 0; i < N; ++i) {
33     gemm(a[i], b[0], c[i], i, A, B, N);
34   }
35 #pragma omp taskwait
36 }
```

Fig. 1: A basic GEMM kernel as implemented in OpenMP variants (top left), CUDA (bottom left) and OmpSs (right).

2 Background and Motivation

As heterogeneous systems spread through the marketplace, so do programming models that target them. While a variety of programming models exist, most fit into one of three categories: (1) loop-offload models; (2) block-and-grid models; and (3) blocked-task models.

Loop-offload models include variants of Accelerated OpenMP [9], OpenACC [3] HMPP [13], PGI accelerator directives [21], and Intel OpenMP offload extensions for their Xeon Phi coprocessors. They extend an OpenMP-like annotated, serial, source model with data-movement declarations to offload work to a device with a distinct address space. The top left of Figure 1 shows a basic molecular modeling kernel (GEMM) implemented serially with OpenMP, Accelerated OpenMP, and our proposed Accelerated OpenMP extensions. With no pragmas, the loop runs serially, as one would expect. The OpenMP pragma on line 4 workshares the loop across CPU cores. Accelerated OpenMP's pragma (lines 6-7) adds explicit *in* copies of the `a` and `b` arrays and an *inout* copy of `c`. Each of these first two pragmas workshares the loop iterations across a single address space, *either* CPU cores *or* a single GPU. We discuss the third pragma at the end of this section.

Block-and-grid models include CUDA [1] and OpenCL [2]. These low-level models specifically target GPU-like hardware by offloading blocks or groups of threads to an array of cores, each of which is a SIMD unit. Generally these cores share memory with one another but not directly with the CPU. The lower left of Figure 1 shows an example using CUDA. In addition to changing the array accesses, explicit memory allocation and copies are required to move data to and from the device. The loop is converted into a grid of threads, each of which executes a single iteration in the `cudag()` kernel, which must be called with the number of blocks and threads per block. As with the loop-offload example, this code uses exactly one GPU.

Blocked-task models, like OmpSs [14] and StarPU [**?**], specify tasks and their dependencies in terms of blocks of data (and sometimes other tasks). The right-hand code-block in Figure 1 uses OmpSs to implement the GEMM kernel with load balancing across CPUs and GPUs, so it contains *both* CPU *and* CUDA kernels, both aliased to the `gemm()` function by the compiler. Each call to `gemm` is given the start address of the block, in this case a row, to process. These calls are converted into tasks, which are enqueued into the OmpSs runtime with their data. Each task can then be scheduled, individually, on any device an implementation is available for. Since the task size is *fixed*, each task must encompass enough work to occupy all compute units on a GPU long enough to amortize the overhead of scheduling it; on the other hand, each task must also be small enough not to overload a single CPU core.

Each programming model has its advantages and dis-

advantages. The block-and-grid approach (e.g., CUDA or OpenCL) is highly efficient on the GPU and offers maximum control over them. The loop-offload version requires the least change from serial or OpenMP code, but it offers less control. Blocked-task models offer control through direct use of the other models as well as automatic load-balancing across all compute resources. Unfortunately, they also require the greatest departure from the original code.

Therefore, we need a programming model that offers the performance of block-and-grid models, flexibility of blocked-task models, and programmability of loop-offload models. Our proposed extensions, along with our prototype library implementation, brings us closer to this goal by introducing work-sharing across devices to Accelerated OpenMP without requiring a specific task granularity from the user. The third pragma, in the upper left of Figure 1 (lines 9-11) illustrates how our proposed extension would work-share the GEMM loop across an arbitrary number and type of supported devices. Thus, it offers more flexibility in the region than even blocked-task models, while maintaining the serial loop as written.

```
//items in {} are optional
#pragma acc region \
  hetero(<cond>{,<devs>{,<sched.>{,<ratio>{,<div>}}}})\
  part_copy{in/out}(<var>{(<size>}[<cond>:<num>{:<halo>}])\
  persist(<var>)
#pragma acc depersist(<var>)
```

hetero() inputs

<cond>	Boolean, true=coschedule, false=ignore
<devices>	Allowable devices (cpu/gpu/all)
<scheduler>	Scheduler to use for this region
<ratio>	Initial split between CPU and GPU.
	How many times to divide the iteration space

part_copy() and {de}persist() inputs

<var>	Variable to copy.
<size>	Size of each "item" in the array/matrix.
<cond>	Whether this dimension should be copied.
<num>	Number of items in this dimension.
<halo>	Number of halo elements required.

Fig. 2: Our proposed extension

3 Design

This section presents the design of our proposed extension, schedulers and memory management infrastructure. CoreTSAR safely divides annotated, un-blocked, serial loops, as used in many traditional OpenMP applications, and schedules them across heterogeneous resources. We add a clause to Accelerated OpenMP that is similar to the `schedule()` clause. The OpenMP programming model imposes the following constraints on our design:

1) Use existing, unchanged code in the Accelerated OpenMP loop region;
2) Treat the accelerated loop as a group of related tasks that are defined by the loop code and the region directive including its associated clauses;
3) Maintain data consistency outside of the region and do not alter data accesses in the existing loop body although we can extend the data copy clauses of the region.

By following these constraints we preserve programmability while adding significant new functionality.

3.1 The Proposed Extension

The CoreTSAR interface consists of two parts, which Figure 2 depicts. The `hetero()` clause specifies how to schedule the region and which classes of device should be considered. For memory management, we add the `part_copy()` clauses to provide the runtime with sufficient information to partition input and output data for the region safely.

Our clauses are permitted on the accelerator directive or on any top-level accelerator loop construct. Unlike normal copy clauses, `part_copy` is not allowed on data clauses as it only has meaning when directly associated with a loop. We still support data regions, but only for cases where complete replication of the input/output is desired, as opposed to only those data regions that are required. We define the properties of our clauses in greater detail in the following sections.

3.2 Scheduling

In order to workshare the iterations in a given region efficiently, we must offer appropriate scheduling policies. Each policy in CoreTSAR treats the iterations within a loop region as a group of related tasks, which allows us to select the scheduling granularity adaptively. For example, CoreTSAR can break a region with 10,000 iterations into four chunks or a thousand or any number less than 10,000 for scheduling, without modification and without user intervention. This capability is critical for efficiently scheduling across heterogeneous systems, as a single grain size is rarely appropriate for all available devices.

Existing OpenMP schedules for CPUs divide the iteration space either evenly across processors statically or into *chunks* that are assigned dynamically. The static version is simple, efficient to implement, and consistent, but does not load-balance. Alternatively, OpenMP's chunk based schedules (*dynamic* and *guided*) load-balance well in homogeneous architectures. However, they suffer high overhead due to synchronization at each work-request and especially as a result of the lack of data locality their dynamic algorithms cause. In heterogeneous systems they would also incur repeated kernel launches and data transfers. We dealt with these issues in our initial work with CoreTSAR [20], by designing a set of adaptive schedules that statically divide the work within each pass through a region but load-balances across passes. This scheme proved effective, but it does not tolerate imbalanced workloads well, whereas chunk-based schemes can. To address that case and broaden our evaluation, we have developed a number of new chunk-based designs as well as a hybrid of the two approaches.

I =total iterations
i_j =iterations for compute unit (CU) j
f_j =fraction of iterations for CU j
p_j =recent time/iteration for CU j
n =number of CUs
$t_j^{+/-}$ =time over/under equal

$$min(\sum_{j=1}^{n-1} t_j^+ + t_j^-) \quad (1)$$

$$\sum_{j=0}^{n} f_j = 1 \quad (2)$$

$$f_2 * p_2 - f_1 * p_1 = t_1^+ - t_1^- \quad (3)$$

$$f_3 * p_3 - f_1 * p_1 = t_2^+ - t_2^- \quad (4)$$

$$\vdots$$

$$f_n * p_n - f_1 * p_1 = t_{n-1}^+ - t_{n-1}^- \quad (5)$$

Fig. 3: Our adaptive scheduler's deviation minimization algorithm as a linear program, variables at left

3.2.1 Static/Adaptive Schedulers

Our static and adaptive schedulers [19], [20] predict the time that each device will take to complete an iteration in the next pass and generate a single task for each device sized such that all finish the region as close to simultaneously as possible. These schedulers make two assumptions: (1) the average runtime of an iteration in the region is constant on a device; and (2) subsequent passes through the region have the same performance ratio as the previous pass. Also, our schedulers begin with a default time per iteration for each device until we have gathered runtime data. This default is either a user-defined value, one saved from a previous run, or one based on the formula *1/(deviceSIMDWidth/baseDeviceSIMDWidth)*. While we do not claim that this formula accurately models the relative performance of devices, in practice we have found it to be accurate for dense floating-point kernels. We leave further exploration of default values as future work.

The linear program in Figure 3 uses the time per iteration value for each device to calculate the fraction of the total available iterations that should be allocated to each device. In words, the program finds the fractions of work that result in the minimum deviation between predicted execution times. (Our initial version of this linear program calculated the optimal number of iterations directly as an *integer* solution, giving theoretically optimal splits based on our model. This integer solution, however, was impractical to run online due to high calculation costs.) The linear program formulated in Figure 3 dramatically reduces the calculation costs and is designed to still yield an optimal fractional result, allowing the solution to be off by up to one iteration on each device but decreasing the computation time by several orders of magnitude.

Our *static* schedule applies this linear program to the default, or supplied, values once at the beginning of the first pass through a region, then reuses the result thereafter. The adaptive schedules (*adaptive*, *split* and *quick*) use a first pass with the static schedule as a training phase. The first time that we encounter a CoreTSAR region, we assign work based on the static schedule and then measure the times on each device. For each following scheduling decision, we use measured times per iteration in the linear program, converging on a more efficient schedule. Our design intentionally includes all recurring data transfer and similar overheads required to execute an iteration on a particular device, naturally incorporating data transfer and launch overheads.

Adaptive trains on the first full pass through the region, then adapts at the beginning of each subsequent pass. *Split* is designed to adapt within regions that either cannot tolerate a full pass with a poor schedule, or only run once per application run. *Split* breaks each pass into several evenly split subpasses, based on the `div` input. It treats each subpass as the same as a full pass with *adaptive*. While *split* provides better, and earlier, load-balancing for some applications, it increases overhead in each pass. *Quick* balances between *split* and *adaptive* by executing a small subpass for its first training phase, similarly to *split*. It then schedules and runs all iterations remaining in the first pass, and uses the adaptive schedule for all subsequent passes. This schedule suits applications that cannot tolerate a full pass using the static schedule or the overhead of extra scheduling steps in every pass.

3.2.2 Chunk Schedulers

Chunk schedulers are exemplified by the OpenMP *dynamic* schedule, in which a chunk size specifies the number of iterations assigned to each thread each time it requests work. These schedulers effectively use a work queue approach, which offers natural load balancing. While it is a classic load balancing approach, it is most effective when used with homogeneous computing resources with fast synchronization mechanisms, which is not the environment that CoreTSAR targets. Thus, we present variations on this method for hybrid systems.

Specifically, we design three new schedules (*chunk*, *chunk static* and *chunk dynamic*). *Chunk* serves as our baseline chunk scheduler, and is functionally identically to OpenMP's *dynamic* schedule. *Chunk static* scales the chunk size for each device based on the same scheme used in the *static* schedule above. Thus, larger chunks are provided to devices that complete their iterations faster, with the goal of each chunk running for the same length of time regardless of the target device. Finally, *chunk dynamic* begins in the same way as *chunk static* then dynamically adapts the chunk size for each device based on their performance. Unlike the adaptive schedulers, it does not employ the linear program to determine the new chunk size since the chunk schedulers do not have a natural barrier point where all times have been updated. Instead, it employs an annealing approach that computes a weighted average of the time per iteration for each device, and attempts to reduce the time per iteration by increasing or decreasing the chunk sizes. For example, if the time per iteration on a device decreases with an increased chunk size, *chunk dynamic* again increases that chunk size. In this design, each device is independent and does not block on the others in order to adapt.

3.2.3 Hybrid Scheduler

In addition to the schedulers that are strictly chunk or ratio based, we also investigate a *hybrid chunk* schedule that begins as a chunk dynamic schedule and after the first pass transitions into the adaptive scheduler. *Chunk dynamic* adapts and load balances quickly during the first pass while refining the split. However, after that first pass, it incurs unnecessary overhead in contention and memory transfers, where adaptive excels. Using *chunk dynamic* in place of *static* for the training phase of *adaptive* naturally fuses the advantages of both schedulers.

3.3 Memory Management

In order to maintain memory coherency across address spaces while dynamically splitting the region, CoreTSAR requires information about the memory access pattern of each iteration of the loop. The primary design goal of our memory manager is to support unblocked input and output data naturally. Thus preserving the data layout of the serial code. Our interface covers a wide range of cases while a method to specify any possible association is an ongoing topic of research, including our future work. As we discuss in Section 4, our prototype library implements the memory management for all examples evaluated in Section 5.

For each variable the `part_copy()`, or partial copy, interface requires at least a variable name, one dimension to copy, and the number of items in that dimension. Given the clause `part_copyin(a[1:N])`, `a[i]` will be copied, where `i` is the current loop iteration, to the device that executes that iteration. If a range of `i` from $5 - 500$ is assigned to a device, `a[5-500]` will be copied. Thus, the partial copy is associated with the loop iteration(s) of the loop being split.

Figure 4 displays two simple examples of patterns produced by our memory-association syntax. The top example specifies that a 10×10 matrix is being registered to the region, and the iterator will be associated with the column dimension, assuming C ordering, since the column dimension's condition is true. The lower example selects the row dimension instead, and additionally specifies that one halo element is required on either side of that dimension. This pattern is typical of stencil-type computations, where halo values are required as input, but are not updated by the device reading them, having the halo argument makes supporting such associations natural.

Our interface does not directly support random access output, reverse indices or indirect indices. For input, these can all be supported at the cost of additional overhead by copying the entire data set, since the input process is non-destructive.

3.4 Example Usage

Figure 5 shows how to use our proposed interface to implement the example in Figure 1. All options, including those that use default values, are specified for

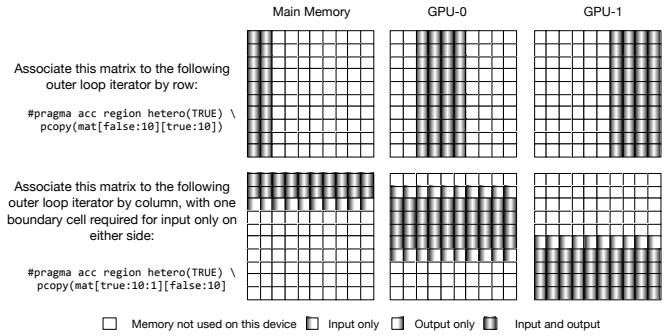

Fig. 4: Example memory association patterns, assuming a pass in which two iterations are assigned to the CPU device, and four each to two GPUs.

```
void runGemm(T **a_a, T **b_a, T **c_a) {
  T *a = *a_a, *b = *b_a, *c = *c_a;
#pragma acc region for part_copy(c[1:N][0:N])        \
        part_copyin(a[1:N:0][0:N:0]) copyin(b[0:N*N])\
        hetero(1, all, adaptive, default, 10)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; ++j) {
      c[(i * N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
      }
    }
  }
}
```

Fig. 5: Our proposed extension to accelerated OpenMP.

`part_copyin(a...)` and `hetero(...)` in the example. The minimum necessary to specify the copy correctly are used for `part_copy(c...)`. In this example, the loop will always be split across devices using the adaptive scheduler with the default ratio and a div of 10. The copies specify that the `a` array is two-dimensional, of size N by N, made up of elements of size `sizeof(T)`, and that iteration i requires row i of `a` but not column i. The `c` copy specifies the same as for `a` except that it should be copied both in and out. The traditional `copyin()` clause from accelerated OpenMP is used for `b` since all participating devices need access to the full region. Complete output, in the form of `copyout()`, is not allowed however because there is no way to resolve the changes between versions. We may investigate this in future work.

4 Implementation

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

4.1 GPU Back-off

Some applications are not amenable to being run on GPUs, or at least the GPUs present in some systems. While iterations of a region may benefit greatly from

running on an NVIDIA c2075, they may perform poorly on an NVIDIA GeForce GT 520. In order to maintain portability across disparate accelerator and CPU capabilities, CoreTSAR implements GPU Back-off support in all adapting schedulers.

The back-off system is implemented differently for each of the two scheduler types. In the adaptive schedulers 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 of passes (default is two). We use multiple iterations since under certain circumstances, such as loading large persistent datasets for the first time, or an inappropriate initial amount of work, a device can be erroneously classified as slower than the CPUs. With the chunk schedulers, we base the decision on whether a given GPU completes fewer iterations than the slowest CPU core during each pass of a configurable number of passes. This difference compensates for the sometimes highly variable time per iteration when bootstrapping chunk schedulers across initial data copies, which can cause false conversions with the adaptive back-off scheme. We discuss the effects of this extension further in Section 5.

4.2 Memory Management

The existing memory interface of Accelerated OpenMP is insufficient to express the relationships necessary to handle certain kinds of memory association. While Accelerated OpenMP does natively support copying a subset of an array, it does not support copying multiple subsets of one array, nor does it support non-contiguous rectangular sections such as a subset of the columns of a 2D array.

In order to support our desired memory association interface, CoreTSAR implements its own memory manager, using the `deviceptr()` clause to pass CoreTSAR managed memory into Accelerated OpenMP regions. We offer a straightforward syntax by which users specify the data required by a given iteration. Given that information, CoreTSAR automatically copies the ranges of data required by whatever iterations are assigned to a given device for that pass. When possible the memory manager uses pinned memory to accelerate copies, as well as asynchronous copies to and from the device in order to overlap them with scheduling and synchronization.

Currently, the CoreTSAR memory manager handles a restricted set of partial copies. In addition to the straightforward one-to-one relations, CoreTSAR also supports stencils through padding, and row, column or planar associations on two and three dimensional matrices. In order to support reductions we provide an API inspired by user-defined reductions in OpenMP 4.0. We discuss the details further in Section 4.4. While only a subset of the possible cases, these mappings are sufficient to implement all benchmarks evaluated in this paper.

4.3 Data Packing and Padding

Our original implementation of the memory interface [20] had a material weakness. That version of CoreTSAR allocated the full size of each memory region on each device in order to preserve offset accuracy. In other words, any input or output array/matrix supplied to CoreTSAR was allocated in full in all participating address spaces. Managing subset allocation and access without invalidating offsets and iterator values is a difficult problem, especially in languages like C.

We have redesigned CoreTSAR to support three kinds of regions depending on how the data is mapped. The first, and most simple, case is a one dimensional partial array or two dimensional array that is associated by rows. Since all of the resulting subsets are contiguous, the runtime provides an offset pointer that can be indexed by the original offsets without issue. No further action or overhead is required for this case, and a significant amount of storage on accelerators can be saved. The second case is where a two or more -dimensional array is associated by columns. CoreTSAR can pack these, but must have control over the calculation of offsets into the resulting matrix. As such, we handle this case in our translator for contiguous arrays accessed with `array[i][j]` style syntax, but currently do not support dynamically-allocated C arrays accessed with the `array[i*row_size + j]` syntax, though these can be supported by directly using the C API functions. Third, associations can use both the row and column associations, resulting in a region resembling a plus-sign being assigned to each device. Since these require the full range in both rows and columns, even though they may not need the corners, CoreTSAR is forced to allocate the full size of such arrays.

By allowing data regions to be packed, CoreTSAR gains two extra capabilities beyond reducing memory usage on target devices. The packing functionality allows any chunk-based scheme to place a low bound on memory use by selecting a small chunk size. This allows large data sets on the host to be streamed through accelerators without enough memory to hold even their assigned sub-part of the problem. When used in this mode however, CoreTSAR becomes similar to a blocked-task system, including the increased task management and data-transfer overhead that implies.

Perhaps more importantly however, the capability to adjust indexing, as described for column-wise associated multi-dimensional arrays, allows regions not only to be shrunk to save space, but also padded for alignment. As is well known, memory alignment is important for the performance of SIMD computations and coalesced memory accesses are important to the performance of GPU kernels. Given the ability to pad rows beyond the data assigned to each device, or even rows of data that are mis-aligned by the user, CoreTSAR can ensure that each row is aligned for most efficient access on each target. We implement this optimization by ensuring that

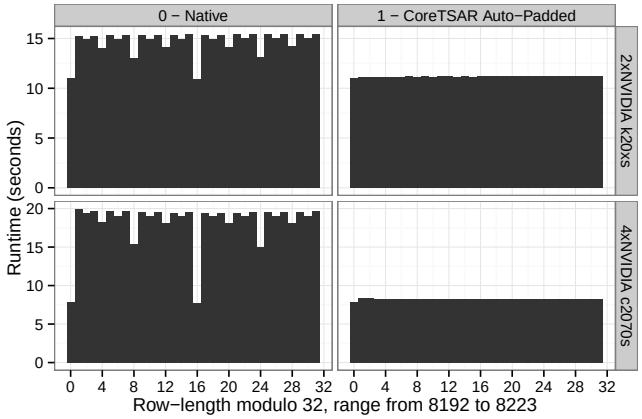

Fig. 6: Runtime of GEMM kernel on square matrices, statically scheduled across GPUs only with and without auto-padding

the length of each row in a matrix is a multiple of the target device's SIMD width. While in some cases this choice is more strict than required, it is consistently sufficient to ensure reasonable alignment.

Figure 6 shows the effect that even small changes in row-width can have on performance without padding, and how our auto-padding helps. The figure represents the performance of a general matrix multiplication kernel when run with row and column lengths ranging from 8,192 to 8,223 elements in increments of one, specifically the x-axis values are the number of elements over 8,192 in each row. On our primary evaluation system, with four c2070 GPUs, a square matrix of size 8192×8192 runs in 7.9 seconds. Increasing that size by only one element on each side more than doubles that runtime to 19.5 seconds. In fact, every odd-numbered increase in size takes approximately the same increased time, while each power of two increase does better up to 16, or a total of 8208, which performs the same as the original 8192. Another system with a pair of k20x GPUs shows a nearly identical stepped pattern as well. The difference in performance is somewhat smaller, ranging from 10.9 seconds at the zero and 16 positions and 15.2 at the odd offsets.The L2 read request performance counters provide a partial explanation for the wide range in performance. When padding is enabled, the difference in L2 requests with a multiple of 16 row length is consistently within 10%; for any odd length it balloons to 80% more L2 accesses for the un-padded version. This increase is due to a greater number of reads being required to accommodate the mis-aligned read requests of each warp on the GPUs, increasing contention and lowering cache efficiency overall. On the right-hand-side of the plot, you can see that CoreTSAR's automatic padding smooths out these issues. Also, when dividing a data-set column-wise, this padding support can ensure alignment even when the appropriate amount of work to be assigned is not a multiple of the target device's native SIMD width, an important consideration for several of our benchmarks.

4.4 CoreTSAR API and Usage

This section describes the low-level API to the CoreTSAR library in detail.

ctsar_init Initializes an instance of the CoreTSAR runtime, one such instance should be used for each region that is to be separately scheduled. The parameters allow a user to set the default scheduler, allowed devices, the default time per iteration for each accelerator as an array of doubles (NULL for defaults), and how finely the split and quick schedulers should divide regions (NULL for default).

ctsar_next Computes the division of work for the region associated with `c` based on `size` total iterations. This function is also responsible, updating appropriate memory regions on each target device and starting timers to evaluate each device's performance.

ctsar_loop In order to support *split*, *quick* and the chunk schedulers, CoreTSAR must reevaluate the loop with each thread repeatedly. The `ctsar_loop` function serves as the condition for a do/while loop surrounding each region. In addition to managing repeats, the loop function is responsible for synchronization, GPU back-off support, copying data back from all devices, completing reductions, and calculating performance statistics at the end of each pass.

ctsar_reg_mem{_2d} These functions register a host buffer with CoreTSAR. The full version takes a pointer to CPU memory, the size of an element of the input data, the number of element in each row/column, the number of halo elements required, and a flag option that allows the user to control copy direction and type. The non-2D version is shorthand for 1D arrays. The return value is a pointer to the memory assigned to the calling thread, which may or may not be identical to the original pointer.

The flags value controls whether memory is copied in or out or both, as well as whether to copy persistently, partially by rows or partially by columns and whether padding is to be allowed, if it is, an extra output parameter for the new row size is required. Partial copies are integral to the correct functioning of CoreTSAR as they make automatic merging of output possible. They also improve performance of input operations. The 2D interface supports all specifications discussed in Section 3, except that it does not handle matrices with dimensionality higher than 2.

Regardless of the flags, CoreTSAR allocates an appropriate size buffer on the device associated with the calling thread. If the region is set to persistent, data is immediately and asynchronously copied from the CPU array into the newly allocated memory, where it resides until it is explicitly removed with a call to `ctsar_unreg_mem()`.

ctsar_unreg_mem De-registers the pointer from the region instance, frees the memory that stores the state of the data, and frees persistent regions.

ctsar_retarget_mem Re-target allows a user to specify that the region already allocated for pointer `old` should

```
void runGemm(T **a_a, T **b_a, T **c_a) {
  ctsar * s = NULL; int div = 10;
  ctsar_init(&s,N,CTSAR_ADAPTIVE,CTSAR_DEV_ALL,NULL,&div);
#pragma omp parallel default(shared)
  do{
    T *a = ctsar_reg_mem(s, a_a[0], sizeof(T)*N, N,
            CTSAR_MEM_PARTIAL | CTSAR_MEM_INPUT);
    T *b = ctsar_reg_mem(s, b_a[0], sizeof(T)*N, N,
            CTSAR_MEM_INPUT);
    T *c = ctsar_reg_mem(s, c_a[0], sizeof(T)*N, N,
            CTSAR_MEM_PARTIAL | CTSAR_MEM_INOUT);
    ctsar_next(s,N);
#pragma acc region for deviceptr(a,b,c) independent \
        if(ctsar_get_type(s) == CTSAR_DEV_GPU)
    for (int i = CSTART(s); i < CEND(s); ++i) {
      for (int j = 0; j < N; ++j) {
        c[(i * NJ) + j] *= B;
        for (int k = 0; k < N; ++k)
          c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
    } }
  }while(ctsar_loop(s));
}
```

Fig. 7: CoreTSAR library version of GEMM

be used to store the data pointed to by `new` on all devices. A typical use is to swap buffers for double buffering, although it can also be used to implement blocked data transfers by re-targeting a pointer to the new start pointer before entering a region.

ctsar_reg_reduc This function registers a reduction. Because each memory space will have its own reduction result, CoreTSAR must safely initialize the temporary variables in each memory space and combine those results into a meaningful final value. The identity pointer points to an appropriate initial value to use on each device. For example, in a sum the identity would usually be 0, in a product 1, and so on. The `item_size` specifies the size of the elements to be reduced. The `reduc` argument is function pointer that should accept two void pointer arguments, the first of which is both a value to be reduced and the output, the second is another value to be reduced. This function is called repeatedly to accumulate the final value as each device finishes execution. For simple reductions, the body of the function can be as simple as `*((int*)a)+=*((int*)b)`.

CSTART/CEND Macros used to retrieve the start and end values to use for iteration in the loop region.

Figure 7 presents an example using this interface to implement the extension as presented in Figure 5. In this example, CoreTSAR is initialized with the adaptive scheduler, default ratio, and div of 10. The parallel do-while loop allows our library to reevaluate the code region as necessary by looping with the `ctsar_loop(s)` call until done. The data regions are registered, as partial input, complete input, and partial input/output, and the appropriate pointers for those data regions on each device are returned into the local copies of pointers `a`, `b` and `c`. The `ctsar_next()` call calculates the number of iterations to be completed in this pass by each device. Once it is complete, the `CSTART()` and `CEND()` macros return the appropriate iterator range values for the device that evaluates them. This syntax can either be used manually, or generated by our python/libclang-based source-to-source translator.

While the code is extended significantly around the loop, we do not replicate or alter any code in the loop body. The Accelerated OpenMP `if()` clause determines if a thread runs on a GPU or CPU core. If the device is a CPU, the loop is run serially on the associated core completing its assigned iterations. If it is a GPU-controlling thread, the `acc region` directive workshares the assigned iterations across the associated GPU. All codes used in our evaluation are implemented in this fashion.

5 Evaluation

This section evaluates the CoreTSAR library. We compiled all benchmarks with the PGI Accelerator compiler compiler suite version 12.9. Optimization flags are `-acc -ta=nvidia -O3 -mp=allcores`. Table 1 lists our test platforms. Unless otherwise specified, tests were run on escaflowne. In tests with GPUs enabled, one CPU core is used to control each selected GPU and 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 in all measurements.

Reported times and speedups include all activity that the original OpenMP CPU code did not require, 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 device to complete its assigned iterations, from which we can compute the time that devices wait for others to complete, the time spent to calculate the split for the next pass and, as a subset of that, the time to solve the linear program. Finally, we track the time per iteration for each device, as described in Section 3.

5.1 Benchmarks

We use four applications and the PolyBench/GPU [15] benchmark suite in our evaluation. CG [8] is a direct port of the NAS conjugate gradient benchmark. GEM [4] is a molecular modeling application for the study of the electrostatic potential along the surface of a macromolecule that has been extensively studied for GPU optimization [12]. Helmholtz is a discrete finite difference code that uses the Jacobi iterative method to solve the Helmholtz equation. K-means is a popular iterative clustering method. Our implementations of the 15 PolyBench/GPU benchmarks execute each computational kernel 10 times to mimic use in an iterative scientific application more closely. Tests at 5 and 15 kernel executions yield similar relative results. Since we are evaluating scheduling behavior, and not computational kernel performance, we made minimal changes in porting each benchmark. As such, our computational kernels are not optimized for the GPU other than by

System name	CPU Model	CPU Cores/die	CPU Dies	CPU RAM (MB)	GPU Model	GPU Cards	GPU Cores	GPU RAM (GB)
amdlow3	E3300	2	1	2,012	Tesla C2050	1	448	3
armor1	E5405	4	2	3,964	GeForce GT 520	1	48	1
dna2	i5-2400	4	1	7,923	GeForce GTX 280	1	240	1
escaflowne	X5550	4	2	24,154	Tesla C2070	4	448	6
hokiespeed	E6545	6	2	24,154	Tesla M2050	2	448	3

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

the compiler. Nonetheless, CoreTSAR can easily support optimized implementations through the same syntax.

For our purposes, benchmarks can 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 characterizes these properties for each benchmark. The table exhibits a wide range in number of passes through the parallel region – 1 to 1900 passes in the applications, and as high as 10240 passes for the GRAMSCHMIDT benchmark. Our adaptive scheduler operates primarily at the boundaries of parallel regions, so this number can greatly affect our results. For example, in the GEM benchmark, the adaptive schedulers are identical to the static scheduler because the training pass is the only pass in the application. Conversely, CG performs many short passes, which allows CoreTSAR to adjust scheduling decisions but incurs high scheduler overhead and data copy costs.

The table also shows a wide range of performance ratios. Values range from a $10\times$ slowdown to a $113\times$ speedup going from eight CPU cores to one GPU. Running GEM on only one GPU finishes the problem more than $10\times$ faster than on eight server class Intel CPU cores. CORR and COVAR also show extreme suitability, largely due to the static schedule employed in the CPU tests. Because the workloads are imbalanced, each CPU core performs a different amount of work. The GPU test, because of the load-balancing effect of over-provisioning work-groups on GPUs, handles this variation better. If we use the OpenMP dynamic schedule, COVAR runs in approximately 150 seconds, $10\times$ faster than the static performance. Alternatively, GRAMSCHMIDT and Helmholtz are not suited to GPU computation according to these results. Generally, the suitabilities match our expectations, with the exception of CG. Our previous work, and that of others, has found that CG is suitable for GPUs. Some of our experiments on other platforms showed a ratio of approximately 0.55 on one GPU. Here, the GPU version takes more than $5\times$ longer than the CPU version. This is due to the high cost of data redistribution across GPUs each iteration. We leave optimization of CG to future work.

Benchmark	Passes	Time/ pass	CPU time	GPU time	Speedup on 1GPU
CG	1900	0.045	16.31	92.37	0.17
GEM	1	5.336	71.05	5.65	12.59
Helmholtz	50	0.138	1.18	7.22	0.16
kmeans	7	0.583	5.70	4.33	1.32
ATAX	10	0.646	32.23	6.60	4.88
BICG	10	0.822	21.86	8.78	2.49
CORR	10	0.162	157.73	1.64	96.07
COVAR	10	1.328	1558.30	13.80	112.90
FDTD2D	5000	0.000	0.99	1.23	0.80
GEMM	10	1.262	301.34	3.04	99.18
GESUMMV	10	1.902	2.10	20.38	0.10
GRAMSCHMIDT	10240	0.004	4.21	40.38	0.10
MVT	10	0.058	1.62	0.60	2.72
SYR2K	10	1.461	14.39	15.53	0.93
SYRK	10	0.769	7.86	8.18	0.96
THREEDCONV	10	1.031	5.77	10.95	0.53
THREEMM	30	0.284	126.03	3.78	33.35
TWODCONV	10	0.607	2.86	6.46	0.44
TWOMM	10	1.445	204.66	6.32	32.41

TABLE 2: Benchmark characteristics, times in seconds, time/pass for static schedule with CPUs and one GPU.

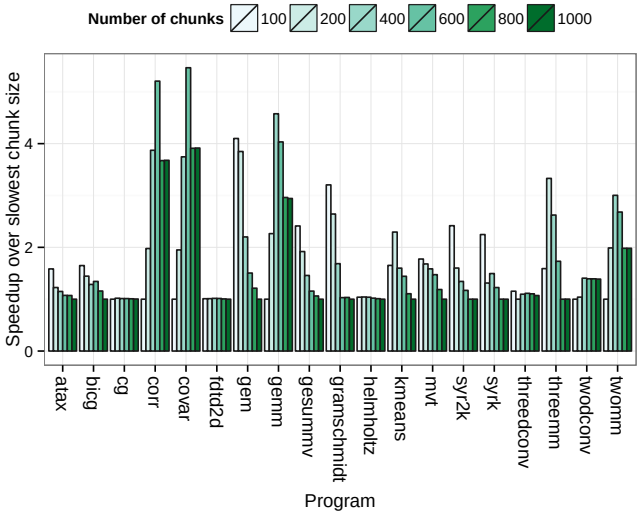

Fig. 8: Performance across chunk sizes for each benchmark with the basic Chunk scheduler

5.2 Input Parameters

As mentioned above, we use the default values for our tests unless otherwise specified. However, chunk size does not have an obvious default. Figure 8 illustrates the performance for the basic chunk scheduler across chunk sizes for each benchmark using one GPU. We do not report chunk sizes in terms of absolute iterations, which has little meaning across benchmarks. Instead, we compare by the number of chunks into which the region is partitioned. The performance of some applications varies little based on chunk size. Others, such as CORR and COVAR, have a range of as much as $3\times$. These ranges shift or even reverse in some cases as the number of GPUs or scheduler changes, creating even more variability. Due to the sensitivity to this parameter, all subsequent results for chunk-based schedulers use the best chunk size for that benchmark, scheduler, and GPU count combination.

5.3 CoreTSAR Performance

We begin with an evaluation of the overall speedup achieved for benchmarks across schedulers and GPU counts on escaflowne, as Figure 9 depicts. All plots are based on the speedup over a chunk-based CPU schedule equivalent to OpenMP's dynamic schedule across the 8 CPU cores. We can group these results roughly into three groups of behavior: those that scale to all four GPUs; those that benefit from GPUs but do not scale to more than one; and GPU-averse applications.

5.3.1 GPU Amenable Applications

Eleven benchmarks scale to four GPUs on escaflowne, resulting in between $3.5\times$ and nearly $200\times$ speedup. First GEM, GEMM, kmeans, SYRK, SYR2K, TWOMM and THREEMM scale nearly linearly from one to four GPUs, missing linear only because of the use of one CPU core for the addition of each GPU. Slightly off of linear are CORR and COVAR, which gain performance at approximately one quarter of that rate, but consistently up to all four GPUs. Also in this group are ATAX, BICG, and MVT, which clearly taper off after two GPUs, since these benchmarks do not have enough work available at this problem size, to occupy all four GPUs fully. Further, we cannot increase the problem size without overflowing the GPU memory due to the way CoreTSAR's memory model currently handles mappings. In another peculiarity of these three benchmarks, the chunk scheduler performs almost identically to the CPUs. While all three reap significant performance benefits when run on GPUs, they are the only benchmarks that use column-wise partial copies. The overhead of column-wise copies for each chunk apparently causes the runtime to deactivate all GPU threads for the basic chunk scheduler.

In terms of individual benchmark behavior, GEMM achieves the most speedup, which occurs with the static GPU-only configuration. While this schedule is not an adaptive, it is still facilitated by CoreTSAR, and for extremely GPU suitable applications can outperform the adaptive schedules. The CORR and COVAR benchmarks superficially behave similarly, but for a different reason. In their imbalanced workloads, each iteration i does $n-i$ units of work. Thus, they violate the assumption of the adaptive schedulers that the average work per iteration is constant. We expected one or more of the chunk schedulers would perform best in this scenario, but both CORR and COVAR are highly sensitive to overhead, and cannot tolerate the additional launches and copies of the chunk schedulers. Thus, the static schedulers (GPU and static) perform best in most cases. In the four GPU case for each, however, the split scheduler surges ahead. Split *stops using the CPU cores* and schedules across the GPUs in the three and four GPU cases. Our linear program does not handle varying time per iteration with heterogeneous hardware, but given relatively homogeneous hardware it handles the heterogeneous iterations much better. Using only GPUs, no CPU cores, with the Adaptive schedule achieves a further 10-20% performance improvement over the next best schedule in each case for CORR and COVAR.

Also unexpectedly, kmeans performs best with the basic chunk scheduler. With a precisely selected chunk value kmeans does quite well but, as Figure 8 shows, its performance varies by as much as 50% across chunk sizes we tested. The adaptive schedulers are more robust in that they do not require users to search the input space in order to find a reasonable initial parameter.

Overall CoreTSAR scales well to at least four GPUs without loop body or memory layout changes for GPU-amenable applications. Further, each scheduler is stronger for certain tasks than others, and the adaptive scheduler is the best overall choice, even with the best chunk sizes for chunk schedulers. It remains stable, and within approximately 10% of optimal for all amenable benchmarks with homogeneous iterations. For heterogeneous iterations, static and chunk are better options.

5.3.2 GPU-Averse Applications

These are applications that *do not* run well on GPUs. Some are so sensitive to it that running any part of the job on a GPU causes slowdown. These are included to evaluate CoreTSAR's response to regions that should not use GPUs, or to running normally amenable applications on a system where the accelerator is particularly slow. While Jacobi solvers in general, and Helmholtz solvers in particular, are not GPU averse as a class, the implementation of Helmholtz that we evaluate is. Our Helmholtz is a generic CPU OpenMP version that runs correctly but slowly when compiled for the GPU. It never performs better by using a GPU for any work. This category also includes three PolyBench/GPU benchmarks (FDTD2D, GESUMMV and GRAMSCHMIDT). Each runs slower on a GPU than on one CPU or runs many passes, accentuating copy overhead.

In each case, schedulers that run more often, and thus convert the GPU threads to CPU threads faster, incur less performance loss. For the same reason, GPU-averse benchmarks that run many small passes perform better. For example, GESUMMV suffers more than the others by running 10 passes rather than 50 or thousands. For each of these benchmarks, the ability to convert GPU control threads to CPU threads is crucial. Without GPU backoff support, the total runtime of Helmholtz more than doubles for both adaptive and chunk based schedules, and as much as triples for the split schedule.

Three other benchmarks (CG, THREEDCONV, and TWODCONV) fall into this category, but only marginally. Each can benefit from the first GPU. GPUs complete iterations faster than CPUs for these benchmarks, but they only have enough work to saturate a single GPU, or face increasing data transfer overhead as more are added. CG passes through the region enough times (1,900) that all but one GPU are converted to CPU threads very early in the computation,

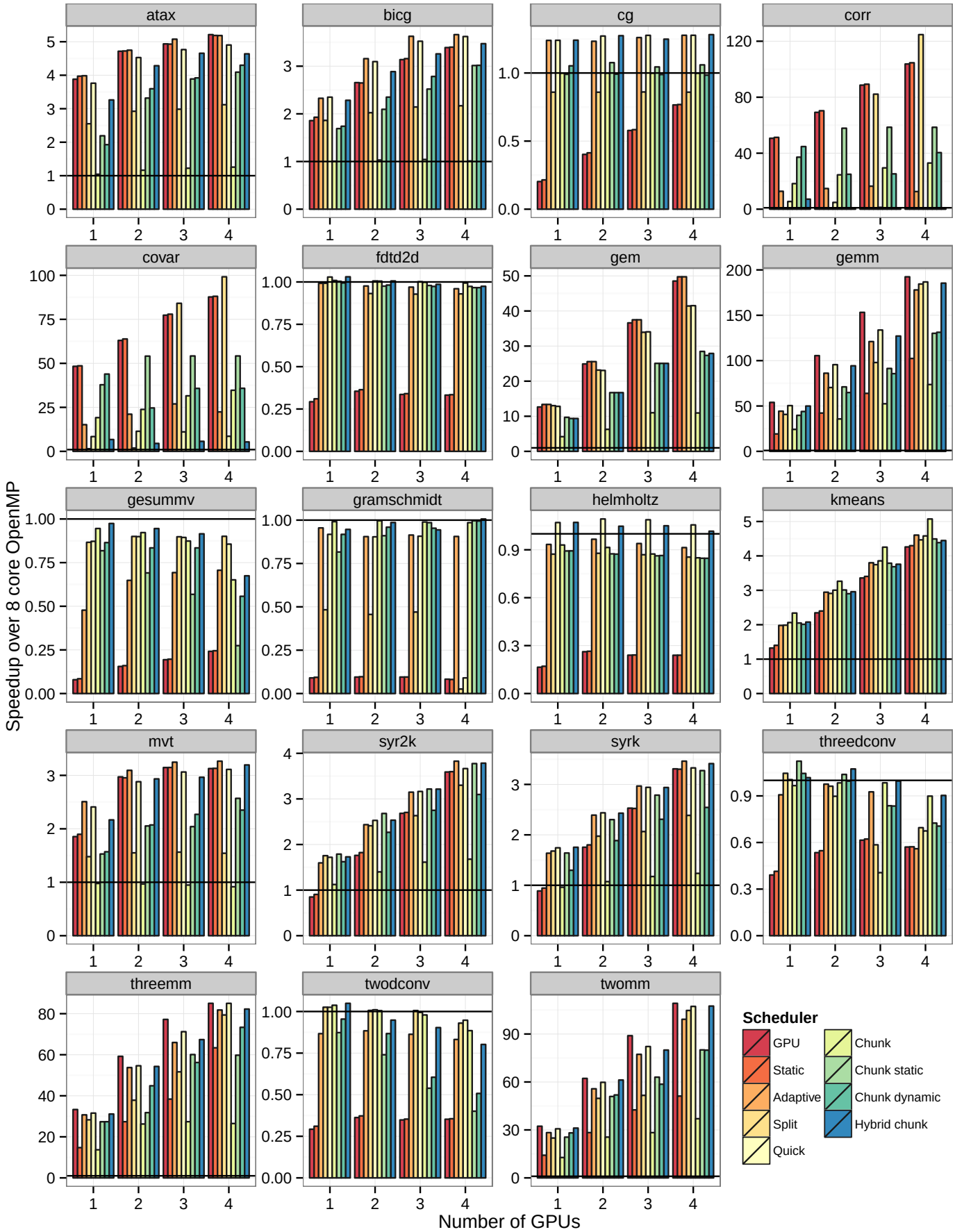

Fig. 9: Performance across schedulers and number of GPUs for all benchmarks, normalized to CPU OpenMP across 8 cores.

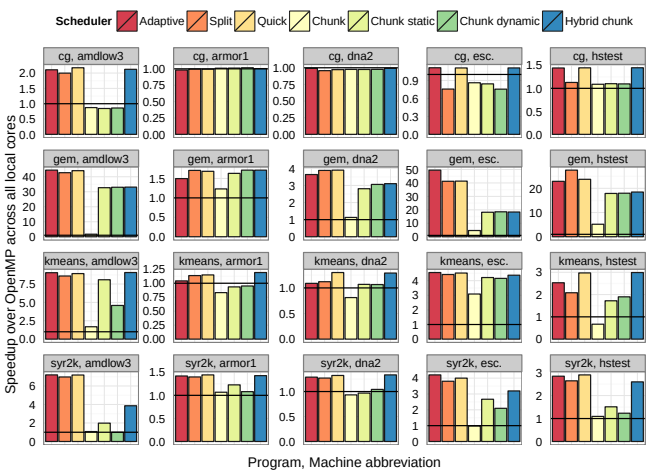

Fig. 10: CoreTSAR speedup across systems

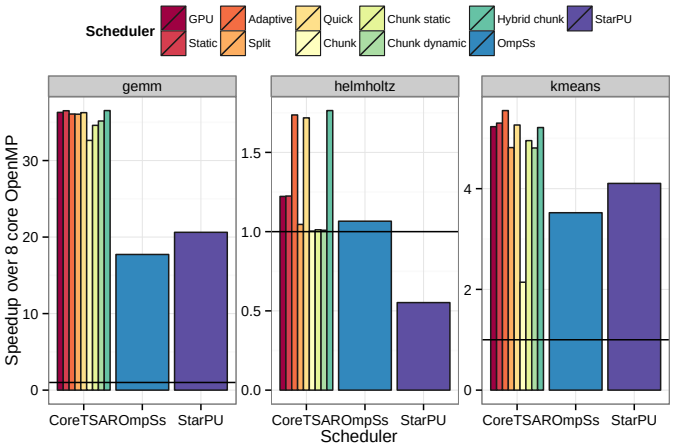

Fig. 11: Comparison of CoreTSAR with OmpSs and StarPU.

so it achieves roughly constant performance from one to four GPUs. The convolution codes do not run long enough to hide the overhead of extra GPUs enabled in the first few passes and show degrading performance.

5.4 Adaptation Across Machines

We now evaluate CoreTSAR's performance across several disparate systems. All systems run the same OS image and execute identical binaries for all tests. Table 1 lists the hardware in each system in detail. 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 GPU.

As some of our benchmarks require more memory than the smaller GPUs posses, we selected a representative subset (CG, GEM, kmeans, and SYR2K) with problem sizes that fit onto all evaluated GPUs. Figure 10 shows results for these benchmarks across all five test platforms. 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, partly due to the extreme imbalance between its Intel Celeron processor and NVIDIA C2050 GPU. Even CG shows material speedups on amdlow3, as much as $2\times$. More importantly, even though speedup and overall performance shift across the various systems for each benchmark, the distribution of performance by scheduler is similar. Thus, the right CoreTSAR scheduling algorithm is more related to the application than the hardware. Allowing the scheduler to be determined once per region, rather than once per machine. Further, these results show that the default adaptive scheduler is effective across hardware configurations, with only GEM as an issue, as a result of its single iteration. 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 best.

5.5 Comparison with Blocked Task Schedulers

In order to compare CoreTSAR's scheduling with a state of the art heterogeneous task scheduler, we employ those designed to support blocked task models. Specifically we port three benchmarks (GEMM, kmeans, and Helmholtz) to two freely available implementations of this type of model, OmpSs and StarPU. As with Accelerated OpenMP and CoreTSAR, we use the most straightforward port possible, transforming only the loop regions that CoreTSAR targets. For example, Figure 1 lists a literal transcription of the GEMM implementation on OmpSs, calling functions defined in Figure 1.

In order to provide an accurate comparison, the CoreTSAR codes evaluated here use the CUDA and C functions created for OmpSs and StarPU rather than using Accelerated OpenMP. In fact these functions were compiled into a single object file with nvcc that was then linked with the CoreTSAR, OmpSs and StarPU scheduling code, thus all three are scheduling over *identical* compute kernels. The OmpSs version was run with the versioning-stack scheduler, to support alternative implementations, as well as flags to allow prefetching and overlapping of data transfers for benchmarks where these offered speedup (slowdown was observed in one case). The StarPU implementations used the "dmda" scheduler with the history-based performance model, trained on at least ten runs before results were collected.

GEMM and Helmholtz run each row of the main outer loop as an individual task. The outer loop for kmeans is fine grained, so we block it into chunks of 1000 iterations for OmpSs and StarPU, and also use 1000 iteration chunks as the default for CoreTSAR's *chunk* schedulers although we allow it to adapt at runtime where capable. Each only copies the data necessary for a given task. For example, we only request the three rows necessary for a given Helmholtz row. We also disable CoreTSAR's persistent memory support, since OmpSs does not provide an equivalent feature, though StarPU does.

Figure 11 presents the speedup results, calculated as

speedup over all 8 CPU cores with the OpenMP dynamic schedule, with OmpSs and StarPU on the far right. While unrelated to the performance comparison, Helmholts shows a performance benefit using GPUs in this case. In truth, the CPU version compiled with gcc is significantly slower ($9\times$) than the version evaluated earlier, while the CUDA and OpenACC versions perform similarly.

Each of StarPU and OmpSs are block schedulers, operating much like our Chunk scheduler, and so we expect that they would perform similarly. The expectation holds holds for Helmholtz, wherein OmpSs performs almost identically to CoreTSAR's Chunk scheduler with StarPU trailing by roughly 50%. In kmeans and GEMM each performs quite differently, with OmpSs and StarPU outperforming Chunk on kmeans and being heavily outperformed by it in GEMM.

While the computation and data transfers are nearly identical between the schedulers, the performance of CoreTSAR using one of the granularity adapting schedulers is consistently higher due to reduced overheads. Since CoreTSAR never explicitly creates the individual tasks, it never pays the cost to allocate or to initialize them, only paying for the aggregate tasks it runs. This benefit is especially noticeable in GEMM where CoreTSAR is $3\times$ faster than OmpSs and $2\times$ faster than StarPU scheduling the same work. Given the ability to adapt task granularity at runtime, all three would yield similar performance. It may be worthwhile to consider adding CoreTSAR, or a similar task-splitting design, to each of OmpSs and StarPU to reduce overhead for this type of computation.

6 Related Work

With the proliferation of GPUs and other computational accelerators, several programming models and task schedulers have been proposed specifically for these environments. In addition to the blocked task schedulers, StarPU [6], [5] and OmpSs [14], [11], which we discussed in Section 2, other designs have been proposed. Two major factors distinguish our work from these schedulers. First, they schedule at the granularity of discrete tasks, which in each case is defined by a function call, and forces the user to select the appropriate granularity of work even within a group of related tasks. Second, they require that the task functions are implemented in terms of blocks of data. These blocks generally need to be contiguous chunks of data, for common cases StarPU offers "filters" as a convenient way to divide data into equal size chunks, and recent work on OmpSS [10] has added support for potentially non-contiguous rectangular regions to be passed to tasks. With CoreTSAR, a task granularity may optionally be used, but is not required and often does not result in the best division of work. As to data blocks or transformation of tasks to operate on them, CoreTSAR handles unblocked accelerated OpenMP code, preserving the semantics of the original parallel region.

More relevant are the approaches taken by Qilin [16] and the scheduling framework presented by Ravi et al. [17], [18]. These authors present novel heterogeneous programming APIs that support adaptive scheduling between CPUs and a GPU. The Qilin API is in the form of a C++ template library that operates on special array structures and allows runtime generation of CPU and GPU code. Ravi et al.'s work generates CPU and GPU code from generalized reduction specifications. Both require reimplementation of existing codes in the associated model, constrain the adaptive scheduling approach to that used by the respective system, and target only one GPU. Qilin uses an adaptive approach similar to the one that we used in our previous work on Splitter [19] to support one GPU. However, they calculate the division in a training pass and simply reuse it in latter runs. The framework by Ravi et al. uses a chunk-based mechanism, with an option to combine chunks for scheduling on the GPU much in the way our dynamic chunk schedule does. Alternatively, CoreTSAR handles memory movement and adaptive scheduling of work while preserving existing code inside the region. Further it supports a range of scheduling mechanisms allowing a user to select an adaptive or chunk-based approach on a per region basis, as well as supporting an arbitrary number of arbitrarily capable GPUs and CPUs.

Our adaptive scheduling policies are also highly related to the approach taken by Ayguadé et al. [7] in looking for an alternative to the schedule clause in OpenMP. Rather than employing dynamic, or chunk style scheduling, they proposed the use of a learning scheme to do a static split. Their specific prediction methods and targets were different from ours, but their assertion that the adaptive policies sometimes benefited CPUs as well may be another reason to incorporate something like our adaptive schedules into OpenMP.

7 Conclusion

We have presented the design and implementation of CoreTSAR (Task-Size Adapting Runtime). We make four primary contributions: the design of our scheduler for adaptive scheduling across arbitrary numbers of heterogeneous devices; an implementation and optimization of that design; the design and evaluation of seven adaptive scheduling policies; and our evaluation across four scientific codes, 15 benchmark kernels and a side-by-side comparison with OmpSs and StarPU. We achieve speedups as high as $3.74\times$ over the best performance that uses all cores and a single GPU. When compared to the original CPU performance on 8 cores, we achieve as much as $180\times$ for one benchmark. Further, we present an extension to our memory management system that transparently aligns matrices during mapping, improving performance in some cases by as much as $2.5\times$. These results clearly demonstrate the benefits to be gained from runtime adaptation of task sizes and 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 more comprehensive memory association support. Starting from the padding transformation we presented here, we will pursue more automatic transformations to allow loop body code to reference different memory layouts transparently. In the main scheduler, CoreTSAR could automatically detect NUMA issues, and the association of GPUs 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] CUDA programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2007.

[2] The OpenCL Specification. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, Nov. 2012.

[3] OpenACC 2.0 Application Programming Interface Specification. http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf, June 2013.

[4] R. Anandakrishnan, T. R. W. Scogland, A. T. Fenley, J. C. Gordon, W. Feng, and A. 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, June 2010.

[5] C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report RR-7240, Laboratoire Bordelais de Recherche en Informatique - LaBRI, RUNTIME - INRIA Bordeaux - Sud-Ouest, Mar. 2010.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *International Euro-Par Conference on Parallel Processing*. Springer-Verlag, Aug. 2009.

[7] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the Schedule Clause Really Necessary in OpenMP? In *Workshop on OpenMP Applications and Tools*. Springer-Verlag, June 2003.

[8] D. H. 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 Summary and Preliminary Results. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing)*, pages 158–165, 1991.

[9] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski. OpenMP for Accelerators. In *Lecture Notes in Computer Science: OpenMP in the Petascale Era*, pages 108–121. Springer Berlin Heidelberg.

[10] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *ACM International Conference on Supercomputing*. ACM, June 2013.

[11] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. *International Parallel and Distributed Processing Symposium*, pages 557–568, 2012.

[12] M. Daga, T. R. W. Scogland, and W. Feng. Architecture-Aware Mapping and Optimization on a 1600-Core GPU. In *International Conference on Parallel and Distributed Systems*, pages 316–323, Tainan, Taiwan, 2011. IEEE Computer Society.

[13] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A Hybrid Multi-Core Parallel Programming Environment. In *GPGPU 2007: Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[14] A. Duran, E. Ayguadé, 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.

[15] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-Tuning a High-Level Language Targeted to GPU Codes. *Innovative Parallel Computing*, pages 1–10, 2012.

[16] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Dec. 2009.

[17] V. T. Ravi and G. Agrawal. A Dynamic Scheduling Framework for Emerging Heterogeneous Systems. In *International Conference on High Performance Computing (HiPC)*, pages 1–10, 2011.

[18] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *ACM International Conference on Supercomputing*. ACM, June 2010.

[19] T. R. W. Scogland, B. Rountree, W. Feng, and B. R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *International Parallel and Distributed Processing Symposium*, pages 144–155. IEEE Computer Society, May 2012.

[20] T. R. W. Scogland, B. Rountree, W. Feng, and B. R. de Supinski. CoreTSAR: Adaptive Worksharing for Heterogeneous Systems. In *International Supercomputing Conference*, Leipzig, June 2014.

[21] M. Wolfe. Implementing the PGI Accelerator Model. In *Annual Workshop on General Purpose Processing with Graphics Processing Units*. ACM, Mar. 2010.

Thomas R.W. Scogland is a Ph.D. candidate in the Department of Computer Science at Virginia Tech (VT), where he is a member of the Synergy Lab. His primary research area is in high-performance computing, with a focus on scheduling in heterogeneous systems and green computing. He received a B.S. degree in Computer Science from Purdue University in 2007 a NDSEG Graduate Fellowship in 2009 and his M.S. from Virginia Tech in 2012. He is a student member of the ACM and IEEE.

Wu-chun Feng is the Elizabeth & James E. Turner Fellow and Professor of Computer Science, Electrical & Computer Engineering, and Health Sciences at Virginia Tech (VT), where he also directs the Synergy Laboratory. Grassroots projects that he leads include the Green500, Supercomputing in Small Spaces, and MyVICE. His research interests encompass a broad range of topics in efficient parallel and distributed computing, including high-performance computing and networking, energy-efficient (or green) supercomputing, heterogeneous and GPU computing, cloud and grid computing, MOON computing, bioinformatics, and computer science pedagogy for K-12.

He received B.S. degrees in Computer Engineering and Music (Honors) and M.S. degree in Computer Engineering at Penn State University in 1988 and 1990, respectively. He then earned his Ph.D. in Computer Science at the University of Illinois at Urbana-Champaign in 1996. He is a Distinguished Member of ACM and Senior Member of IEEE.

Barry Rountree received a BA in Theater Arts and Drama from the Ohio University Honors Tutorial College, an MS in Network and System Administration from Florida State University, and a PhD in Computer Science from the University of Arizona. His current work focuses on optimizing massively parallel performance under hard electrical power bounds, particularly with regard to exascale system designs.

Bronis R. de Supinski is the Chief Technology Officer (CTO) for Livermore Computing (LC) at Lawrence Livermore National Laboratory (LLNL). In this role, he is responsible for formulating LLNL's large-scale computing strategy and overseeing its implementation. His research has explored topics including programming models, algorithms, performance, code correctness and resilience for future large scale systems. He currently continues his interests in these topics, particularly programming models, and serves as the Chair of the OpenMP Language Committee. Throughout his career, Bronis has won several awards, including the prestigious Gordon Bell Prize in 2005 and 2006, as well as an R&D 100 for his leadership of a team that developed a novel scalable debugging tool. He is a member of the ACM and the IEEE Computer Society.