CoreTSAR: Adaptive Worksharing for Heterogeneous Systems *

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

* Department of Computer Science, Virginia Tech, Blacksburg, VA 24060 USA
† Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551 USA

Abstract. The popularity of heterogeneous computing continues to increase rapidly due to the high peak performance, favorable energy efficiency, and comparatively low cost of accelerators. However, heterogeneous programming models still lack the flexibility of their CPU-only counterparts. Accelerated OpenMP models, including OpenMP 4.0 and OpenACC, ease the migration of code from CPUs to GPUs but lack much of OpenMP's flexibility: OpenMP applications can run on any number of CPUs without extra user effort, but GPU implementations do not offer similar adaptive worksharing across GPUs in a node, nor do they employ a mix of CPUs and GPUs. To address these shortcomings, we present CoreTSAR, our library for scheduling cores via a task-size adapting runtime system by supporting worksharing of loop nests across arbitrary heterogeneous resources. Beyond scheduling the computational load across devices, CoreTSAR includes a memory-management system that operates based on task association, enabling the runtime to dynamically manage memory movement and task granularity. Our evaluation shows that CoreTSAR can provide nearly linear scaling to four GPUs and all cores in a node *without* modifying the code within the parallel region. Furthermore, CoreTSAR provides portable performance across a variety of system configurations.

1 Introduction

Heterogeneity is becoming more prevalent in all areas of computing, from supercomputers to cell phones. The increasing prevalence of GPUs and computational coprocessors has spawned a vast array of tools and programming models over the past several years, but the majority of codes remain CPU-only. Of these new models, Accelerated OpenMP holds the promise of increasing adoption,

* 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)

especially in high-performance computing (HPC). Accelerated OpenMP models, including OpenACC [6] and OpenMP 4.0 [5], extend the classic OpenMP preprocessor directives with the capability to offload data-parallel regions to accelerators [1].

Assuming the ease of use is sufficient to convince developers to embrace accelerators, one major issue still remains. Accelerated OpenMP only offloads computation to a single device (e.g., GPU), leaving the CPU cores or other accelerators in the system idle. There are mechanisms that allow users to manually partition their work across devices but *no* direct support for cross-device worksharing. As a result, users must *manually* partition their job, manage memory coherence, and load balance across devices. In the best case, a user may use a task scheduling library such as OmpSs or StarPU to handle load balancing and memory transfers. Even then, however, the user must manually divide their workload into explicit tasks.

Our solution, CoreTSAR (short for Core Task-Size Adapting Runtime), provides a cross-device worksharing construct for Accelerated OpenMP. The key extension to make this possible is a memory-management approach that allows a user to specify the association between their computation and data, that frees CoreTSAR to handle coherence and task granularity automatically. CoreTSAR includes a proposed set of extensions to Accelerated OpenMP, the design of a cross-device memory manager, scheduling policies to support the clauses, and a real-world implementation of the system. Together, they provide adaptive worksharing of parallel loop regions across an arbitrary number of devices with an arbitrary number of distinct address spaces. In this paper, we make the following contributions:

- Accelerated OpenMP extensions to adaptively workshare parallel regions across an arbitrary number of arbitrarily heterogeneous devices
- The design and implementation of a task-associative, memory-management interface, thus allowing CoreTSAR to adapt task granularity at runtime
- A library and C API implementation of our scheduler and memory manager
- An evaluation demonstrating that CoreTSAR can improve performance over existing task-management approaches.

The remainder of the paper is composed as follows. Section 2 provides background and motivation. Section 3 describes the design and implementation of CoreTSAR, including our task-management concept, scheduling mechanisms, and memory management. Section 4 presents our evaluation. Finally, we finish with related work in Section 5 and conclusion in Section 6.

2 Background and Motivation

Models for heterogeneous computing generally fall into three categories: offload models; block and grid models; and task block models. Domain specific languages are also available, but we focus on general purpose options. Each of the

[1] Since our evaluation is based on GPU accelerators, the terms “accelerator” and “GPU” are used interchangeably throughout.

```
void kmeans_it(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl) {
//OpenMP
#pragma omp parallel for
//Accelerated OpenMP
#pragma acc parallel for copyout(m[0:no])   \
        copyin(fc[0:nco*ncl],fo[0:no*nco])
//Accelerated OpenMP + extension
#pragma acc parallel for pcopy(m[1:no])\
            copyin(fc[0:nco*ncl]) copyin(fo[no*nco])\
            hetero(1, all, adaptive, default, 10)
  for (i=0; i<no; i++) {
    m[i] = findc(no,ncl,nco,fo,fc,i);
  }
}
```

```
__global__ void
void it_gpu(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl,
               size_t start, size_t end) {
    uint i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < end-start) {
        m[i] = findcu(no,ncl,nco,fo,fc,i+start);
    }
}
void kmeans_it(int *m, float *fo, float *fc,
        size_t no, size_t nco, size_t ncl,
        size_t start, size_t end) {
    dim3 dB, dG;
    dB.x = 64;
    dB.y = dB.z = 1;
    dG.x = ((end-start)/dB.x)+1;
    dG.y = dG.z = 1;
    cudaMalloc(&cm, no);
    cudaMalloc(&cfo, no*nco);
    cudaMalloc(&cfc, no*ncl);
    cudaMemcpy(cm, m, no,cudaMemcpyHostToDevice);
    cudaMemcpy(cfo,fo,no*nco,cudaMemcpyHostToDevice);
    cudaMemcpy(cfc,fc,no*ncl,cudaMemcpyHostToDevice);
    it_gpu<<<dG,dB>>>(
            cm, cfo, cfc, no, nco, ncl, 0, no);
    cudaMemcpy(m,cm,no,cudaMemcpyDeviceToHost);
}
```

```
#pragma omp target device(smp,cuda)
void it_helper(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl,
               size_t start, size_t end);

#pragma omp target device(cuda)
__global__ void
void it_gpu(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl,
               size_t start, size_t end);
/* implementation same as CUDA, see left */

#pragma omp target device(smp)
#pragma omp task input([no*nco]fo, [ncl*nco]fc)\
                 inout([end-start]m)
void it_helper(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl,
               size_t start, size_t end){
    for (int i=0; i<end-start; i++) {
        m[i] = findc(no,ncl,nco,fo,fc,i+start);
    }
}

#pragma omp target device(cuda) copy_deps\
                    implements(it_helper)
#pragma omp task input([no*nco]fo, [ncl*nco]fc)\
                 inout([end-start]m)
void it_cuhelper(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl,
               size_t start, size_t end) {
    dim3 dB, dG; dB.x = 64; dB.y = dB.z = 1;
    dG.x = ((end-start)/dB.x)+1; dG.y = dG.z = 1;
    it_gpu<<<dG,dB>>>(cm,cfo,cfc,no,nco,ncl,start,end);
}

void kmeans_it(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl) {
  const int BS = 1000;
  for (i=0; i<no-BS; i+=BS) {
    it_helper(&(m[i]), fo, fc, no, nco, ncl, i, i+BS);
  }
  it_helper(&(m[i]), fo, fc, no, nco, ncl, i, no);
#pragma omp taskwait
}
```

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

approaches has strengths and weaknesses in terms of programmability, performance, and flexibility, which we will discuss here in terms of the three example implementations of the kmeans clustering algorithm presented in Figure 1.

Offload models execute annotated parallel regions on a GPU. Examples include Accelerated OpenMP (OpenMP 4.0 [12], OpenACC [6], Intel OpenMP for MIC and Cray's accelerated OpenMP [5]) and C++ AMP. Generally they require the least change from the original serial code. The upper left of Figure 1, despite its size, actually contains four different versions. If none of the pragmas are honored, then the loop executes in serial. If the OpenMP pragma on line 4 is honored, the loop is work-shared across CPU cores. To target a GPU, the OpenACC directive on lines 6 and 7 copies both the `fc` and `fo` arrays to the GPU and `m` to and from it. This directive *moves* the loop from the CPU to a GPU: when the `parallel for` directive is applied, no CPU cores (or other GPUs) participate in the loop. One must manually split the loop and data to target multiple devices.

Block and grid models represent data-parallel kernels as a grid of blocks of threads, where synchronization in a kernel is only possible within the blocks. OpenCL [11] and NVIDIA's CUDA both fall into this group. They are some of the most efficient and most used GPU programming models, offering low-level

control at the cost of verbosity. The bottom left code block in Figure 1 presents a basic CUDA port of the OpenMP kmeans code. Even with the larger number of management lines, the CUDA version does not free the GPU memory, check error codes, or perform GPU selection and initialization (so the conservative default queue is used). As with the OpenMP version, this code only uses one GPU.

Task block models do not actually specify a programming model for accelerators but rather provide task scheduling across heterogeneous resources. This group includes StarPU [2] and OmpSs [9]. The right side of Figure 1 presents an OmpSs implementation of the kmeans loop. The first four pragmas specify that the `it_helper` function is an OmpSs task that depends on `fo`, `fc` and a slice of `m`. The `implements` clause on line 20 informs OmpSs that `it_cudahelper`, which uses the CUDA implementation at the bottom left minus the memory movement, is the CUDA implementation of `it_helper`. In `kmeans_it()` the tasks are enqueued by calling the helper function with appropriately offset pointers. OmpSs automatically distributes this work across all CPU cores and GPUs. This flexibility provides performance portability, at a cost. The user must partition their work into tasks appropriate for either an entire GPU or a single CPU core simultaneously.

These models are all useful under the right circumstances. Optimally we would have the fine-grained control of block and grid models, the simplicity and programmability of offload models and the runtime flexibility and performance portability of task block models. We propose syntax and offer a runtime implementation of CoreTSAR(Task-Size Adapting Runtime), a system that can be used to add flexibility to offload and block and grid models, and also to extend task-block models with adaptive task granularity.

3 Design and Implementation

Our primary goal with CoreTSAR is to create a runtime to support worksharing across devices for use with Accelerated OpenMP. This goal imposes certain design constraints. Most importantly, it must not require any changes, even for memory movement, to the loop body beyond those for Accelerated OpenMP. For example, no pragmas or API calls may be inserted into the loop, nor memory access patterns be changed, as task scheduling systems often require. All information necessary for CoreTSAR to provide the correct data for *any* range of iterations to *any* device's memory space must be provided in the directive outside the loop. Further, we must preserve data consistency outside the region: main memory must hold the same values when the loop exits as it would have with Accelerated OpenMP.

Figure 2 depicts the syntax of our extension. The `hetero()` clause specifies how to treat the region. Specifically, whether to schedule it, which scheduler to use, which devices to schedule it across and, if desired, initial values for the work-split ratio. The `pcopy()` clause specifies the association between iterations and data. It is similar to a copy clause in OpenACC, except that it only applies to

loop directives and only copies the specific memory elements that are associated with the range of iterations assigned to a given device. We support current OpenACC `copyin` and `alloc` data clauses in order to replicate the entire input or allocation, `copyout` is unsupported. The third pragma in the top left block of Figure 1 uses our extension to divide the loop across all devices using the adaptive scheduler, copying all of `fc` and `fo` as input, but only the necessary elements of `m` as output.

```
//items in {} are optional
#pragma acc parallel for hetero(<condition>{,<devices>{,<scheduler>{,<ratio>{,<div>}}}})\
                         pcopy{in/out}(<var>[<cond>:<num>{:<boundary>}]) persist(<var>)
#pragma acc depersist(<var>)
```

Fig. 2: Our proposed extension

The remainder of this section discusses the design and implementation of a runtime system to support the proposed extension. Our design has two main components: the scheduling and task inference portion; and the memory specification and management portion.

3.1 Scheduling and Tasks

As our previous work in heterogeneous task scheduling showed [16], adaptive worksharing by predicting performance, rather than employing work-queues or discrete tasks, can significantly reduce overhead in heterogeneous scheduling. Overhead is reduced by assigning work *statically* within a region, and re-balancing on the next entry into the region, thus reducing the number of tasks to manage and launch. The significant downside to our previous approach however was that the model only supported the modeling of performance across two devices, preventing it from targeting systems with higher levels of heterogeneity. That work also proposed a solution as future work, using an integer-based optimization approach.

I = total iterations available
i_j = iterations for compute unit j
f_j = fraction of work for compute unit j
p_j = average time/iter. for compute unit j (1)
n = number of compute devices
$t_j^{+/-}$ = time over (or under) equal

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

$$\sum_{j=0}^{n} f_j = 1 \tag{3}$$

$$f_2 * p_2 - f_1 * p_1 = t_1^+ - t_1^- \tag{4}$$

$$f_3 * p_3 - f_1 * p_1 = t_2^+ - t_2^- \tag{5}$$

$$\vdots$$

$$f_n * p_n - f_1 * p_1 = t_{n-1}^+ - t_{n-1}^- \tag{6}$$

Fig. 3: Linear program variables, objective and constraints

The original integer-based optimization directly computed the number of iterations to assign to each device, but incurs too much overhead to be used for

runtime scheduling, on the order of seconds for less than ten devices. Figure 3 presents an alternative approach designed for CoreTSAR, optimized for use at runtime. Our new linear optimization minimizes the total deviation between the predicted runtimes for all devices to run their assigned work, and expresses output as the fraction of total iterations that should be assigned to a given device. Switching to fractional output removes the costly refinement to integer output and increases numerical stability. The downside is that fractional output allows up to n iterations to go unassigned, any iterations left-over this way are assigned in round-robin fashion to devices in descending order of performance.

This design assumes that trading one CPU core to control a GPU will improve performance. However, some applications benefit more from the CPU core. Thus, CoreTSAR also includes a heuristic allowing it to convert a GPU controlling thread back to a CPU thread in such cases. After each pass the time per iteration (TPI) of each GPU is compared against that of the slowest CPU core. If a GPU's TPI falls below that value for two consecutive iterations, the GPU thread releases the GPU and continues as a CPU thread.

3.2 Static Scheduling

On the first entry into a region, our static schedule uses the linear program to assign iterations, then reuses that result for all subsequent passes. To increase portability, we compute default relative times per iteration at runtime rather than using a precomputed static value (the user can also specify a value). Our default assumes that one instruction cycle on a GPU core, or SIMD lane on a multiprocessor, takes the same time as one cycle on a single SIMD lane of a CPU core. We thus compute the time per iteration for each GPU as $p_g = \frac{1}{m/s}$ and for CPU cores as $1 - p_g$ (where m is the number of cores on the GPU and s the SIMD width of a CPU core; in the case of multiple GPUs, all devices are normalized to the largest). 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, but leave these for future work.

3.3 Adaptive Scheduling

Our adaptive schedules (*Adaptive*, *Split* and *Quick*) use the static schedule for the first pass. We then use the time that each device takes to complete its iterations in the preceding pass, kept as a weighted average over up to five passes, as input to our linear program for the next pass. All recurring overheads required to execute an iteration on a particular device are included in that time (but not one-time overheads such as the copying of persistent data). Thus, we incorporate data-movement and launch overheads into the cost of each 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 breaks each region into several evenly split sub-regions based on the `div` input.

Each sub-region is then treated individually and scheduled as Adaptive would a full region, providing faster load balancing at the cost of increased overhead. The *Quick* schedule balances between the *Split* and *Adaptive* schedules by executing a small sub-region for its first training phase, similar to the way *Split* starts. 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.

3.4 Memory Management

Efficient and minimal data movement is essential to the performance of heterogeneous codes. To handle memory movement without explicit tasks, we allow the user to specify the association between a loop range and its input and output pattern. Our interface takes a pointer, optionally the size of each element and for each dimension whether to associate that dimension with the iterator, the length, and the number of boundary values required. If a dimension is marked as an iterator dimension, then all values in that dimension corresponding to assigned iterations are copied. For example, Figure 4 shows the data associations for two simple cases. On top, the `pcopy(mat[false:10][true:10])` clause specifies that a 10×10 matrix is to be managed and the iterator will select the column, since the column dimension's condition is true. The second example uses `pcopy(mat[true:10:1][false:10])` instead, which has a drastic effect on the result. Now, the rows are associated with the iterator rather than the columns, since the association value is true for the row dimension rather than the column dimension. Further, this example is designed for a stencil-type code which requires boundary values as input, but not as output, so the boundary size in the row dimension has been set to one to copy one row above and one below as input.

Our design handles reductions by doing partial reductions on each device, and a finalizing pass on the CPU into the final target variable. The high-level interface has no extension for this, OpenMP syntax is already sufficient, but our library API provides a mechanism similar to that of user-defined reductions to manually construct more complex reductions. While this interface does not support fully general input and output, we believe it to be a worthwhile first step in that direction, one we intend to pursue further in future work.

4 Results and Evaluation

To evaluate our prototype, we have ported six applications to OpenACC directives and extended them with CoreTSAR scheduling. We evaluate these applications across a range of systems, schedulers, and configurations. We also provide a performance comparison of three of the applications with OmpSs and StarPU.

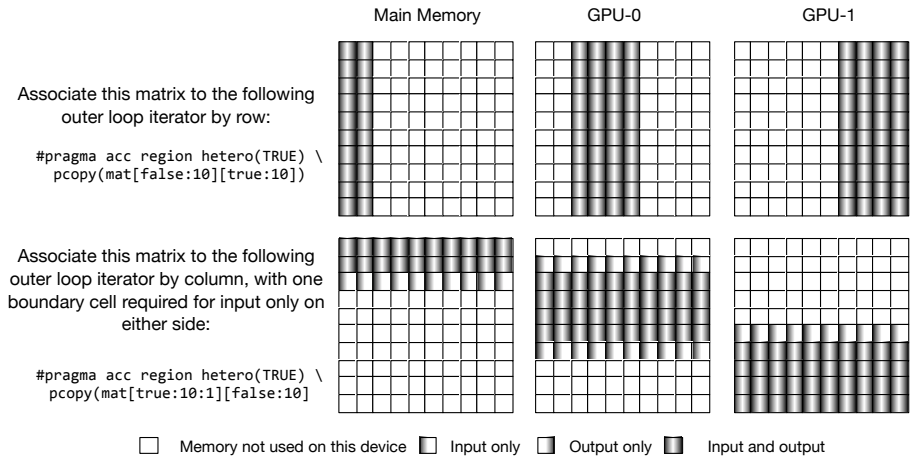


Fig. 4: Example memory associations, assuming a pass in which two iterations are assigned to the CPU device, and four each to two GPUs.

4.1 Benchmarks

Our six benchmarks exhibit a range of characteristics that reflect possible use cases. Minimal changes were made to port the original serial or OpenMP version of each benchmark to OpenACC and subsequently CoreTSAR.

The values in Table 1 characterize the benchmarks. The number of passes, relative pass length, number of iterations per pass and ratio particularly impact CoreTSAR operation. We derive the CPU/GPU ratio, the relative performance of the CPU cores of our test system as compared to one of its GPUs, from the best found through brute-force testing. A score of 1.0 allocates all iterations to the CPU cores, a score of 0 allocates all iterations to the GPU.

Benchmark	Passes	Iterations/ pass	Time/pass (CPU)	GPU runtime	Resulting CPU/GPU Ratio
CG	1900	75,000	0.02	273.04	0.85
CORR	10*	2,048	6.36	70.97	0.01
GEM	1	258,800	1098.10	107.43	0.06
GEMM	10*	2,048	1.262	3.04	0.01
Helmholtz	100	4,000	0.08	73.64	1.00
kmeans	7	1,210,000	1.14	4.79	0.41

Table 1: Benchmark characteristics, times in seconds (* polybench suite benchmarks can have a variable number of passes, we use 10 for our tests)

First, CG is the conjugate gradient benchmark from the NAS parallel benchmarks OMP version 3.3. The offloaded portion runs many (1,900) short passes and contains a high percentage of indirect array accesses. This version of CG is relatively unsuitable for GPUs with a ratio of 0.85, implying that a GPU is only 20% faster than one CPU core. CORR and GEMM are both from the PolyBench/GPU [10] suite. CORR is an upper triangular matrix solver with a

completely unbalanced workload. Iteration i computes $n - i$ output values so no two iterations, or ranges of iterations, are the same. GEMM is a general matrix multiplication that we schedule row-wise, it is highly suitable to GPU computation but is sensitive to memory contention and NUMA effects. GEM is a molecular modeling visualization application that computes the electrostatic potential along the surface of a macromolecule. While GEM is extremely well suited to GPU computation, as we have shown in our previous work [1, 7], it only runs a single pass by default. Helmholtz implements the Helmholtz equation using the Jacobi method. Our implementation is based on a CPU OpenMP implementation that is not well suited to GPUs due to its memory access pattern and conditional nature. Kmeans implements the kmeans clustering algorithm, and is near an even ratio (0.5): running on a GPU takes the same amount of time as on the CPU cores. It also has many iterations per pass, which allows CoreTSAR to make very fine grained adjustments to each device's workload.

Overall, the benchmarks exhibit a wide range in each category. Pass counts for example range from one to 1,900; iterations per pass from 2,048 to 1,210,000; and ratios from 0.01 to 1.0.

4.2 Experimental Setup

CoreTSAR has been implemented in full as a C library, with a source-to-source translator implementing the pragma syntax based on python and libclang, on top of PGI OpenMP and OpenACC. To solve our linear optimization problem, we use the lp_solve library[4], an optimized linear program solver, in a mode that allows it to incrementally refine the solved tableau on each pass. We evaluate all five systems listed in Table 2. However, unless otherwise specified, we run tests on escaflowne. Each machine has the same OS configuration (Debian Squeeze).

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

Table 2: Hardware composition of each test platform, Intel CPUs and NVIDIA GPUs throughout

Unless otherwise noted, all benchmarks are implemented with OpenACC directives and compiled with the PGI compiler version 12.9. OmpSs tests are compiled with mnvcc version 1.3.5.8, using the performance configuration of the NANOS++ libraries and the versioning-stack scheduler. OmpSs options for prefetching and asynchronous transfers are used only on helmholtz, the other two benchmarks incur a slowdown when they are used. The modified CoreTSAR and StarPU versions compared with OmpSs are compiled with nvcc and linked with GNU OpenMP (gomp). The StarPU implementations used the "dmda" scheduler with the history based performance model, trained on ten plus runs before results

were collected. CUDA toolkit version 4.1 is used in all tests. All threads are bound to cores at the beginning of execution, before memory allocation and initialization.

All reported performance measurements time the core phase of the benchmark only. All marshaling, staging, copying, or other preparation necessary to use an implementation is also included. We exclude only identical application parts such as file IO and result verification.

4.3 CoreTSAR Performance

Figure 5 presents the performance impact of CoreTSAR. The graph represents speedup over a statically scheduled eight core CPU run, with a black bar marking the baseline in each sub-plot. Without CoreTSAR unmodified code would be either at that baseline, or the GPU mark with one GPU. All others, including static and GPU for a GPU count greater than one, use CoreTSAR's facilities. Of the six benchmarks, four scale nearly linearly from one to four GPUs given the right scheduler. As expected, Helmholtz and CG do not.

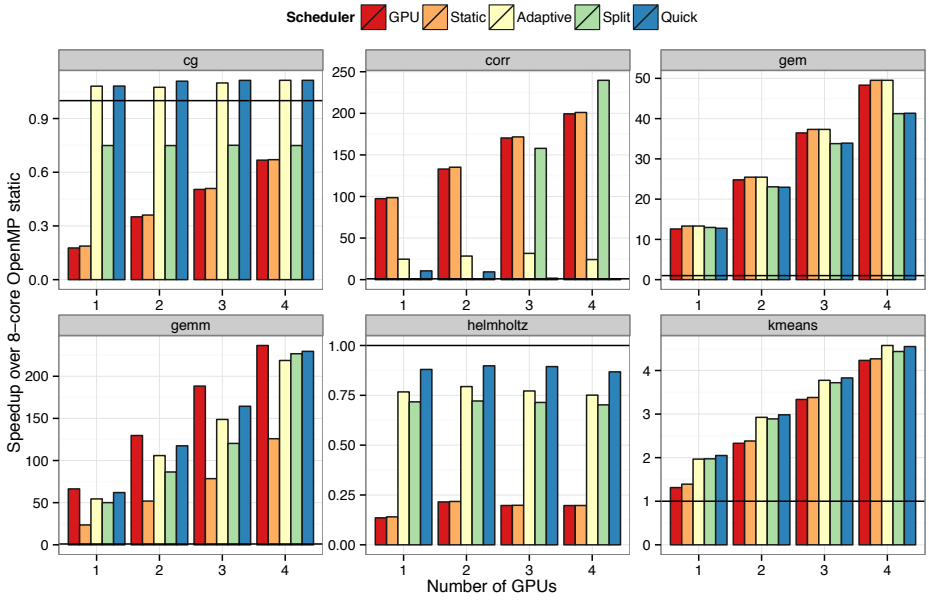


Fig. 5: Performance of CoreTSAR enabled benchmarks on escaflowne normalized to eight core OpenMP

Both CORR and GEMM display high GPU suitability, each approaching an overall speedup of 250× on four GPUs. We obtain the best performance for GEMM on escaflowne by statically scheduling the computation; the quick scheduler is almost as good. CORR is less predictable. Its heterogeneous workload per iteration causes the adaptive, split, and quick schedulers all to make incorrect early decisions about how to assign work. While overall the static schedule fares

best, the split schedule overtakes it for four GPUs because the linear model stops assigning work to the *CPU* cores and adaptively schedules work across the four GPUs. It is worth noting that both the adaptive and quick schedulers converge on efficient assignments after the first few passes, but the cost of the early mispredictions overtake that benefit. Using the ratios from a previous run produces even more favorable results, we intend to investigate this further in future work.

Both GEM and kmeans also scale near linearly to four GPUs, but with slightly different characteristics. GEM benefits most from the default static split across CPUs and GPUs. Its natural ratio is so close to the default split that, while the dynamic schedulers can improve on it, the overhead of synchronization and additional assignments increases the overall execution time. On the other hand, kmeans does best with the adaptive schedules. The optimal choice shifts between quick and adaptive depending on the number of GPUs, but they remain within 5% of each other regardless. Unlike GEM, kmeans is more CPU suitable, and the GPU or static schedules underutilize those resources.

CG is not GPU averse, since it can benefit from the use of a GPU, but it does not scale to more than one GPU. When we allow use of more than one GPU, the increased memory transfer overhead causes a slowdown. This example demonstrates that some applications can benefit from GPUs, but may still need to back off of their use. Helmholtz, on the other hand, gains no benefit. It does however stay within 20% of the CPU performance given a scheduler that can quickly deactivate GPUs. When allowed to use cross-run historical data, Helmholtz consistently matches the CPU performance.

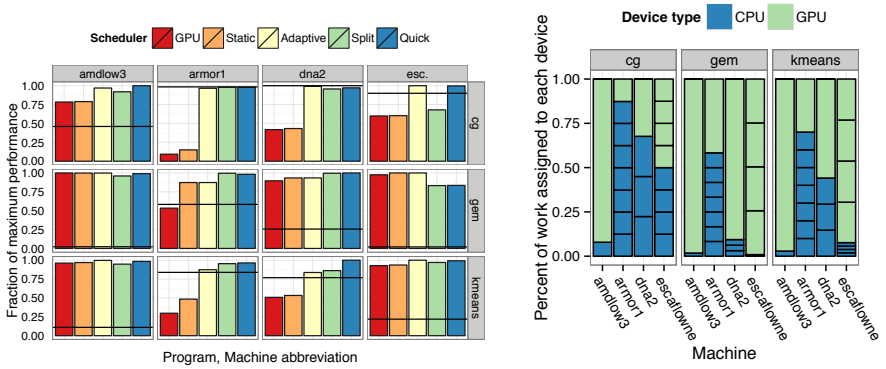
Overall, our results demonstrate that CoreTSAR adapts well to different workloads. We attain good scaling for applications that are amenable to GPU computing or coscheduling. Alternatively, CoreTSAR backs off appropriately for those that are not.

4.4 Adaptation Across Machines

We have shown that CoreTSAR can provide efficient worksharing across a varied number of GPUs in one system. We now evaluate its performance portability across a range of systems, those listed earlier in Table 2. Of these systems, escaflowne, used for our primary evaluation, is the largest and the only multi-gpu system. Representing CPU-centric systems, armor1 contains two capable quad-core Intel Xeon CPUs with a low-power consumer desktop GPU. Conversely amdlow3 contains a dual-core Celeron CPU and a powerful Tesla C2050 GPU. Lastly dna2 represents a more typical workstation with a mid-range quad core CPU and previous-generation consumer GPU.

Figure 6a presents results for the three benchmarks that benefit from coscheduling. Helmholtz is CPU-suited on all machines, and corr and gemm are GPU-suited on all machines, so we elide their results for space. Each result is normalized to the best performing configuration for that benchmark on that system with the best time at 1.0 and lower being worse. First, while CG gains minimally in performance on escaflowne, and not at all on armor1 and dna2, it attains a 2× speedup over the CPU result on amdlow3. Second, we find that

appropriate adaptive schedulers tend to hold across different systems. For both CG and kmeans the quick scheduler performs best across all tested machines, regardless of their composition. GEM is a bit different, in that on some systems the quick or split schedulers are slightly beneficial, but on escaflowne they are slightly worse than the static split. Run in production, where a user will invoke the visualization routine repeatedly, the quick scheduler is fastest everywhere.



(a) Performance across systems

(b) Work distribution across devices with the Adaptive scheduler

Fig. 6: Cross-system performance and assignment results. The black bar in (a) represents OpenMP CPU performance across all cores in that system.

Even though the appropriate scheduler remains the same across machines, the workload distribution across resources shifts significantly, as Figure 6b shows. Each bar represents the total amount of work run during the course of a given benchmark, each CPU cell represents the work in iterations completed by a particular CPU core, and each GPU cell the work completed by a particular GPU. Each benchmark's distribution shifts across machines based on the machine's suitability. The most striking of these is CG, which uses the GPU for very little on three of the four systems, but almost exclusively on amdlow3.

4.5 Comparison with State of the Art

Our evaluation has demonstrated that CoreTSAR achieves good and portable performance. We now compare its performance to that of two state-of-the-art heterogeneous task schedulers, OmpSs and StarPU. In order to compare all three fairly, we ported three of our benchmarks (kmeans, Helmholtz and GEMM) from OpenACC to CUDA/C and extended that version with each scheduler. The CoreTSAR code evaluated here uses the exact same CUDA/C implementations as OmpSs and StarPU, in fact *linked from the same binary*. Additionally, we configure all three schedulers to use the same initial granularity for each benchmark, amounting to approximately 2000 tasks per pass.

Figure 7 presents our results for these benchmarks on escaflowne for all schedulers targeting all eight CPU cores and four GPUs. The result for GEMM is

rather unexpected, with OmpSs and StarPU both about 2× slower than the CoreTSAR version. We initially suspected that this was the result of extra data transfers, or even an error in our implementation of the memory movement in OmpSS and StarPU, but manually minimizing the data transfers did not materially change the result. Rather, overhead from creation, management and scheduling of individual tasks is to blame for the difference. CoreTSAR has the advantage of automatically altering the granularity of tasks, rather than running user-defined chunks. In the adaptive scheduler for example, a single kernel is run on each GPU, whereas in OmpSs hundreds to thousands may be run. No matter how efficient the runtime, there is a cost for such fine-grained management.

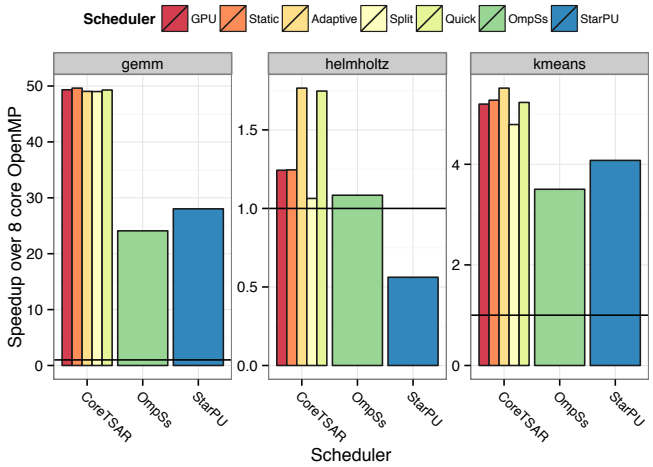


Fig. 7: Speedup comparison between CoreTSAR, OmpSs and StarPU

Helmholtz is of interest for a different reason. The CUDA and serial C version gets materially faster by running on four GPUs, especially with the CoreTSAR adaptive or quick schedules with a nearly 75% improvement. The reason it performs so differently from the version evaluated above is that the nvcc compiler produces significantly slower CPU code than the PGI compiler used elsewhere, allowing the GPUs to outperform the CPUs. This result reinforces the idea that allowing automatic coscheduling, even in cases where it does benefit some machines or configurations, can be beneficial. OmpSs also improves on the CPU-only performance by about 5%, and StarPU almost 2× slower. Our evaluation found that this is due to underutilizing the CPU cores in favor of the GPUs, as well as the same overheads that plague GEMM.

Finally the kmeans results show OmpSs and StarPU scaling to a respectable 3.5×, but still trailing the CoreTSAR adaptive schedulers. For cases where the application launches and immediately waits for a range of related tasks, CoreTSAR consistently performs well. OmpSs and StarPU on the other hand perform well with many asynchronously launched tasks which wait rarely, and suffer when the full synchronization is more frequent. We believe these are complimentary designs, and will investigate granularity adaptation for general task graph scheduling systems as future work.

5 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 [8] and Intel's TBB [15]. These mechanisms tend to offer simplified syntax for shared memory parallelism, but little to no support for heterogeneous architectures or distributed memory. Scheduling policies for these mechanisms have also been the focus of significant research. Work by Ayguadé et al. [3], directly influenced the design of CoreTSAR. They investigated the removal or extension of OpenMP schedule clauses by calculating the distribution of work in future passes through a region based on times seen for each core in previous ones. Their results showed the method was not always optimal, but that the solution was efficient and stable. Our ratio-based design works similarly, although with a different mechanism to determine the split between devices.

Task block models, such as StarPU [2] and OmpSs [9], began to tackle the problem of scheduling tasks across heterogeneous resources based on a directed acyclic graph. These models provide a powerful platform for scheduling on heterogeneous systems, but require the user to determine task granularity manually. We believe these designs to be complementary to our own, combining the ability to adaptively adjust task granularity with task-block style arbitrary graph execution could yield powerful results.

Ravi et al. [13] presented a scheduling framework for multicore systems with a single GPU that builds on their generalized reduction framework and code generator [14]. While we avoid the chunk scheduling scheme and its additional transfer overheads, they present an approach that uses chunk-based scheduling while mitigating the overhead through runtime techniques.

6 Conclusion

We present four major contributions: the design of our task-size adapting runtime for adaptive worksharing across heterogeneous devices; the design of a task-associative memory model; an implementation of the designs; and our evaluation across six codes, and four systems. Our system yields linear speedups on up to four GPUs for amenable benchmarks, and show a high degree of performance portability across a set of highly disparate system configurations. These results clearly motivate the addition of a co-scheduling interface, such as the `hetero()` clause that we propose, to Accelerated OpenMP; the results also highlight the benefits of automatically adapting task granularity at runtime.

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. We intend to investigate this expanded interface as future work.

References

1. R. Anandakrishnan, T. R. Scogland, A. T. Fenley, J. C. Gordon, W.-c. Feng, and A. V. Onufriev. Accelerating Electrostatic Surface Potential Calculation with Multi-Scale Approximation on Graphics Processing Units. *Journal of Molecular Graphics and Modelling*, 28(8):904–910, 2009.
2. C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In H. Sips, D. Epema, and H. Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704, pages 863–874. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
3. E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera. Is the Schedule Clause Really Necessary in OpenMP? In *WOMPAT'03: Proceedings of the Workshop on OpenMP Applications and Tools 2003*. Springer-Verlag, June 2003.
4. M. Berkelaar, P. Notebaert, and K. Eikland. lp_solve:(mixed integer) linear programming problem solver. http://lpsolve.sourceforge.net/5.0/, 2003.
5. J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski. OpenMP for Accelerators. In B. M. Chapman, W. D. Gropp, K. Kumaran, and M. S. Mller, editors, *OpenMP in the Petascale Era*, volume 6665, pages 108–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
6. CAPS Enterprise, Cray Inc., NVIDIA and the Portland Group. The openacc application programming interface, v1.0. http://www.openacc-standard.org, Nov. 2011.
7. 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.
8. L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, Mar. 1998.
9. A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.
10. S. Grauer-Gray, L. Xu, R. Searles, and S. Ayalasomayajula. Auto-tuning a High-Level Language Targeted to GPU Codes. *cis.udel.edu*.
11. Khronos OpenCL Working Group and others. The opencl specification. *A. Munshi, Ed*, 2008.
12. OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.
13. 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.
14. V. T. Ravi, W. Ma, D. Chiu, and G. 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.
15. J. Reinders. Intel Threading Building Blocks. 2007.
16. T. R. W. Scogland, B. Rountree, W.-c. Feng, and B. R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *2012 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Shanghai, China.