LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-738086

Experiences Using CPUs and GPUs for Cooperative Computation in a Multi-Physics Simulation

O. Pearce

September 6, 2017

Eleventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), 2018
Eugene, OR, United States
August 13, 2018 through August 13, 2018

Disclaimer

Experiences Using CPUs and GPUs for Cooperative Computation in a Multi-Physics Simulation

Olga Pearce
Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore CA 94550, USA
olga@llnl.gov

ABSTRACT

Top supercomputers in the TOP500 list have transitioned from homogeneous node architectures toward heterogeneous manycore nodes with accelerators and CPUs. These new architectures present significant challenges to developers of large-scale multiphysics applications, especially at DOE laboratories that have invested heavily in scalable MPI codes over decades. Much of these scientific application porting efforts for the new heterogeneous architectures are focused on running the computation on the accelerators, which usually comprise >90% of the FLOPS of the system. We describe an approach to utilizing the remaining FLOPS on a heterogeneous machine by running a portion of the computation on the CPUs cooperatively with the GPU computation. We present a proof-of-concept implementation in ARES, a multiphysics ALE-AMR code at LLNL. ARES uses a portability layer, RAJA, which enables us to utilize the same source code for both the CPU and the GPU. We develop an approach to utilize both types of processors cooperatively in a mixed-processor system. Our implementation divides the work between the computing resources via domain decomposition, and utilizes all cores of the CPU and all of the GPUs on the node for computation. Load balancing is necessary to use the heterogeneous resources effectively. We present preliminary results on early delivery pre-Sierra machines at LLNL, showing up to an 18% performance benefit of using the CPUs on the heterogeneous nodes for computing in addition to using the GPUs.

CCS CONCEPTS

• **Computing methodologies-Parallel programming languages**;

KEYWORDS

GPU, multi-physics, simulation

ACM Reference Format:
Olga Pearce. 2018. Experiences Using CPUs and GPUs for Cooperative Computation in a Multi-Physics Simulation . In *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3229710.3229711

1 INTRODUCTION

Until recently, the supercomputers on the TOP500 [1] list were primarily architectures with homogeneous nodes. One of the first mixed-processor systems, Roadrunner [6], took the number one spot on the TOP500 list in June 2008, and became the first petaflop machine. It paired IBM PowerXCell 8i coprocessors with standard dual-core x86 CPUs. Difficult to program, Roadrunner became a one-off machine, but its heterogeneous design foreshadowed the coming age of HPC accelerators. NVIDIA brought to HPC its general-purpose GPUs and its CUDA programming environment, while Intel introduced its manycore Xeon Phi, offering an x86-flavored coprocessor with a more CPU-like software development model. Currently, accelerators are the only commodity-based technology enabling multi-petaflops within reasonable power envelopes, and several supercomputers with heterogeneous nodes are poised to headline the upcoming TOP500 lists.

While much of the scientific application porting efforts for the new heterogeneous architectures focus exclusively on utilizing the accelerators, accelerator performance can be limited by such restrictions as memory size and bandwidth, data transfer overhead, and kernel launch overhead. At the same time, the heterogeneous nodes often contain top of the line CPUs, computational resources of which should not be overlooked when trying to achieve optimal performance.

Heterogeneous computing has long utilized both types of processors on a system for improved performance. Many of the heterogeneous computing efforts utilize separate code paths for the different processors, an approach not feasible for large scientific simulations with hundreds of thousands of lines of code. In this work, we employ a performance portable approach which enables us to explore the different modes of

ICPP '18 Comp, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6523-9/18/08…$15.00
https://doi.org/10.1145/3229710.3229711

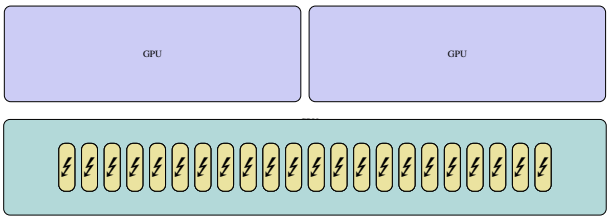

Figure 1: CPU only: Computing kernels on the CPU cores only, all of the GPUs are idle

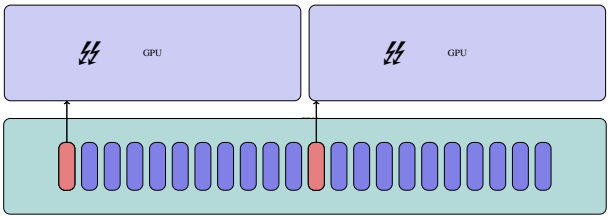

Figure 2: 1 MPI/GPU: One CPU core drives each GPU, computing kernels on the GPUs only

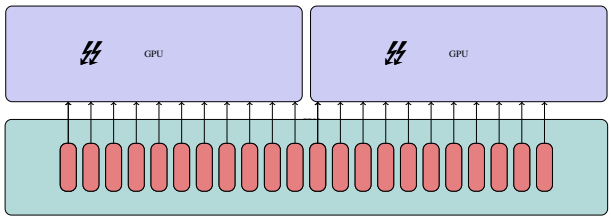

Figure 3: n MPI/GPU: Multiple CPU cores drive each GPU, computing kernels on GPUs. Requires MPS.

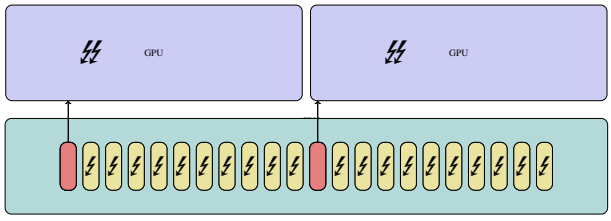

Figure 4: Heterogeneous: Subset of CPU cores drive the GPUs (GPUs compute kernels), the rest of CPU cores compute kernels

utilizing the heterogeneous nodes while maintaining a single source code of the application. We present a proof of concept implementation in a large multi-physics application, ARES. We compare the performance of using the CPU solely to offload the computation to the GPU, and utilizing both the CPUs and the GPUs cooperatively for computation.

Our contributions include:

(1) An approach to utilize both types of processors cooperatively in a mixed-processor system.
(2) A proof of concept implementation for utilizing the GPUs and all CPU cores cooperatively to perform loop computation in a large multi-physics application, ARES.
(3) A hierarchical domain decomposition scheme for the heterogeneous node.
(4) Performance comparison of three different modes of utilizing a heterogeneous node: 1. One MPI process per GPU to drive the GPU; 2. Multiple MPI processes per GPU to drive the GPU; 3. One MPI process per GPU to drive the GPU, and other MPI processes executing on the remaining CPU cores.

This paper is organized as follows. Section 2 describes the different modes of utilizing heterogeneous nodes. Section 3 describes the multi-physics application we use for this study, ARES. Section 4 introduces RAJA, the performance portability layer we use in this study. We introduce our approach to utilize both types of processors cooperatively in a mixed-processor system in Section 5. Section 6 describes how we divide work between the two types of processors. Section 7 compares performance of different modes of utilizing the heterogeneous nodes. We discuss related work in Section 8.

2 MODES TO UTILIZE HETEROGENEOUS NODES

The next flagship supercomputer at the Lawrence Livermore National Laboratory will be SIERRA, with 2-socket nodes containing two POWER9 CPUs and four Volta GPUs. With the GPUs comprising 95% of the FLOPs of the machine, the main focus of the application porting work has been to utilize the GPUs effectively. We explore the different modes to utilize these heterogeneous nodes.

Figure 1 shows a single socket of a SIERRA node. The lightning bolts on the CPU cores indicate the traditional way scientific applications use a node of the supercomputer: an MPI task is bound to each core and executes its own work.

Figure 2 shows how this MPI application might utilize the heterogeneous node when first ported to the GPUs: an MPI task is bound to a single core per GPU, and is used to launch kernels on the GPU. The lightning bolts over the GPUs indicate that the GPUs are executing their portions of work. Two CPU cores (in red) are used to drive the GPUs, while the remaining CPU cores are idle.

Figure 3 shows the MPI application using all cores on the CPU to offload work to the GPUs. Because each process has a unique context and only a single context can be active on a device at a time, multiple processes (e.g., MPI processes) cannot operate concurrently on a single GPU. NVIDIA provides a Multi-Process Service (MPS) [2], a software layer between the application and the driver. MPS routes all CUDA calls through a single context, allowing for the multiple processes to execute concurrently. With MPS, there is a potential for the different GPU operations to overlap. The caveat is that the kernel launch overhead is higher.

C-style for-loop:

```
1: double* x; double* y;
2: double a;
3: for (int i = begin; i < end; ++i ) {
4:     y[i] += a * x[i];
5: }
```

RAJA-style loop:

```
1: double* x; double* y;
2: double a;
3: RAJA::forall<exec_policy>(begin, end, [=] (int i) {
4:     y[i] += a * x[i];
5: });
```

Figure 5: RAJA abstraction

Figure 4 shows our proposed approach to utilizing the heterogeneous node. We propose to bind an MPI task to each CPU core. We then use a single CPU core to launch kernels on the GPU. We use the remaining CPU cores to perform additional computation cooperatively with the GPUs.

While using a single core to launch kernels on the GPU may be sufficient when the kernels are large, it has been informally stipulated that multiple cores may be necessary to keep the GPU occupied if the kernels are small. When a single core is sufficient to drive the GPU, we propose to utilize the remaining cores for additional computation. Although it is hard to project performance to future hardware and software stack, in this work we explore the performance of these three approaches on the currently available early delivery systems.

3 MULTIPHYSICS SIMULATIONS

Multidimensional multiphysics simulations are widely used to simulate physics phenomena. Running multiphysics simulations is computationally expensive, and utilizing supercomputers is key to advancing our understanding of science through simulation. With the advance of heterogeneous architectures, multiphysics simulations are now poised to utilize these new architectures to advance science.

We explored our ideas on how to use the heterogeneous architectures in ARES, a massively parallel, multi-dimensional, multi-physics code at the Lawrence Livermore National Laboratory (LLNL). ARES supports an array of physics, including arbitrary Lagrangian-Eulerian (ALE) hydrodynamics, high-order Eulerian hydrodynamics, elastic-plastic flow, 3T plasma physics, high-explosive modeling, diffusion, S_N radiation, particulate flow, laser ray-tracing, MHD, dynamic mixing, and non-LTE opacities. ARES is capable of running massively parallel applications on millions of processors [8],[14]. ARES applications include pulsed power, high-explosive experiments, Inertial Confinement Fusion (ICF) modeling, National Ignition Facility (NIF) debris, and instability experiments.

Excluding libraries, ARES has roughly 700,000 lines of primarily C/C++ code. ARES uses MPI for inter-node parallelism. Its 2D/3D block-structured mesh, with or without Adaptive Mesh Refinement (AMR), is spatially decomposed into domains, and domains are assigned to MPI processes.

For fine-grained intra-node parallelization needed to utilize accelerators, ARES has been ported to the RAJA loop abstraction described in Section 4.

4 RAJA PROGRAMMING MODEL

Developers of large scientific simulations have a strong incentive to have a single source code of their applications for maintainability. Several programming model abstractions have been proposed (Kokkos [10], RAJA [16]) to provide an ability to execute a single source code on multiple platforms. For this work, we use RAJA [16], a performance portability layer developed at the Lawrence Livermore National Laboratory and used for on-node parallelization in ARES.

RAJA is an on-node parallelization approach for C++ applications. RAJA uses C++ lambda functions and templates to separate loop traversal from a loop body. The mechanism enabling various loop traversal schemes (e.g., sequential, SIMD, parallel across multiple cores) is implemented in the RAJA library and is invisible to the application developer. RAJA provides multiple programming model backends (e.g., CUDA, OpenMP).

Figure 5 shows how to convert a simple C++ loop to RAJA. The left side shows a C++ style loop, and the right side shows the same loop with RAJA constructs. On the right, the loop body is expressed as a lambda function parameterized by the iteration variable. The lambda captures the variables used within it *by reference*, so they are visible at the scope of the call site. The **RAJA::forall** is templated with an execution policy, and takes the loop body and its iteration bounds as the parameters.

The execution policy specified controls which RAJA programming model backend executes the loop. Figure 6 shows an outline of two RAJA backend implementations, OpenMP and CUDA. The backends hide specific compiler directives and lower level programming models from the user. The user only sees the abstract execution intent shown in Figure 5.

5 COMPUTING ON BOTH TYPES OF PROCESSORS ON A HETEROGENEOUS NODE

To compute cooperatively on different types of processors, an application needs to have the control code to specify which of the tasks each process should perform, a way to execute the kernels on the different types of processors, and application

OpenMP backend outline:

```
1: template <typename LOOP_BODY>
2: inline void forall(
3:    omp_parallel_exec, int begin, int end,
4:    LOOP_BODY loop_body)
5: {
6:    #pragma omp parallel for
7:    for (int i = begin; i < end; ++i ) {
8:       loop_body(i);
9:    }
10: }
```

CUDA backend outline:

```
1: template <typename LOOP_BODY>
2: inline void forall(
3:    cuda_parallel_exec, int begin, int end,
4:    LOOP_BODY loop_body)
5: {
6:    //setup begin, end, len, gridSize, ...
7:    forall_cuda_kernel<<<gridSize,
8:       BLOCK_SIZE, 0, stream>>>(
9:          createLaunchBody(...,loop_body(i)),begin,len);
10: }
```

Figure 6: RAJA backends

```
1: if (run_on_gpu) then
2:    //RAJA backend: GPU specific (CUDA, OpenMP)
3:    typedef DynamicPolicy<AresPolicy, GPU> AresArchPolicy;
4:    RAJA::forall<AresArchPolicy>(..., kernel);
5: else
6:    //RAJA backend: CPU specific (Serial, OpenMP)
7:    typedef DynamicPolicy<AresPolicy, CPU> AresArchPolicy;
8:    RAJA::forall<AresArchPolicy>(..., kernel);
```

AresArchPolicy is now:

- **CUDA** exec_policy on GPU-driving MPI processes
- **Sequential** exec_policy on CPU-only MPI processes

Figure 7: Dynamically selecting a RAJA policy in ARES

data should be allocated accordingly. We will address the performance optimization side of work division in Section 6, and describe the code necessary for executing cooperatively on a heterogeneous node in this section.

An application needs robust control code which directs some of the MPI processes to offload work to the GPUs, while the remaining MPI processes compute on the CPU cores directly. Our experience indicates that the CPU core/GPU binding needs to be carefully set up to avoid performance degradation.

First, the application has to be able to execute the kernel code on the specified processor. In our case, we instruct some of the MPI processes to offload the kernels to the GPU, while other MPI processes execute their kernels on the CPU cores directly.

Second, the application needs a memory allocation scheme that can allocate the memory according to the task specific to each CPU core. In a spatially decomposed MPI application where each MPI process owns its data, it was a matter of specifying what to do for each MPI task.

Finally, MPI communication can become more complex on heterogeneous machines.

5.1 Heterogeneous kernels

In order to execute kernels on different types of processes, a single-source application would need to use a performance portability layer that enables it to specialize the code for both types of processes.

In our experiments, we use RAJA, a loop abstraction utilized by several multiphysics simulations at LLNL, including ARES. Our application, ARES, defines several execution policies, indicating whether the loop is thread safe, not thread safe, has a significant amount of work, etc. These execution policies can then be defined to use different RAJA backends depending on the architecture. We leveraged the existing RAJA-based implementation but added the control code to inject additional context, specifying where the process should execute the loop by defining an architecture-specific policy. AresArchitecturePolicy is the architecture-appropriate policy selected at runtime from predefined RAJA policies, as shown in Figure 7. In the future, we plan to use the MultiPolicy runtime policy selection mechanism in RAJA.

Our runtime policy selection results in CUDA-specific policies used on MPI processes driving the GPU, and sequential execution policies on the CPU-only MPI processes.

We did come across a compiler issue hampering the performance of this approach. We rely on __host__ __device__ decorated lambdas for our performance portable implementation. The CUDA Toolkit 8.0 EA release has support for __host__ __device__ decorated lambdas. However, when running such code on the CPU (e.g., using a RAJA CPU execution policy), the performance is substantially worse than running the same code on the CPU without any CUDA host-device decorations (execution time can be 100x to 300x slower). The issue is that nvcc passes the lambda back to the host compiler wrapped in a std::function object. The effect is that each time the lambda is invoked (e.g., at each loop iteration) a virtual function dispatch is required. We expect this compiler issue to be resolved in the future, which will allow us to assign more work to the CPU.

Data type:	Process executes on CPU core:	Process offloads execution to GPU:
Control code	Malloc	Malloc
Mesh data	Malloc	cudaMallocManaged(UM)
Temporary data	Malloc	cudaMalloc (cnmem memory pools)

Figure 8: ARES memory allocation

5.2 Heterogeneous memory allocation

Application memory should be allocated according to where the kernels will be executed. The application we studied, ARES, differentiates memory use by context, whether it is control code, mesh data, or temporary data. We used the control code to inject additional context, specifying where the process executes the kernels, as shown in Figure 8. When executing on the CPU, all data should be allocated on the CPU. When executing on the GPU, mesh data is allocated in Unified Memory (UM) so it can be accessed from the CPU and the GPU. ARES uses memory pools for the temporary data for performance optimization.

During the implementation, we encountered assumptions made by the libraries used by ARES. When the libraries are compiled to use CUDA, they often allocate memory on the GPU. We had to break these assumptions to avoid touching the GPU memory from the processes executing solely on the CPU; touching the GPU memory from these processes degraded the performance of the application.

5.3 Heterogeneous communication

Communication on a machine with heterogeneous nodes is more complicated as well. Currently in ARES, the communication happens through the host (CPU) only. Future hardware and software will enable direct communication between GPUs, called GPU direct. We plan to explore how GPU direct communication impacts the performance of the different approaches to utilizing the heterogeneous nodes.

6 DIVIDING WORK ON A HETEROGENEOUS NODE

6.1 Domain Decomposition

Physical simulations typically divide the simulated space into domains (spatial decomposition), and assign each domain to an MPI process. This MPI process is then responsible for computing its portion of the answer on its domain. For many algorithms, this means traversing the domain (e.g., some portion of the mesh) and performing computation per domain element (e.g., mesh zone). Domain decomposition has a strong impact on the performance of the application because it impacts load balance, memory accesses, communication overhead, and in some cases method convergence.

Communication overhead tends to be lowest when the decomposed domains are near squares in 2D or cubes in 3D, rather than long slabs. Figure 9 gives a simplified illustration

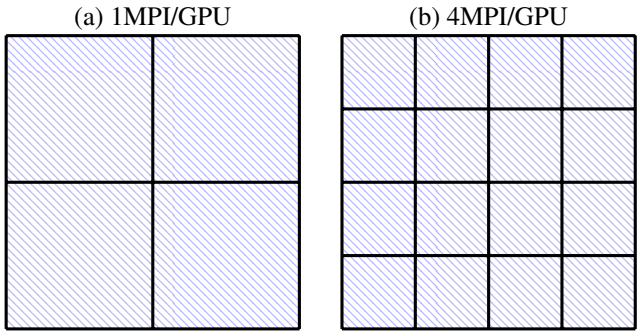

Figure 9: Domain decomposition: 4 vs. 16 domains

of a decomposition in 2D, with 'square' domains for 4 MPI ranks (an equivalent of one MPI rank per each GPU on the node) vs. 16 MPI ranks (an equivalent of four MPI ranks per each GPU on the node). Consider the halo exchange for a single node; the communication overhead is significantly higher when using 16 MPI ranks than when using four. Additionally, on a small scale, the number of neighbors in the halo exchange goes up dramatically when increasing the number of ranks on the node. Therefore, if using 'square' decomposition, the approaches of utilizing the heterogeneous node which have more than a single MPI rank per GPU have the potential of having significantly higher communication costs than the default approach with a single MPI rank per GPU.

To avoid higher communication costs, we introduced a hierarchical domain decomposition method. The first step in our hierarchical domain decomposition method is to divide the work into the number of GPUs available to solve the problem, as shown in Figure 10 (a). Then, for the approaches utilizing more than one MPI process per GPU, we further divided the domain into smaller domains as appropriate. For the 4 MPI ranks per GPU approach with MPS, in this second step we subdivided the work on a GPU in a single dimension, as shown in Figure 10 (b). The subdivision in a single dimension kept the number of neighbors communicating in the halo exchange minimal. We experimentally verified that this approach does in fact minimize the communication overhead of using additional MPI ranks on the node.

6.2 Heterogeneous Load Balancing

In a heterogeneous case, additional care should be taken to load balance the workload between the CPUs and the GPUs since their loads will be processed at different speeds. How quickly each type of processor can process the work depends on many factors:

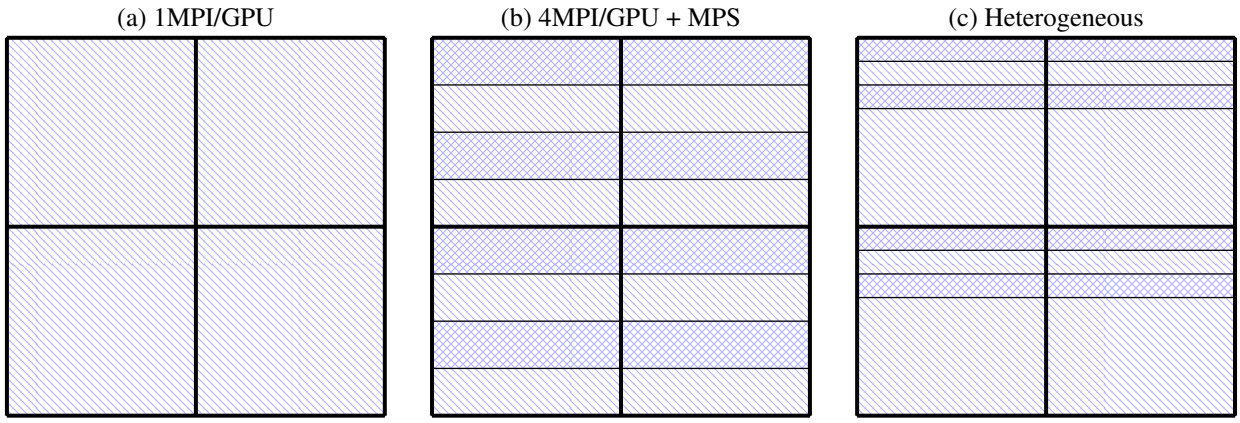

Figure 10: Domain decomposition in ARES: Keeping the size of the x-dimension the same for all approaches

- Processor speed;
- Memory capacity and bandwidth;
- Additional data transfers that may be necessary (as is the case for the GPU);
- Kernel launch overhead (as is the case for the GPU);
- Communication overhead.

To achieve load balance in the heterogeneous case, we used a weighted decomposition between the CPU cores and the GPUs, assigning less work to the CPU cores, as illustrated by the thin slabs in Figure 10 (c). We started with an initial guess of work split between the processors based on FLOPS. We measured the respective contributions of CPU vs. GPU, and adjusted the split to achieve load balance. However, we discovered that on the Sierra early delivery systems at LLNL, performance of the CPU is hampered by the compiler issue described in Section 5.1. While we were able to account for this performance issue and adjust the load balance accordingly, the compiler bug severely limits the effective CPU speed and therefore the amount of work we are currently able to assign to the CPU cores. Once the compiler issue is resolved, we expect to be able to assign significantly more work to the CPU cores, speeding up overall execution using the heterogeneous approach.

Changing hardware and software stacks make it difficult to project performance of Sierra based on the early delivery Sierra systems we are working with currently. Our study of the performance implications of different modes of using the heterogeneous nodes and varying the work assignment to processors is exploratory and a snapshot in time.

7 PERFORMANCE COMPARISON

In this section, we compare the performance of running a single MPI process per GPU, using 4 MPI processes to drive the GPU with MPS, and our heterogeneous approach which uses 4 MPI processes to drive the GPU, and the remaining 12 cores on the CPU to perform a portion of the computation.

For all the studies below, we use a single node of RZHasGPU, a small cluster with GPUs at Lawrence Livermore National Lab (LLNL). Each node of RZHasGPU has two 8 core Intel Xeon E5-2667 v3 processors and 4 NVIDIA Tesla K80 GPUs. Each node has 128 GB of memory (8 GB per CPU core), and each GPU has 12 GB of GPU global memory. RZHasGPU runs the TOSS 2 operating system.

We study the performance of a common hydrodynamics test problem, a 3D Sedov blastwave problem [18] shown in Figure 11. The Sedov problem stresses the hydrodynamics calculation in ARES. For the GPUs, we use the RAJA CUDA backend.[1] For our experiments, we varied the size of the problem in all three dimensions.

Figure 12 shows the performance of a single MPI process per GPU (four MPI processes total), using 4 MPI processes to drive each GPU with MPS (16 MPI processes total), and our heterogeneous approach with a single MPI process to drive each GPU and 12 more MPI processes (16 MPI processes total). The y-axis shows the runtime for the different approaches. The main x-axis shows the total size of the problem in zones, while the top x-axis shows the size of the y-dimension of the problem in zones. As shown in Figure 10, we cut the y-dimension to form the domains for the CPU cores to process. At the higher end of the y-dimension size, we are assigning 1.5% of zones to the CPU, so the 12 cores are executing 1.5% of work while the four GPUs execute the remaining 98.5% of work. At the lower end of the y-dimension size, the smallest number of zones we are able to assign to the CPU (12 cores) is 15% of zones, which is more than the relative compute power available on the CPU in this heterogeneous node. As a result, for the lower problem sizes in the figure, the CPU cores are the bottleneck, resulting in slower execution times in the Heterogeneous mode. For the higher problem sizes, the

[1] All results generated with pre-release versions of IBM compilers; improvements in performance expected in future releases.

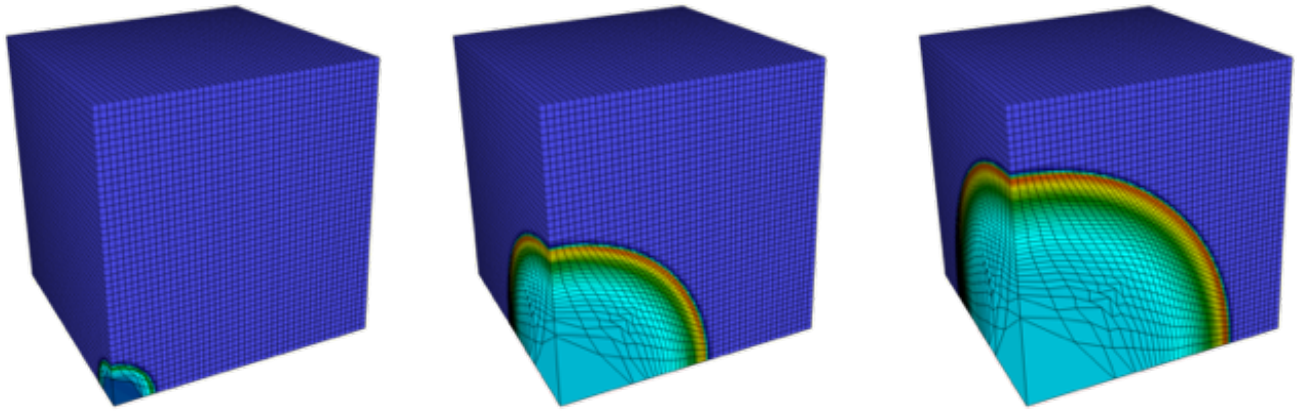

Figure 11: 3D Sedov blastwave problem, a hydrodynamics calculation with 80 kernels

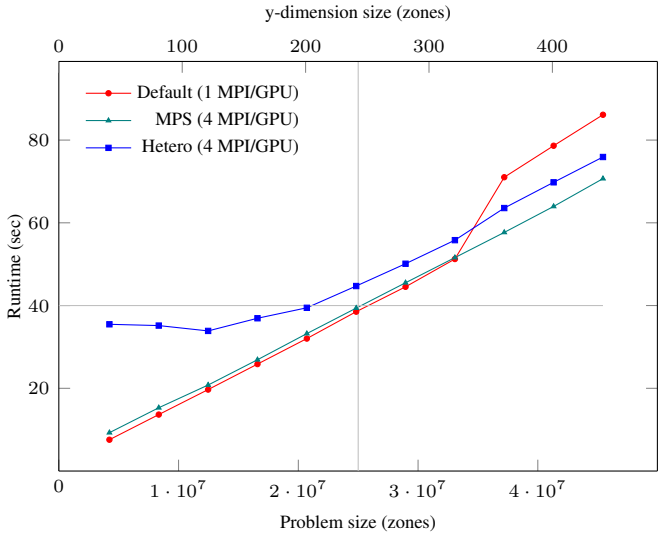

Figure 12: Varying the size of the y-dimension (x=320, z=320)

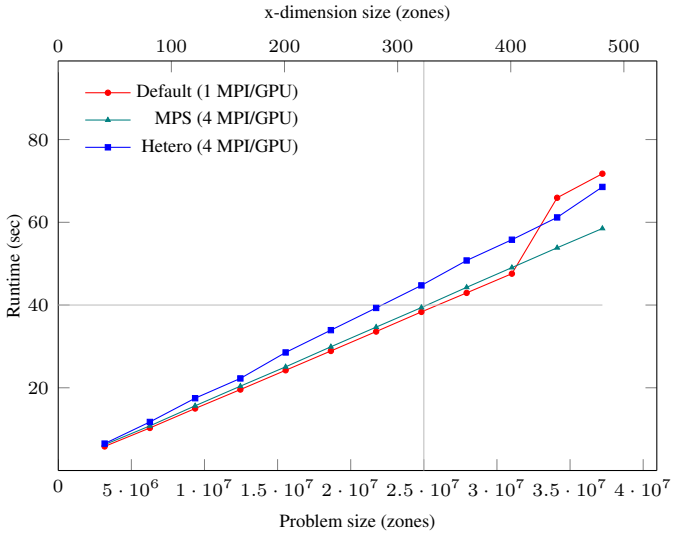

Figure 13: Varying the size of the x-dimension (y=240, z=320)

CPU cores are not overloaded, and the Heterogeneous mode performs well.

Figure 12 also indicates that the Default approach (one process per MPI) hits a memory threshold when the problem size reaches ≈37 million zones (9 million zones per MPI). The other approaches do not hit this threshold because they have four times more domains, and therefore fewer zones per domain. Until the memory threshold point, the MPS and the Default approaches perform similarly. After the threshold, the Default approach pays a penalty while the MPS approach and the Heterogeneous approach continue to scale linearly with the size of the problem. We speculate that this threshold may be due to CPU memory bandwidth utilization, where more MPI ranks (and therefore cores utilized) add additional capacity; the tools to verify this speculation are expected to become available later this year, at which point we intend to verify whether memory bandwidth utilization is in fact the cause for the difference.

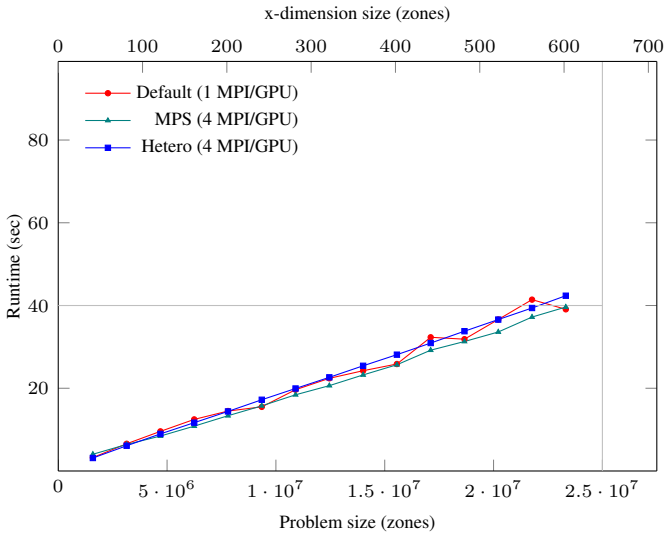

Figure 14: Varying the size of the x-dimension (y=240, z=160)

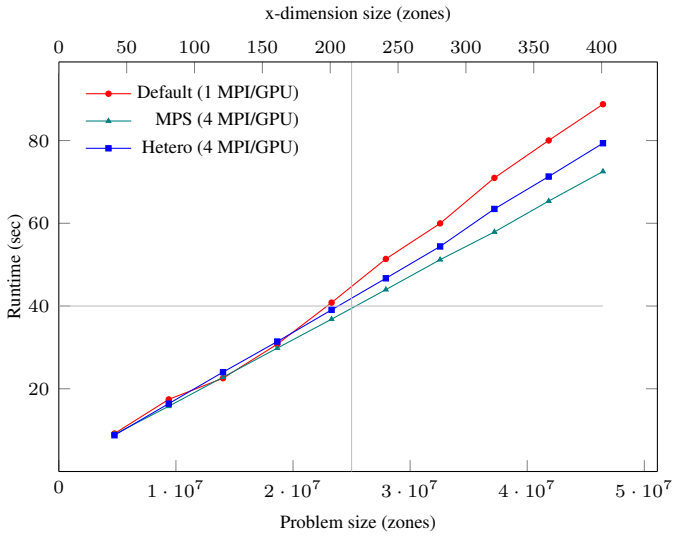

Figure 15: Varying the size of the x-dimension (y=360, z=320)

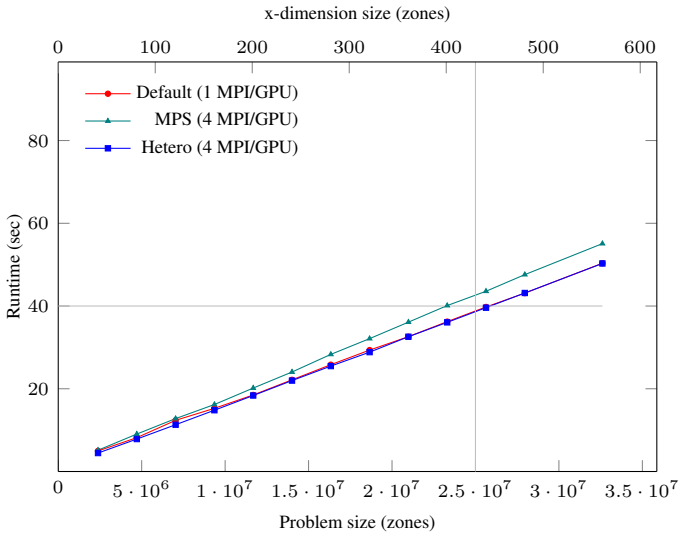

Figure 16: Varying the size of the x-dimension (y=360, z=160)

In Figure 13, we show problems with y-dimension of 240 zones, and z-dimension of 320 zones. We vary the x-dimension as shown on the top x-axis; the total problem size (in zones) is shown on the main x-axis. One MPI process per GPU mode of utilizing the heterogeneous node performs best until the memory threshold. Because the x-dimension of the problems is relatively small, the MPS approach overlaps execution of several kernels launched from different MPI ranks, resulting in better performance. This set of problems has a relatively small y-dimension, which is the dimension in which we partition off work for the CPU, so we are unable to assign small enough portions of work to the CPU cores in the Heterogeneous approach. Since the GPUs have to wait for the CPU cores to finish their work, the overall runtime of the heterogeneous mode is longer for this problem set.

In Figure 14, we show problems with y-dimension of 240 zones, and z-dimension of 160 zones. We vary the x-dimension as shown on the top x-axis. Because the z-dimension is smaller than in Figure 13, the x-dimension size goes to a larger value. The y-dimension is still too small to carve out work for the CPU, thus the runtime of the Heterogeneous approach is longer. One MPI process per GPU and the MPS mode perform similarly.

In Figure 15, we show problems with y-dimension of 360 zones, and z-dimension of 320 zones. We vary the x-dimension as shown on the top x-axis. Similarly to Figure 13, the x-dimension is relatively small, and the MPS approach is able to overlap multiple kernels launched from different MPI ranks, outperforming the other modes. The y-dimension is larger than in Figure 13, enabling better assignment of work to the CPU cores and therefore lower runtime in the Heterogeneous mode. The memory threshold hampers the performance of the single MPI process per GPU mode.

In Figure 16, we show problems with y-dimension of 360 zones, and z-dimension of 160 zones. Because the z-dimension is smaller than in Figure 15, the x-dimension size goes to a larger value. Both the Heterogeneous mode and the one MPI process per GPU mode utilize the GPU well. In this case, the MPS mode is unable to overlap kernels launched from different MPI ranks since they sufficiently occupy the GPU on their own, therefore the MPS performs worse in this set of experiments.

In Figure 17, we show problems with y-dimension of 480 zones, and z-dimension of 320 zones. Again, the x-dimension is relatively small for these problems, resulting in a performance gain from overlapping kernels launched from different MPI processes with computation for the MPS mode. The one MPI process per GPU mode is hampered by the small inner most loop dimension, while the Heterogeneous mode performs almost as well as the MPS mode.

In Figure 18, we show problems with y-dimension of 480 zones, and z-dimension of 160 zones. Because the z-dimension is smaller than in Figure 17, the x-dimension size goes to a larger value. Because the size of the y-dimension is large, and we partition off work for the CPU cores in the Heterogeneous mode in the y-dimension, this is the best case scenario for the Heterogeneous mode. When the one MPI process per GPU mode exceeds the memory bound, we see the Heterogeneous mode continue to scale linearly, resulting in up to 18% performance gain. Because the CPU performance is currently hampered by the compiler issue described in Section 5.1, we are only able to give 1-2% of work to the CPU cores. When the compiler issue is resolved, we expect to be able to give more work to the CPU, and see even better performance in this mode.

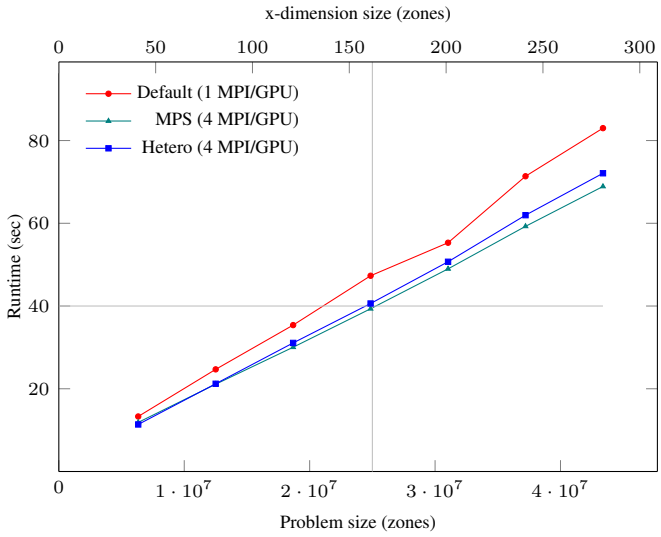

Figure 17: Varying the size of the x-dimension (y=480, z=320)

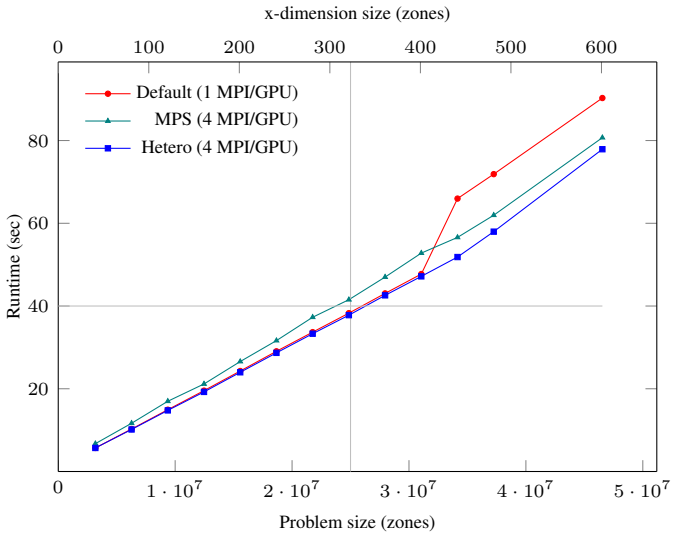

Figure 18: Varying the size of the x-dimension (y=480, z=160)

Overall, utilizing more than one MPI process per GPU can result in performance benefits, either with using MPS or using the Heterogeneous approach. The one MPI process per GPU version has the largest domains, and crosses a memory bound when problems become large. The mode utilizing MPS helps performance when x-dimension is relatively small by overlapping kernels launched from different MPI processes. The Heterogeneous approach is not useful when the dimension used to carve out work for the CPU is too small, but can be helpful otherwise and will likely become even more useful when the current compiler issues are resolved.

8 RELATED WORK

The paradigm of executing on both types of processors within a heterogeneous node has been explored in the context of a single node. A contention aware technique schedules unrelated OpenCL kernels on a heterogeneous node by taking into consideration historical performance, problem characteristics, and device status [11]. A heterogeneous IR query processing implementation uses queue-based runtime scheduling and applies different algorithms on the CPU and the GPU [9]. A heterogeneous FFT approach uses a performance model of both processors to schedule the load, while applying different libraries on the CPU and GPU [15]. A heterogeneous QR factorization implementation proposes a hybrid kernel which overlaps computation on both processors [4]; CPU-only, GPU-only, and hybrid kernels are then statically scheduled, or scheduled dynamically using StarPU runtime system [5].

For MPI applications, an MPI+OpenMP/CUDA parallel programming model has emerged as a viable approach to heterogeneous computing. In this model, a single MPI process is launched on each node, and OpenMP threads are spawned. One of the OpenMP threads is then reserved to interact with the GPU, while the rest are used for computation on the CPU. Examples include a long-wave radiation simulation [3]. The workload can be divided into independent columns and assigned to different processors using relative CPU/GPU power [13].

Several approaches have been proposed to split data-parallel tasks into sub-tasks and distribute across multiple devices. An OpenCL approach uses machine learning to partition the tasks into chunks [12]. An OpenMP proposal computes the iteration split between the two types of processors via a linear program [17]. Both are application-agnostic and therefore treat the iteration space as flat, which may or may not lead to the best performance.

Our approach is closest to the OpenCL and OpenMP approaches in that we use a performance-portable model and divide iterations between the different processors. Instead of relying on a runtime system, though, we decide on the sub-tasks within the application, similar to how the MPI applications typically divide work. We find that in 3D iteration spaces, the knowledge of the iteration space and the loop structure can help in decomposing the space optimally.

Most of the above approaches explore load balancing among the heterogeneous processes, stating that it is non-trivial. Even for runtime-driven load balancing, there is a trade-off between small work chunks which allow better balance, and large work chunks which the GPU is able to process faster by overlapping computation and communication (and streaming the remaining data) and pipelining [7]. Our approach is static within an iteration, but the decomposition can be adjusted between iterations, meaning that while we cannot adjust work immediately, we do not have the performance hit from scheduling chunks that are too small.

9 CONCLUSIONS

We proposed an approach for utilizing a heterogeneous node by using the GPUs and CPU cores cooperatively to perform loop computation. We presented a proof of concept implementation in ARES. Our implementation uses the same source code for the CPU and the GPU by using the RAJA portability layer for on-node parallelization. We divide work in the application via domain decomposition, and observe that using more than one MPI rank per GPU can result in significantly higher communication overhead. For both the approach using MPS, and for our Heterogeneous approach, we develop a hierarchical domain decomposition strategy which keeps the work per GPU the same as 1 MPI process per GPU approach. We then propose how to divide the work further for the MPS approach, and develop a load balancing strategy for our Heterogeneous approach. We compared performance of the one MPI process per GPU implementation, heterogeneous implementation, and 4 MPI processes per GPU with MPS. The heterogeneous implementation gets up to an 18% performance improvement over the default mode of executing ARES.

While load balancing between the CPUs and GPUs is non-trivial, we expect higher benefits from the Heterogeneous approach once the compiler issues currently hampering the CPU performance are resolved. Additionally, using more cores of the CPU may better utilize the CPU memory bandwidth, which is not fully tapped by a single MPI rank per GPU approach.

10 ACKNOWLEDGEMENTS

We would like to thank the RAJA developers and the ARES developers for providing the source code which became the basis for the development of our Heterogeneous approach.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-738086).

REFERENCES

[1] TOP500 Supercomputing Sites. In *http://www.top500.org*.

[2] NVIDIA Multi-Process Service (MPS). In *https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf*, May 2015.

[3] J. Agulleiro, F. Vzquez, E. Garzn, and J. Fernndez. Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction. *Ultramicroscopy*, 115:109 – 114, 2012.

[4] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 932–943, Washington, DC, USA, 2011. IEEE Computer Society.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Concurrency & Computation: Practice & Experience, Euro-Par 2009 best papers issue*, 2010.

[6] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: The architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press.

[7] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, Jan. 2013.

[8] R. Darlington, T. McAbee, and G. Rodrigue. A Study of ALE Simulations of Rayleigh-Taylor Instability. In *Computer Physics Communications*, volume 135, pages 58–73, 2001.

[9] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance ir query processing. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 421–430, New York, NY, USA, 2009. ACM.

[10] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[11] C. Gregg, J. Brantley, and K. Hazelwood. Contention-aware scheduling of parallel code for heterogeneous systems. In *USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, 2010.

[12] D. Grewe and M. F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 286–305, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] F. Lu, J. Song, X. Cao, and X. Zhu. Cpu/gpu computing for long-wave radiation physics on large gpu clusters. *Comput. Geosci.*, 41:47–55, Apr. 2012.

[14] B. E. Morgan and J. A. Greenough. Large-Eddy and Unsteady RANS Simulations of a Shock-Accelerated Heavy Gas Cylinder. In *Shock Waves*, April 2015.

[15] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. 2008.

[16] R. D. Hornung and J. A. Keasler. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, Sept 2014.

[17] T. R. Scogland, B. Rountree, W.-C. Feng, and B. R. de Supinski. Heterogeneous task scheduling for accelerated OpenMP. In *Parallel & Distributed Processing Symposium (IPDPS)*, May 2012.

[18] L. I. Sedov. Propagation of strong shock waves. In *Journal of Applied Mathematics and Mechanics*, volume 10, pages 241–250, 1946.