SEER: An on-node Performance-Portable C++ Library for Heterogeneous Systems*

Pedro Valero-Lara,*[0000-0002-1479-4310] and Jeffrey S. Vetter[0000-0002-2449-6720]

Oak Ridge National Laboratory, Oak Ridge, TN, 37830, USA
*Corresponding author: `valerolarap@ornl.gov`

Abstract. Heterogeneous and multi-device nodes are widely used in high-performance computing and data centers. However, current programming models do not provide simple, transparent, and portable support for automatically targeting heterogeneous nodes. In this paper, we present SEER, a new C++ library that provides a descriptive programming model to enable applications to benefit from heterogeneous nodes in a transparent and portable way across multiple device types. SEER provides efficient memory management and can select the proper device[s] depending on the computational cost of the applications. All this is completely transparent to the programmer, thereby providing a highly productive programming environment. We evaluate the SEER library on two heterogeneous nodes of Summit (#5 TOP500) and Crusher supercomputers. Notably, the smaller-scale Crusher test bed machine uses identical hardware and software as ORNL's Frontier (#1 TOP500). This work also includes a detailed performance study conducted with a set of representative test cases in high-performance computing (e.g., Basic Linear Algebra Subprograms (BLAS), Tridiagonal Solve, and Conjugate Gradient). SEER provides high accelerations of up to 30× for sparse matrix and 8× for batch BLAS applications thanks to automatic and transparent device selection and multi-device exploitation respectively.

Keywords: C++ · Metaprogramming · Performance Portability · Multi-Device · Heterogeneous.

1 Introduction

Hardware trends have seen a significant increase in the number and kind of devices (CPUs, GPUs, others) deployed in a single compute node. Examples include some of the fastest supercomputers in the world, such as Oak Ridge National Laboratory's (ORNL's) *Summit* and *Frontier* supercomputers. This trend is being driven by the attractive power-to-performance ratio provided by these new and specialized architectures and their usefulness for AI, high-performance computing (HPC), and scientific applications. As we continue to miniaturize chips, Moore's Law is expected to become obsolete [18], thus limiting the computational capabilities. Without a better technology solution, vendors have no choice but to develop more specialized architectures and accelerators to achieve cost-effective performance and scalability, and this push to specialized hardware has likewise increased the complexity of HPC and the scientific codes that leverage HPC. In the future, this trend is only expected to increase.

Current programming models, such as the C++ libraries Kokkos or RAJA [23, 1] cannot automatically and transparently target multiple devices on a heterogeneous node. This limitation decreases programming productivity. In this paper, we describe a novel solution that targets performance portability when using multiple devices within a heterogeneous node. This novel solution is a transparent (i.e., architecture agnostic),

* This research used resources from the Oak Ridge Leadership Computing Facility and the Experimental Computing Laboratory at ORNL, which is supported by the US Department of Energy's (DOE's) Office of Science under Contract No. DE-AC05-00OR22725. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DOE's Office of Science and the National Nuclear Security Administration.
This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the DOE. The publisher, by accepting the article for publication, acknowledges that the US Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of the manuscript or allow others to do so, for US Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

multi-device, and performance-portable C++ library called SEER. The goal of this work is to create a solution that allows programmers to focus on the applications and implementations by abstracting architecture- and vendor-specific details in our programming model, thereby providing a highly productive programming environment for the end-use developer. The primary contributions of this work are (1) a transparent (i.e., architecture-agnostic), performance-portable, and multi-device programming model for heterogeneous systems; (2) a transparent and intelligent autotuning solution that decides the best device[s] to use at run time depending on hardware and application characteristics; and (3) a transparent and efficient memory management system for heterogeneous nodes.

The rest of the paper is organized as follows: Section 2 presents the main characteristics of the C++ SEER library and programming model. We describe the performance analysis in Section 3. Related work is covered in Section 4. Finally, the conclusions and future directions are summarized in Section 5.

2 The SEER Programming Model

The SEER C++ library (Figure 1) is divided into three main modules: memory (memory management), compute (parallel constructs), and autotuning (device[s] selection). Although the autotuning module is common for the different back ends, the other modules (i.e., memory and compute) have different and separate implementations depending on the back end. We implemented four different back ends, OpenMP, CUDA, HIP, and OpenMP Target, which correspond to the most relevant parallel programming models used in current and upcoming HPC architectures (e.g., multicore CPUs and NVIDIA, AMD, and Intel GPUs).

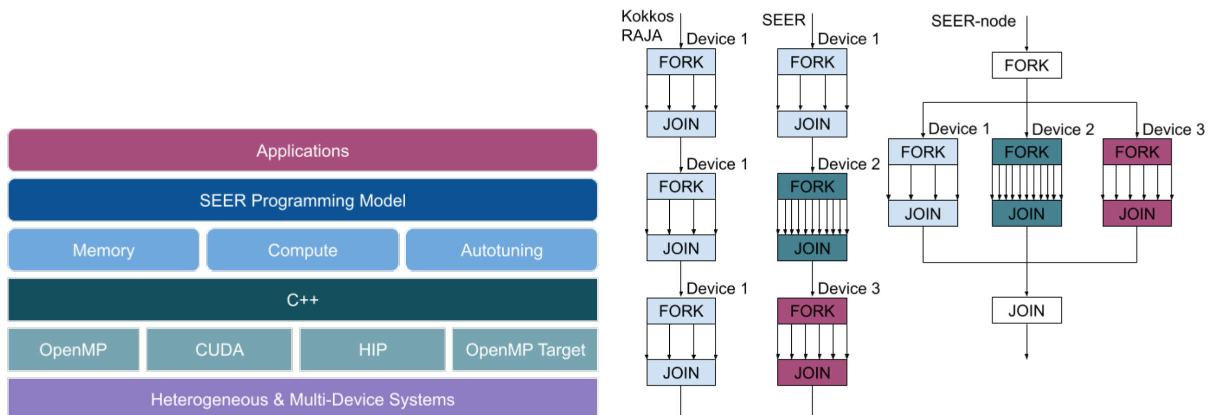

Fig. 1. SEER software stack (left) and SEER and SEER-node fork-join models (right).

SEER differs from other existing data-parallel C++ solutions, such as Kokkos [23], RAJA [1], or SYCL [19] in four main ways. (1) Instead of the programmer specifying which device to use at compilation time [23, 1], SEER decides in a transparent way which devices (i.e., back ends) to use at run time. (2) The devices to be used can change along the execution of the application depending on the computational cost of the operations carried out and the hardware characteristics. (3) Different devices and/or back ends can coexist and be used simultaneously, thereby providing a real heterogeneous and multi-device solution. (4) In SEER, the computation is defined as functions. These functions are written in C++ and are compatible for all back ends; OpenMP (CPUs), OpenMP Target (Intel GPUs), CUDA (NVIDIA GPUs), and HIP (AMD GPUs).

To better illustrate the capacity of the SEER library, Figure 1 depicts an example of the fork-join model leveraged by SEER and SEER-node compared with the existing Kokkos [23], RAJA [1] and SYCL [19] models. As with the other models, a SEER code can be seen as a set of parallel constructs, such as `parallel_for` or `parallel_reduce`. Each of these constructs exploits a fork-join model. In SEER, we can use different

devices during the execution, which is completely transparent from the programmer's point of view. A different granularity can be exploited by each of the parallel constructs depending on the characteristics of the hardware.

Also, we implemented SEER-node that enables SEER applications with the capacity to use more than one device in parallel. Applications that can compute multiple parallel constructs on different devices and separate memory spaces in parallel can benefit from the use of SEER-node.

In the following subsections, we describe in more detail the three main components of the SEER library, as well as the singularities of the SEER-node extension to enable SEER with the capacity to use multiple devices in parallel.

2.1 Memory

```
int SIZE = 2000;
seer::memory *mem_axpy;
seer::memory *mem_dot;

mem_axpy = (seer::memory*) malloc (sizeof(seer::memory));
mem_axpy->num_array = 2;
mem_axpy->size_array[0] = SIZE * sizeof(double);
mem_axpy->size_array[1] = SIZE * sizeof(double);
mem_axpy->num_reduction = 0;
seer::create_mem( mem_axpy, 0 );

mem_dot = ( seer::memory* ) malloc (sizeof(seer::memory));
mem_dot->num_array = 2;
mem_dot->size_array[0] = SIZE * sizeof(double);
mem_dot->size_array[1] = SIZE * sizeof(double);
mem_dot->num_reduction = 1;
mem_dot->size_reduction[0] = 1 * sizeof(double);
seer::create_mem(mem_dot, 1);

// Copy array 0 of the seer memory object mem_dot to array 1 of seer memory object mem_axpy
seer::mem_copy( mem_axpy, 0, 1, mem_dot, 1, 0 );

seer::free_mem(mem_axpy, 0);
seer::free_mem(mem_dot, 1);
```

Fig. 2. Example of SEER memory management.

The memory management in the SEER library is completely transparent to the programmer. The programmer does not have to know which device's memory (memory space) is being used or if necessary memory transfers between CPU and GPUs or between GPUs are conducted—this is all taken care of without their intervention. A seer memory object is composed of many memory spaces, as many as the number of devices available. These memory spaces associated with a seer memory object are managed in a transparent way to the programmer. The functions will use the corresponding memory space associated with the device where the function will be executed (see `seer::space` in Figure 3). A SEER application can use as many seer memory objects as necessary, and it is generally recommended to use a different seer memory object per parallel construct that could be computed in parallel. This allows the use of multiple devices and reduces unnecessary data transfers between devices. We can define a seer memory object by using `seer::memory`. This SEER type has different attributes that must be defined before creating a seer memory object, such as `num_array` and `num_reduction`, which defines the number of arrays and reductions that we will need in our

application, respectively. We specify the size of the arrays with `size_array[x]` and the size of the results of the reductions with `size_reduction[x]`. Finally, we can create a memory object with `create_mem` and free a memory object with `free_mem`. For debugging, as a last argument, we use an integer to identify the memory objects created. We can also transfer data between different memory objects by using this identification. Figure 2 shows how to use SEER memory objects.

The memories (memory spaces) of the different devices (CPUs and GPUs) are managed dynamically according to the decisions taken by SEER library, which depends on the features of the hardware and connectivity, and the computational cost of the applications. SEER can use different device memories (memory spaces) for the same application in a transparent way, if the needs or computational cost of the applications change along the execution, as we show in Section 3. If SEER decides to change the target device along the execution, then the data is transferred from one device to the other without user intervention.

On the other hand, it may be necessary for the applications to copy one specific array of one memory object to a different memory object. To do that, we can use the `seer::mem_copy` primitive (Figure 2).

We use the corresponding vendor primitives to carry out the allocations on different device memories or back ends: `cuda/hipMalloc` for NVIDIA and AMD GPUs and `omp_target_alloc` for Intel GPUs. For the inter-device communication, we use `omp_target_memcpy` on the OpenMP Target back end and `cuda/hipMemcpy` and `cuda/hipMemcpyPeer` (for GPU-to-GPU data transfers) for NVIDIA and AMD GPUs. We do not use any special capacity, such as unified memory, instead, we want to keep a higher control on the decisions corresponding to the data movement between devices, which can be controlled by the user manually or by SEER transparently. The use of vendor- or specific- routines for GPU-GPU communication helps to exploit high-bandwidth networks that connect these devices.

2.2 Compute

SEER has two primary data-parallel constructs, `parallel_for` and `parallel_reduce`, as depicted in Figure 4. Parallel SEER constructs are composed of five main components: (1) the number of iterations of the for-loop or reduction, which is typically equal to the size of the arrays; (2) parameters used in the function, which must be defined previously; (3) memory space used in the function (see previous subsection); (4) the tuning factor, which is typically equal to the number of operations to be computed by the function. This is used by the autotuning module to decide which device to use; and finally (5) the function pointer that defines the operations to be computed in each iteration of the loop. Like the second parameter, this last parameter must be defined in advance. For `parallel_reduce`, the second parameter corresponds to the identification of the reduction because one memory space can store the result of multiple reductions. Some details on the parallel implementation of the different back ends, such as the grid size and block size for CUDA/HIP implementations, are transparent to the users.

Users must provide the same function for the different back-ends (OpenMP, CUDA, HIP, and OpenMP Target). Figure 3 shows an implementation of two CUDA functions, as well as the parameters, used in the parallel constructs depicted in Figure 4. The implementation consists of two Basic Linear Algebra Subprograms (BLAS) level-1 routines: AXPY and DOT product.

These constructs are templated according to the data type of the parameters to be passed to the function and the function type. For `parallel_reduce`, we have one additional templated parameter to indicate the type used to store the result of the reduction (see `double` in Figure 3). As mentioned, every SEER parallel construct has a different implementation or mapping for each back end. Figure 5 shows part of the implementations of the `parallel_for` construct on the different back ends.

The code for transferring the function from CPU memory to NVIDIA or AMD GPU memory is not included in Figure 5. This action is implemented via `cuda/hipMemcpyFromSymbol`.

2.3 Autotuning

The autotuning module is in charge of device selection for the different parallel constructs. This process is also completely transparent to the programmer. Estimating the time for the different parallel constructs

```
struct paxpy {double alpha;};
struct pdot {};

typedef void(*axpy_func) (int ind, paxpy params, seer::space data_space);
typedef void(*dot_func) (int ind,  pdot params, seer::space data_space, double &tmp);

__device__ void axpy(int ind, paxpy param, seer::space mem)
{
  double* Y = (double*) mem.array[0];
  double* X = (double*) mem.array[1];
  Y[ind] += param.alpha * X[ind];
}
__device__ void dot(int ind, pdot param, seer::space mem, double &tmp)
{
  double* X = (double*) mem.array[0];
  double* Y = (double*) mem.array[1];
  tmp += X[ind] * Y[ind];
}
```

Fig. 3. Example of SEER functions and parameters.

```
seer::parallel_for <paxpy, axpy_func> ( SIZE, paxpy, mem_axpy, SIZE * 2.0, axpy );
seer::parallel_reduce <pdot, dot_func, double> ( SIZE, 0, pdot, mem_dot, SIZE * 2.0, dot );
```

Fig. 4. SEER data-parallel constructs.

is a complex task, due to the particularities of such operations, synchronizations, data movements, among others overheads or latencies [29]. The model used to estimate the time (see Equation 1) is based on the work described by [24], which was proven to be very effective for automatic device (CPU-GPU) selection. This process is carried out every time a parallel construct is called.

$$Time = \frac{Tuning\ Factor}{Device\ Capacity} + Communication\ Time \quad (1)$$

We must know the tuning factor to efficiently select the proper device to use. In the test cases evaluated in this paper, we used the number of operations to be computed by the parallel constructs as a tuning factor. However, it is also possible to use other characteristics of the applications, such as the size of the matrix or the number of non-zero elements, among others. The tuning factor is passed as an argument as the fourth parameter of `parallel_for` and the fifth parameter of `parallel_reduce`. It is also very important to know the characteristics of the different devices available in our system and how these are connected. To do that, the programmer can provide a set of flags at compilation time to identify the CPU (`SEER_CPU_ARCH`) and GPU (`SEER_GPU_ARCH`) architectures, the number of GPUs (`SEER_GPU_NUM`), or how the CPU and GPU are connected (`SEER_CPU_GPU_NET_ARCH`). Other important hardware features, such as memory bandwidth and the number of cores are implicit in these flags. Then, the first term corresponds to the ratio between `Tuning Factor` and `Device Capacity`. This last is a combination of the performance provided by the different devices (CPUs and GPUs), in terms of GFLOPs, and other components, such as the memory bandwidth of the different devices and the latency (CPU-GPU and GPU-GPU communication bandwidth) of the networks connecting the different devices.

The *Communication Time* factor is considered only if it is necessary to make any data transfer between devices. We keep the information about the location of every memory space along the life cycle of the applications. So, the decision about which device to use also depends on the overhead corresponding to the memory transfer between different devices. This term basically estimates the necessary time to transfer the data used by the parallel construct between the different devices, taking into account the bandwidth and latency of the network topology of our heterogeneous system.

```
//CUDA and HIP kernel
__global__
void kernel_parallel_for(int n, params p, seer::space *mem, function* f){
 int ind = threadIdx.x + blockIdx.x * blockDim.x;
 if ( ind < n ){
  seer::space data_dev;
  data_dev.array = data_space->array;
  (*f)( ind, p, data_dev );
 }
}
//SEER parallel for implementations
void parallel_for(int n, params p, seer::memory *mem, function f){
 //OpenMP
 #pragma omp parallel for shared(mem) num_threads(num_threads)
 for ( int i = 0; i < n; i++ ){
  f( i, p, mem->arr_host);
 }
 //OpenMP Target
 #pragma omp target teams distribute parallel for \
         map(to:f) map(mem->arr_device[dev_id])
 for ( int i = 0; i < n; i++ ){
  f( i, p, mem->arr_device[dev_id] );
 }
 //CUDA
 kernel_parallel_for<<< num_blocks, block_size >>> ( n, p, &mem->arr_device[dev_id], f );
 cudaDeviceSynchronize();
 //HIP
 hipLaunchKernelGGL ( kernel_parallel_for, num_blocks, block_size , 0, 0, n, p,
↪  mem->arr_device[dev_id], f );
 hipDeviceSynchronize();
}
```

Fig. 5. Pseudocode of the `parallel_for` construct implementations on the different back ends.

2.4 SEER-node

The SEER-node enables SEER with the capacity to use more than one device in parallel (as shown in Figure 1) without increasing the complexity of the code and potentially achieving better performance. The SEER-node extends the SEER model in two aspects: (1) it adds an identification as part of the parameters of the parallel constructs, and (2) it creates a simple mechanism that allows the programmer to express this higher level of parallelisms in an agnostic and simple manner by using two macros (`SEER_FORK(N)` and `SEER_JOIN()`). Also, to distinguish between SEER and SEER-node primitives, the names of the parallel constructs are different (e.g., `parallel_for_node` and `parallel_reduce_node`) and they keep the same interface for memory management.

These new parallel constructs must be encapsulated inside a fork-join macro. This macro is defined by using `SEER_FORK(N)` and `SEER_JOIN()` primitives, where N is the parallel degree to be exploited. This degree can be associated to the characteristics of the application, how many operations (i.e., parallel constructs) can be run by the application in parallel, or the number of devices in our system (i.e., devices or compute units in our system). So this factor is limited by either the application or the hardware characteristics. Figure 6 illustrates a simple example by using the parallel constructs shown in Figure 4. Note that for better performance, it is better to use a different memory space in each of the parallel constructs, which can be computed in parallel.

SEER-node is implemented with OpenMP nesting. The number of OpenMP threads used in the upper level is defined by the parameter used in `SEER_FORK(N)`, as we can see in Figure 6. We have different

```
SEER_FORK(2)
seer::parallel_for_node <axpy_params, axpy_function> ( SIZE, paxpy, mem_space_0, axpy_op, axpy, 0
↪  );
seer::parallel_reduce_node <dot_params, dot_func, double> ( SIZE, 0, pdot, mem_space_1, dot_op,
↪  dot, 1 );
SEER_JOIN()
```

Fig. 6. Example of SEER-node data-parallel constructs and macros.

implementations depending on the back ends used in the different parallel constructs. For the CUDA and HIP implementations, every OpenMP thread is associated with a GPU. This is possible thanks to the OpenMP + CUDA/HIP interoperability and the `cuda/hipSetDevice` functions. For the OpenMP Target back end, the implementation is very similar to the CUDA/HIP implementations, but instead of using `cuda/hipSetDevice`, we use `openmp_target_set_device`. Finally, for the OpenMP back end, we split the number of OpenMP threads (number of cores available) into the different primary OpenMP threads.

3 Performance Evaluation

We used ORNL's Summit and Crusher systems for the performance evaluation (Table 1). Summit is equipped with two POWER9 CPUs and four NVIDIA V100 Volta GPUs per node, and Crusher has one AMD EPYC 7A53 CPU and four AMD MI250X GPUs, each with 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node.

We evaluate SEER by using four well-known and characteristic test cases that correspond to some of the most popular and widely used computational operations in HPC and scientific computing: BLAS level-1 AXPY, Tridiagonal Solve, Conjugate Gradient (CG), and Batched BLAS. We decided to use these test cases because their characteristics are representative of real scenarios, which will help us evaluate the different features and capabilities of the SEER library, including transparent device selection depending on the computational cost of the operations and hardware characteristics (AXPY), management of dynamic parallelism with transparent data transfer and device selection (Tridiagonal Solve), exploitation of a higher level of parallelism when some parts of the applications can be computed in parallel (CG), and full use of all the resources available in our heterogeneous and multi-device system (Batched BLAS).

3.1 BLAS Level-1 AXPY (Transparent Device Selection)

For our first test case, we use the AXPY operation, which computes a scalar-vector product and adds the result to a vector. This is a simple and well-known BLAS Level-1 operation used in multiple applications and benchmarks [22, 3].

Figure 7 shows the run time for only one CPU (OpenMP) and one GPU (CUDA on Summit and HIP on Crusher in Figure 7) vs. the run time of SEER, which decides which device to use depending of the application and hardware characteristics. As shown, although we see a similar trend in both systems, the result is different in terms of which device is used depending on the size of the vectors. On Summit, SEER decides to use the CPU for vector sizes smaller than 200,000 and use the GPU for bigger sizes. However, on Crusher, the CPU is used for vector sizes smaller than 400,000, and the GPU is used for bigger vector sizes. This is due to the differences in the systems. We see that similar performance is achieved when comparing SEER with OpenMP (i.e., only the CPU is used) or with CUDA/HIP (i.e., only the GPU is used). However, we see a relatively higher execution time when using SEER compared with CUDA on SUMMIT. The main difference between these two codes is that SEER makes use of autotuning to predict the time and make the device selection. In this case, this overhead is about 7% with respect to CUDA. This may be related to the computational cost of this operation, as well as the characteristics of the hardware. We do not see such an overhead when running the same test case on Crusher, and in the other test cases presented below. In most of the cases, this overhead represents less than $< 1\%$ of the total execution time.

System	Summit	Crusher	System	Summit	Crusher
—CPU Architecture—			—GPU Architecture—		
CPU	IBM Power9	AMD EPYC 7A53	GPU	4×NVIDIA V100	4×AMD MI250X 8×GCDs
—Connectivity—			—Compiler—		
GPU-to-GPU	NVLink 2.0 (50 GB/s)	Infinity Fabric (100 GB/s)	CPU	xlc V16	amdclang 14
GPU-to-CPU	NVLink 2.0 (50 GB/s)	Infinity Fabric (36 GB/s)	GPU	nvcc V11.0.221	hipcc 5.1
—SEER flag—			—Compiler flag—		
SEER_CPU_ARCH	IBM_POWER_9	AMD_EPYC_7A53	CPU	-mcpu=power9	-march=znver3
SEER_GPU_ARCH	NVIDIA_V100	AMD_MI250X		-mtune=power9	-mtune=znver3
SEER_NET_ARCH	NVLINK_2	NFINITY_FABRIC	GPU	-arch=sm_70	–amdgpu-target
SEER_GPU_NUM	4	8			=gfx90a
—OpenMP setting—					
export OMP_NESTED=true export OMP_PLACES=cores export OMP_PROC_BIND=(spread,close)					

Table 1. Summary of the Summit and Crusher configurations.

Also, we see that SEER is able to select the best device to use according to the application requirements. Thanks to automatic and transparent device selection, SEER provides accelerations of up to 2.5× and 3.2× on Summit and Crusher respectively. The speedup is computed as the ratio between the slowest reference time (either using a CPU [OpenMP] or a GPU [CUDA/HIP]) and SEER time.

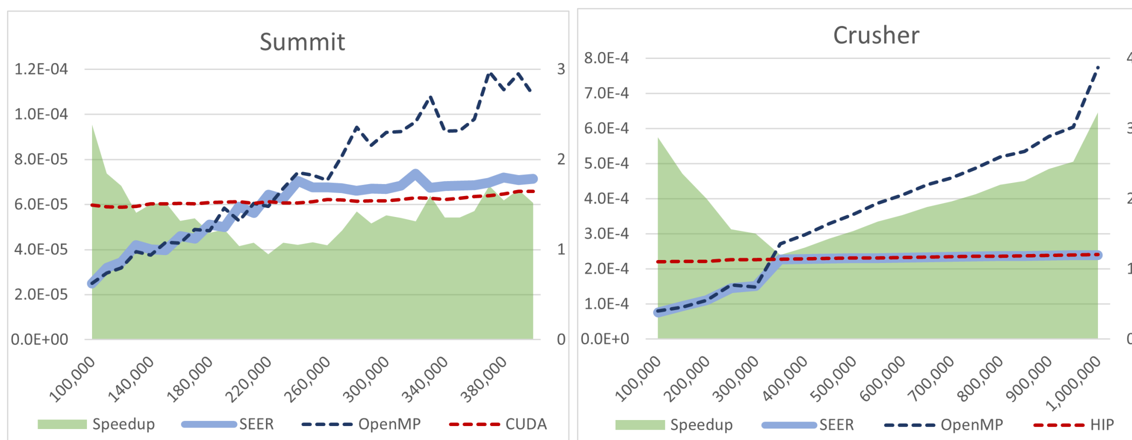

Fig. 7. Execution times (left-side y-axis, seconds) and speedup (right-side y-axis) of the SEER AXPY implementation on Summit (left) and Crusher (right) when increasing the size of the vectors (x-axis). The speedup is computed as the ratio between the slowest reference time (either using a CPU [OpenMP] or a GPU [CUDA/HIP]) and SEER time.

3.2 Tridiagonal System Solver (Dynamic Parallelism with Transparent Data Transfer and Device Selection)

The resolution of tridiagonal linear systems [33, 34] is required in many problems of industrial and scientific interest. Examples include alternating-direction implicit methods, Poisson solvers [33, 34], cubic spline

approximations, numerical ocean models [9], preconditioners for iterative linear solvers [8], and the simulation of the human brain [31], among many others. Usually, solving tridiagonal systems takes most of the computation time of these applications.

The state-of-the-art method to solve tridiagonal systems sequentially is the Thomas algorithm [33, 34], which is a specialized application of the Gaussian elimination that leverages the tridiagonal structure of the system. It consists of two stages commonly denoted as forward elimination and backward substitution.

The algorithm solves a linear $Au = y$ system, where A is a tridiagonal matrix:

$$A = \begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \cdot \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix}.$$

Note that the data structures required by this algorithm are three arrays (a, b and c) of size n that represent the three diagonals of the input matrix and two additional vectors of the same size, u and y, that store the unknowns of the equation (to be calculated) and the right-hand terms of the equation, respectively. The implementation of this algorithm does not require the u array because the result is overwritten in the array y, but we have included u for clarity.

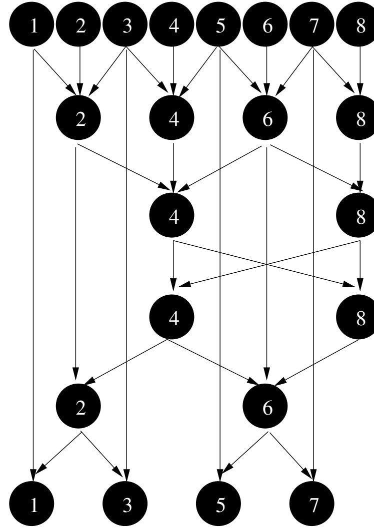

Fig. 8. Cyclic reduction (CR) computational pattern.

The algorithm used in this study is the cyclic reduction (CR) [33, 34], which is a parallel alternative to the Thomas algorithm. CR consists of two phases: reduction and substitution. In each intermediate step of the reduction phase, all even-indexed (i) equations, $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$, are reduced. The values of a_i, b_i, c_i, and d_i are updated in each step according to

$$a'_i = -a_{i-1}k_1, b'_i = b_i - c_{i-1}k_1 - a_{i+1}k_2 \; c'_i = -c_{i+1}k_2, y'_i = y_i - y_{i-1}k_1 - y_{i+1}k_2$$
$$k_1 = \frac{a_i}{b_{i-1}}, k_2 = \frac{c_i}{b_{i+1}}.$$

After $\log_2 n$ steps, the system is reduced to a single equation that is solved directly. All odd-indexed unknowns, x_i, are then solved in the substitution phase by introducing the already computed u_{i-1} and u_{i+1} values:

$$u_i = \frac{y'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}.$$

Overall, the CR algorithm needs $17n$ operations and $2\log_2 n - 1$ steps, which is the optimal algorithm in terms of operations to compute tridiagonal systems. Figure 8 depicts the computational pattern.

This is a complex application to parallelize with two different regions to target for parallelism. One region corresponds to the beginning of the first step (reduction) and the end of the second step (substitution) and has a high level of parallelism, where the use of a GPU can be beneficial. The other region corresponds to the end of the reduction step and the beginning of the substitution step and has a low level of parallelism, where the use of a CPU is more effective (Figure 8). The characteristics of the hardware and the parallelism define the boundaries of these areas. As shown in Figure 9, SEER can detect these areas and select the appropriate device to use. Again, although the trend is similar in both systems, they still exhibit different behavior due to certain differences. In this case, the difference is in the number of reduction/substitution steps in which the CPU/GPU is used. The GPU is used in fewer reduction and substitution steps on Crusher than on Summit.

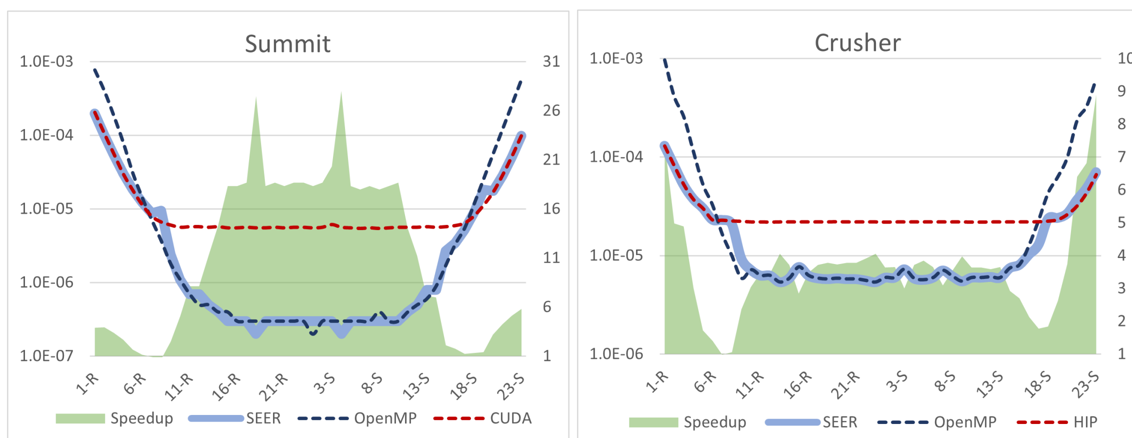

Fig. 9. Execution time in logarithmic scale (left-side y-axis, seconds) and speedup (right-side y-axis) of the SEER Cyclic Reduction algorithm implementation on Summit (left) and on Crusher (right), step by step (x-axis). In the x-axis, R and S mean the Reduction and Substitution steps respectively. The speedup is computed as the ratio between the slowest reference time (either using a CPU [OpenMP] or a GPU [CUDA/HIP]) and SEER time.

In this case, the performance of both architectures (CPU and GPU) is very different depending on the step of the CR algorithm. So, the benefit of using SEER with automatic and transparent device selection provides high accelerations of up to 30× on Summit and 9× on Crusher. Also, while the highest acceleration is found in the last (first) steps of the reduction (substitution) steps on Summit, we see the opposite scenario on Crusher. This is due to the differences between both systems. Once again, we see that our library is able to make good decisions depending on both, hardware features and application requirements, reaching high performance in each of the steps of the algorithm.

3.3 Conjugate Gradient (Transparent Multi-Device Exploitation Limited by Algorithm Parallelism)

Conjugate Gradient (CG) is a well-known and widely used iterative method for solving sparse systems of linear equations. These systems appear in finite difference and finite element methods, partial differential equations, structural analysis, circuit analysis, and much more linear algebra-related problems [21]. Owing to the importance of this operation, it is also used to benchmark supercomputer performance. The HPCG

library[1] offers such a benchmark and is optimized for distributed-memory architectures and implemented in the Message Passing Interface (MPI) and OpenMP. HPCG is used twice a year to update the TOP500 HPCG list,[2] which ranks the fastest supercomputers in the world by their CG computing performance.

Owing to the particular characteristics of the CG method, these kinds of problems can benefit from using SEER-node. Some of the most important and time-consuming steps (parallel constructs) of this method may be efficiently parallelized, thereby considerably reducing the execution time and making them benefit more from additional devices. We implemented the plain CG algorithm [22, 3] without a precondition. This simplifies the study of the optimizations thanks to the elimination of the preconditioning step, which will be considered in future versions of the code. We must also ensure the convergence of the method by using an appropriate input matrix. To this end, we generate a diagonal dominant tridiagonal sparse matrix, which is commonly used in these contexts (e.g., in the HPCG benchmark).

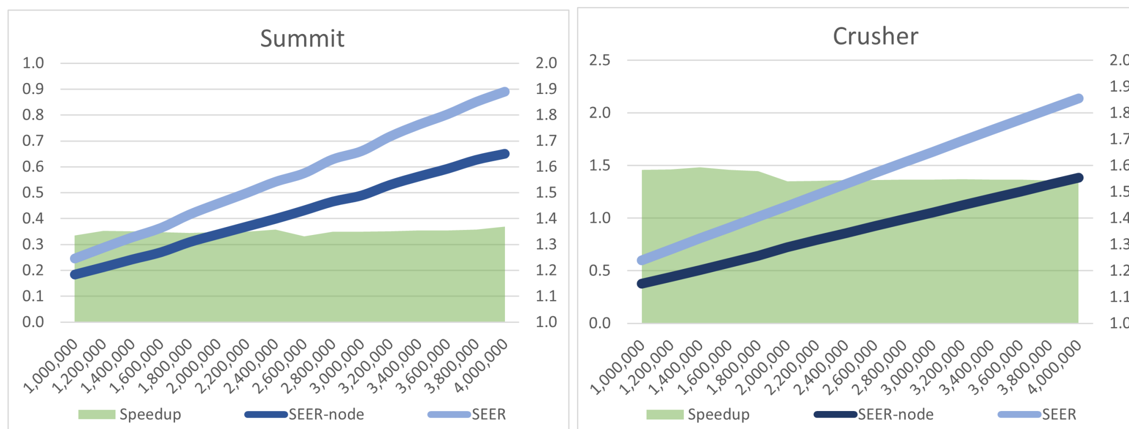

Fig. 10. Execution time in seconds (left-side of the graphs) and speedup (right-side of the graphs) achieved by SEER-node over SEER for the SEER-node implementation of the CG method on Summit (left) and on Crusher (right) with increasing problem sizes.

This is a good example of an application for which the degree of parallelism is limited by the application itself. Although some parts of the CG algorithm can be parallelized, these can only be computed in parallel selectively with certain other parts of the algorithm due to the data dependencies between them. Several pairs of DOT products or AXPY operations can be computed in parallel on different devices. This can be expressed by using SEER-node macros. We must also use two different memory spaces to efficiently exploit a higher level of parallelism.

Figure 10 shows the run time of SEER and SEER-node for CG computation. As shown, the SEER-node optimization achieves a 1.6× speedup on Crusher and a 1.4× speedup on Summit when using two devices w.r.t. the use of one single device (SEER). These speedups represent significant acceleration given the application's relatively limited higher-level parallelism. Notably, given the array size of the operations, only GPUs were used in the tests.

3.4 Batched BLAS (Transparent Multi-Device Exploitation Limited by Hardware Components)

Batched BLAS is a current trend in HPC in which a large linear algebra problem is decomposed into batches that contain many sub-problems that can be solved independently before collating the results [5,

[1] http://www.hpcg-benchmark.org/
[2] https://www.top500.org/lists/hpcg/

30]. Some applications include multifrontal solvers for sparse linear systems [6], tensor contractions for deep learning [4], human brain simulation [31], astrophysics, low rank matrix computations [32], computational fluid dynamics [34], and image among many others [17]. Although most of the literature focuses on using one device at a time, we were able to implement a simple multi-device Batched BLAS code by using the SEER-node model.

Unlike the previous test case, Batched BLAS is a clear example of parallelism being limited by the hardware. It is very common to have many independent problems in batched BLAS, and the number of problems that can be computed in parallel depends on the number of devices available to compute them (i.e., four GPUs on Summit and eight GCDs on Crusher). The code modifications required to leverage more than one device are straightforward. The modification consists of adding two lines of code, one for `SEER_FORK(X)` and one `SEER_JOIN`, and using the SEER-node primitives for the data-parallel constructs (see Section 2.4). In this case, `X` is the number of devices in our system.

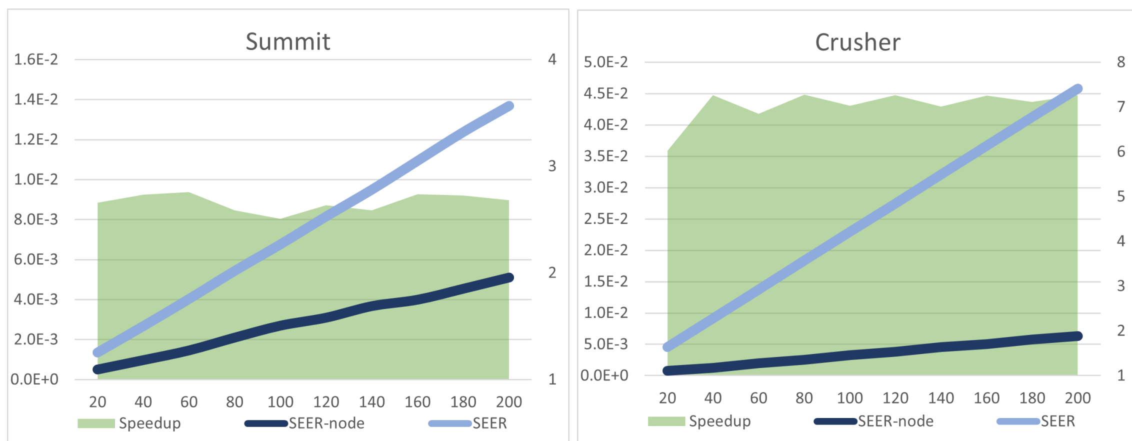

Fig. 11. Execution time in seconds (left-side of the graphs) and speedup (right-side of the graphs) achieved by SEER-node against SEER for the SEER-node implementation of the Batched BLAS AXPY routine on Summit (left) and on Crusher (right) with increasing batch sizes.

Figure 11 shows the performance gained by using SEER-node. As shown, the acceleration is close to the theoretical peak: 3× speedup on Summit and 7× speedup on Crusher.

4 Related Work

Historically, heterogeneous computing has involved a single CPU and a single GPU [34]. Moreover, most efforts were optimized for one application and one particular heterogeneous system [25, 26], and the programmer had to decide where to run each part of the code, thereby making this a very demanding and unproductive programming solution.

Although deploying multiple device types is becoming the norm for many HPC hardware configurations, an important gap exists in the so-called *high-level* programming models to properly support heterogeneous implementations. We can divide high-level programming models into two major groups: (1) those based on pragmas (e.g., OpenMP, OpenACC) and (2) those based on C++ abstraction libraries (e.g., Kokkos [23], RAJA [1], SYCL).

OpenMP recently incorporated a new list of pragmas for GPUs into its specification [27]. These new pragmas allow for offloading the execution onto one GPU by using the OpenMP `target` clause. However, no OpenMP construct currently targets multiple GPUs. Another important reference about the exploitation of heterogeneous systems by using pragma-based programming solution is OmpSs [12, 2]. Also, task-based

runtimes like IRIS [28, 15, 16], XiTAO or Cpp-Taskflow [10] provides a good support for heterogeneous systems. Similarly, SYCL provides an interface that allows the programmer to use either the CPU or the GPU by using different queues.OpenMP, SYCL, and other runtimes force programmers to decide which device to use, and having to choose makes it difficult to develop portable codes for different heterogeneous configurations.

Typically, using multiple devices with OpenACC [22] has required MPI [13]. Matsumura et al. [14] developed and proposed an extension to the OpenACC specification to handle multiGPU systems, and they argued that their extension was competitive with some MPI + OpenACC codes on NVIDIA multiGPU systems. Unfortunately, this extension was not adopted into the OpenACC standard.

The Kokkos team continues to develop new and important features and optimizations that target performance portability among different architectures, including memory management [7], vectorization [20], supporting new back-ends, such as SYCL or OpenACC [29]. Kokkos can be successfully integrated or combined with other programming models such as MPI [11] and HPX, among others [35]. However, in both Kokkos and RAJA, the target architecture must be defined at compilation time when the user specifies the target architecture (e.g., with the `KOKKOS_DEVICES` flag in Kokkos). One must use `KOKKOS_DEVICES = Cuda` to generate a binary for NVIDIA GPUs. Also, only a single device can be used. The MPI + Kokkos interoperability is the current de facto solution for multiple GPUs in Kokkos. Once again, the programmer is responsible for deciding which device(s) to use in these solutions.

To the best of our knowledge, the work described in this paper is the first time that a fully architecture-agnostic, multi-device, and performance-portable programming model has been implemented for heterogeneous systems.

5 Conclusions & Future Work

We implemented SEER, a novel C++ performance-portable and descriptive programming model that targets heterogeneous configurations and can handle device selection and data exchange among different devices automatically and transparently. We have demonstrated the different features of SEER on two heterogeneous nodes from ORNL's Summit and Crusher supercomputers for four well-known and characteristic HPC test cases.

For future work, we plan to (1) use additional architectures (i.e., not just CPUs and GPUs), (2) implement other back ends based on other programming models (e.g., SYCL) or other run times, (3) extend the current capacity to more than one device when computing a single parallel construct, and (4) provide new capacities for better, transparent, and portable exploitation of top-level memory hierarchies on n-dimensional arrays, among other efforts.

References

1. Beckingsale, D., Hornung, R.D., Scogland, T., Vargas, A.: Performance portable C++ programming with RAJA. In: Hollingsworth, J.K., Keidar, I. (eds.) Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019. pp. 455–456. ACM (2019). https://doi.org/10.1145/3293883.3302577
2. Bosch, J., Tan, X., Filgueras, A., Vidal, M., Mateu, M., Jiménez-González, D., Álvarez, C., Martorell, X., Ayguadé, E., Labarta, J.: Application acceleration on fpgas with ompss@fpga. In: International Conference on Field-Programmable Technology, FPT 2018, Naha, Okinawa, Japan, December 10-14, 2018. pp. 70–77. IEEE (2018). https://doi.org/10.1109/FPT.2018.00021, https://doi.org/10.1109/FPT.2018.00021
3. Catalán, S., Martorell, X., Labarta, J., Usui, T., Díaz, L.A.T., Valero-Lara, P.: Accelerating conjugate gradient using ompss. In: 20th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019, Gold Coast, Australia, December 5-7, 2019. pp. 121–126. IEEE (2019). https://doi.org/10.1109/PDCAT46702.2019.00033, https://doi.org/10.1109/PDCAT46702.2019.00033
4. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning (2014). https://doi.org/10.48550/ARXIV.1410.0759, https://arxiv.org/abs/1410.0759

5. Dongarra, J.J., Hammarling, S., Higham, N.J., Relton, S.D., Valero-Lara, P., Zounon, M.: The design and performance of batched BLAS on modern high-performance computing systems. In: Koumoutsakos, P., Lees, M., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M.A. (eds.) International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. Procedia Computer Science, vol. 108, pp. 495–504. Elsevier (2017). https://doi.org/10.1016/j.procs.2017.05.138, https://doi.org/10.1016/j.procs.2017.05.138
6. Duff, I.S., Reid, J.K.: The multifrontal solution of indefinite sparse symmetric linear. ACM Trans. Math. Softw. **9**, 302–325 (1983)
7. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distributed Comput. **74**(12), 3202–3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003
8. Greenbaum, A.: Iterative Methods for Solving Linear Systems. Society for Industrial and Applied Mathematics (1997)
9. Halliwell, G.R.: Evaluation of vertical coordinate and vertical mixing algorithms in the hybrid-coordinate ocean model (hycom). Ocean Modelling **7**(3–4), 285 – 322 (2004)
10. Huang, T., Lin, Y., Lin, C., Guo, G., Wong, M.D.F.: Cpp-taskflow: A general-purpose parallel task programming system at scale. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **40**(8), 1687–1700 (2021). https://doi.org/10.1109/TCAD.2020.3025075, https://doi.org/10.1109/TCAD.2020.3025075
11. Khuvis, S., Tomko, K., Hashmi, J.M., Panda, D.K.: Exploring hybrid mpi+kokkos tasks programming model. In: 3rd IEEE/ACM Annual Parallel Applications Workshop: Alternatives To MPI+X, PAW-ATM@SC 2020, Atlanta, GA, USA, November 12, 2020. pp. 66–73. IEEE (2020). https://doi.org/10.1109/PAWATM51920.2020.00011
12. Korakitis, O., Gonzalo, S.G.D., Guidotti, N., Barreto, J.P., Monteiro, J.C., Peña, A.J.: Towards ompss-2 and openacc interoperation. In: Lee, J., Agrawal, K., Spear, M.F. (eds.) PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022. pp. 433–434. ACM (2022). https://doi.org/10.1145/3503221.3508401, https://doi.org/10.1145/3503221.3508401
13. Kraus, J.: Multi gpu programming with mpi and openacc (2015), https://on-demand.gputechconf.com/gtc/2015/presentation/S5711-Jiri-Kraus.pdf, GPU Technology Conference (GTC)
14. Matsumura, K., de Gonzalo, S.G., Peña, A.J.: JACC: an openacc runtime framework with kernel-level and multi-gpu parallelization. In: 28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17-20, 2021. pp. 182–191. IEEE (2021). https://doi.org/10.1109/HiPC53243.2021.00032, https://doi.org/10.1109/HiPC53243.2021.00032
15. Miniskar, N.R., Monil, M.A.H., Valero-Lara, P., Liu, F.Y., Vetter, J.S.: IRIS-BLAS: towards a performance portable and heterogeneous BLAS library. In: 29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18-21, 2022. pp. 256–261. IEEE (2022). https://doi.org/10.1109/HiPC56025.2022.00042, https://doi.org/10.1109/HiPC56025.2022.00042
16. Monil, M.A.H., Miniskar, N.R., Liu, F.Y., Vetter, J.S., Valero-Lara, P.: Laris: Targeting portability and productivity for LAPACK codes on extreme heterogeneous systems by using IRIS. In: IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop, RSDHA@SC 2022, Dallas, TX, USA, November 13-18, 2022. pp. 12–21. IEEE (2022). https://doi.org/10.1109/RSDHA56811.2022.00007, https://doi.org/10.1109/RSDHA56811.2022.00007
17. Papalexakis, E.E., Faloutsos, C., Sidiropoulos, N.: Tensors for data mining and data fusion. ACM Transactions on Intelligent Systems and Technology (TIST) **8**, 1 – 44 (2016)
18. Powell, J.R.: The quantum limit to moore's law. Proceedings of the IEEE **96**(8), 1247–1248 (2008). https://doi.org/10.1109/JPROC.2008.925411
19. Reyes, R., Brown, G., Burns, R., Wong, M.: SYCL 2020: More than meets the eye. In: McIntosh-Smith, S. (ed.) IWOCL '20: International Workshop on OpenCL, Virtual Event / Munich, Germany, April 27-29, 2020. p. 4:1. ACM (2020). https://doi.org/10.1145/3388333.3388649, https://doi.org/10.1145/3388333.3388649
20. Sahasrabudhe, D., Phipps, E.T., Rajamanickam, S., Berzins, M.: A portable SIMD primitive using kokkos for heterogeneous architectures. In: Wienke, S., Bhalachandra, S. (eds.) Accelerator Programming Using Directives - 6th International Workshop, WACCPD 2019, Denver, CO, USA, November 18, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12017, pp. 140–163. Springer (2019). https://doi.org/10.1007/978-3-030-49943-3_7
21. Shewchuk, J.R.: An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., USA (1994)
22. Toledo, L., Valero-Lara, P., Vetter, J.S., Peña, A.J.: Static graphs for coding productivity in openacc. In: 28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17-20, 2021. pp. 364–369. IEEE (2021). https://doi.org/10.1109/HiPC53243.2021.00050, https://doi.org/10.1109/HiPC53243.2021.00050

23. Trott, C., Berger-Vergiat, L., Poliakoff, D., Rajamanickam, S., Lebrun-Grandié, D., Madsen, J., Awar, N.A., Gligoric, M., Shipman, G., Womeldorff, G.: The kokkos ecosystem: Comprehensive performance portability for high performance computing. Comput. Sci. Eng. **23**(5), 10–18 (2021). https://doi.org/10.1109/MCSE.2021.3098509, https://doi.org/10.1109/MCSE.2021.3098509
24. Valero-Lara, P., Andrade, D., Sirvent, R., Labarta, J., Fraguela, B.B., Doallo, R.: A fast solver for large tridiagonal systems on multi-core processors (lass library). IEEE Access **7**, 23365–23378 (2019). https://doi.org/10.1109/ACCESS.2019.2900122, https://doi.org/10.1109/ACCESS.2019.2900122
25. Valero-Lara, P., Igual, F.D., Prieto-Matías, M., Pinelli, A., Favier, J.: Accelerating fluid-solid simulations (lattice-boltzmann & immersed-boundary) on heterogeneous architectures. J. Comput. Sci. **10**, 249–261 (2015), https://doi.org/10.1016/j.jocs.2015.07.002
26. Valero-Lara, P., Jansson, J.: Heterogeneous CPU+GPU approaches for mesh refinement over lattice-boltzmann simulations. Concurr. Comput. Pract. Exp. **29**(7) (2017). https://doi.org/10.1002/cpe.3919, https://doi.org/10.1002/cpe.3919
27. Valero-Lara, P., Kim, J., Hernandez, O., Vetter, J.S.: Openmp target task: Tasking and target offloading on heterogeneous systems. In: Chaves, R., Heras, D.B., Ilic, A., Unat, D., Badia, R.M., Bracciali, A., Diehl, P., Dubey, A., Sangyoon, O., Scott, S.L., Ricci, L. (eds.) Euro-Par 2021: Parallel Processing Workshops - Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13098, pp. 445–455. Springer (2021). https://doi.org/10.1007/978-3-031-06156-1_35, https://doi.org/10.1007/978-3-031-06156-1_35
28. Valero-Lara, P., Kim, J., Vetter, J.S.: A portable and heterogeneous LU factorization on IRIS. In: Singer, J., Elkhatib, Y., Heras, D.B., Diehl, P., Brown, N., Ilic, A. (eds.) Euro-Par 2022: Parallel Processing Workshops - Euro-Par 2022 International Workshops, Glasgow, UK, August 22-26, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13835, pp. 17–31. Springer (2022). https://doi.org/10.1007/978-3-031-31209-0_2, https://doi.org/10.1007/978-3-031-31209-0_2
29. Valero-Lara, P., Lee, S., Tallada, M.G., Denny, J.E., Vetter, J.S.: Kokkacc: Enhancing kokkos with openacc. In: 9th Workshop on Accelerator Programming Using Directives, WACCPD@SC 2022, Dallas, TX, USA, November 13-18, 2022. pp. 32–42. IEEE (2022). https://doi.org/10.1109/WACCPD56842.2022.00009, https://doi.org/10.1109/WACCPD56842.2022.00009
30. Valero-Lara, P., Martínez-Pérez, I., Mateo, S., Sirvent, R., Beltran, V., Martorell, X., Labarta, J.: Variable batched DGEMM. In: Merelli, I., Liò, P., Kotenko, I.V. (eds.) 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018. pp. 363–367. IEEE Computer Society (2018). https://doi.org/10.1109/PDP2018.2018.00065, https://doi.org/10.1109/PDP2018.2018.00065
31. Valero-Lara, P., Martínez-Pérez, I., Peña, A.J., Martorell, X., Sirvent, R., Labarta, J.: cuhinesbatch: Solving multiple hines systems on gpus human brain project*. In: Koumoutsakos, P., Lees, M., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M.A. (eds.) International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. Procedia Computer Science, vol. 108, pp. 566–575. Elsevier (2017). https://doi.org/10.1016/j.procs.2017.05.145, https://doi.org/10.1016/j.procs.2017.05.145
32. Valero-Lara, P., Martínez-Pérez, I., Sirvent, R., Martorell, X., Peña, A.J.: cuthomasbatch and cuthomasvbatch, CUDA routines to compute batch of tridiagonal systems on NVIDIA gpus. Concurrency and Computation: Practice and Experience **30**(24) (2018)
33. Valero-Lara, P., Pinelli, A., Favier, J., Matias, M.P.: Block tridiagonal solvers on heterogeneous architectures. In: IEEE 10th International Symposium on Parallel and Distributed Processing with Applications. pp. 609–616. ISPA '12 (2012)
34. Valero-Lara, P., Pinelli, A., Prieto-Matías, M.: Fast finite difference poisson solvers on heterogeneous architectures. Comput. Phys. Commun. **185**(4), 1265–1272 (2014). https://doi.org/10.1016/j.cpc.2013.12.026, https://doi.org/10.1016/j.cpc.2013.12.026
35. Wolf, M.M., Edwards, H.C., Olivier, S.L.: Kokkos/qthreads task-parallel approach to linear algebra based graph analytics. In: 2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016. pp. 1–7. IEEE (2016). https://doi.org/10.1109/HPEC.2016.7761649