# Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes

Javier Cabezas
Barcelona Supercomputing Center, Spain
javier.cabezas@bsc.es

Lluís Vilanova
Barcelona Supercomputing Center, Spain
lluis.vilanova@bsc.es

Isaac Gelado
NVIDIA Corporation
Santa Clara, CA, USA
igelado@nvidia.com

Thomas B. Jablin
University of Illinois at Urbana-Champaign
Urbana, IL, USA
jablin@illinois.edu

Nacho Navarro
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center, Spain
nacho@ac.upc.edu

Wen-mei W. Hwu
University of Illinois at Urbana-Champaign
Urbana, IL, USA
w-hwu@illinois.edu

## ABSTRACT

In this paper we present AMGE, a programming framework and runtime system that transparently decomposes GPU kernels and executes them on multiple GPUs in parallel. AMGE exploits the remote memory access capability in modern GPUs to ensure that data can be accessed regardless of its physical location, allowing our runtime to safely decompose and distribute arrays across GPU memories. It optionally performs a compiler analysis that detects array access patterns in GPU kernels. Using this information, the runtime can perform more efficient computation and data distribution configurations than previous works. The GPU execution model allows AMGE to hide the cost of remote accesses if they are kept below 5%. We demonstrate that a thread block scheduling policy that distributes remote accesses through the whole kernel execution further reduces their overhead. Results show 1.98× and 3.89× execution speedups for 2 and 4 GPUs for a wide range of dense computations compared to the original versions on a single GPU.

## Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Parallel Programming

## Keywords

Multi-GPU programming; NUMA

## 1. INTRODUCTION

In the race towards exascale, new node architectures are being proposed to improve both performance and energy efficiency. For example, Summit and Sierra supercomputers will include multiple GPUs and CPUs in each node connected through a high-bandwidth interconnect [3]. This interconnect will allow any CPU and GPU in the system to transparently access all the memories in the system, although with different latency and bandwidth characteristics (i.e., NUMA). Current GPU programming models, such as CUDA [24] and OpenCL [17], present GPUs as external devices with their own private memory. Programmers are in charge of splitting data and computation across GPUs and taking care of data movement, making multi-GPU programming tedious and error-prone. Programming models that allow to easily exploit all the GPUs in the node are needed.

Ideally, all GPU resources should be presented as a single virtual GPU, leaving the problem of computation and data distribution to the compiler and runtime system. Some solutions have been proposed [18, 20], but their designs exhibit fundamental limitations. (1) Large memory footprint overhead due to replication of portions of the arrays that are never accessed. For example, consider a kernel that performs n-dimensional tiling (a common pattern in dense GPU computations [26, 30, 5]) where each computation partition accesses a non-contiguous memory region of a matrix. In such a case, existing proposals transfer the whole memory address ranges accessed by each computation partition, which may include large portions of the array that are never used. This limits the size of the problems that can be handled, and imposes performance overheads due to larger data transfers. (2) High costs due to data coherence as replicated output memory regions need to be merged in the host memory after each kernel call, which often leads to higher performance degradation. (3) Applications that use atomic and global memory instructions resort to single-GPU execution.

In this paper we present AMGE (Automatic Multi-GPU Execution), a programming interface, compiler support and runtime system that automatically executes computations that are programmed for a single GPU across all the GPUs in the system. The programming interface provides a data type for multidimensional arrays that allows for robust, transparent distribution of arrays across GPU memories. The compiler extracts the dimensionality information from the type of each array, and is able to determine the access pattern in each dimension of the array. The runtime system uses the compiler-provided information to transparently choose


*ICS'15,* June 8–11, 2015, Newport Beach, CA, USA.

http://dx.doi.org/10.1145/2751205.2751218.

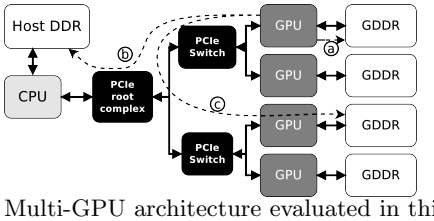


Figure 1: Multi-GPU architecture evaluated in this paper.

the best computation and data distribution configuration to minimize inter-GPU communication and memory footprint.

AMGE assumes non-coherent non-uniform shared memory accesses (NCC-NUMA) between GPUs through a relatively low-bandwidth interconnect, such that all GPUs can access and cache any partition of the arrays. Thus, we ensure that arrays can be arbitrarily decomposed, distributed and safely accessed from any GPU in the system. In current systems based on discrete GPUs, we utilize Peer-to-Peer [1] and Unified Virtual Address Space [24] technologies that enable a GPU to transparently access the memory of any other GPU connected to the same PCIe domain. While remote GPU memory accesses have been used in the past [28], this is the first work to use them as an enabling mechanism for automatic multi-GPU execution.

We also present and discuss different implementation tradeoffs for computation and data distribution across GPUs using a prototype implementation of AMGE for C++ and CUDA. The prototype includes a compiler pass that detects array access patterns in CUDA kernels and generates optimized versions of the kernels for different array decompositions. The runtime system distributes data and computation, and selects the appropriate kernel version. This prototype is evaluated using a set of GPU dense-computation benchmarks, originally developed for single-GPU execution. Results on a real system show 1.98× and 3.89× kernel execution speedups for 2 and 4 GPUs respectively, compared to the original version of the kernels running on a single GPU.

The main contributions of this paper are: (1) A multi-GPU parallelization system that performs robust space-efficient data decompositions to enable larger problem sizes. (2) A novel compiler analysis for GPU kernels that detects per-dimension array access patterns, enabling the runtime system to determine array distributions that minimize the number of remote memory accesses. (3) A simple programming interface that can be easily introduced into languages such as CUDA and OpenCL to robustly and transparently distribute computation and data across several GPUs. (4) An evaluation that shows the efficacy of the remote memory access mechanism for multi-GPU parallelization even when built on an interconnect with limited bandwidth.

# 2. MULTI-GPU ARCHITECTURE

AMGE targets systems that contain several CPUs and GPUs. Each processor is connected to one or more memory modules, but all CPUs and GPUs can access any memory module in the system. Accesses to remote memories have longer access latency and lower bandwidth than local accesses, thus forming a shared memory NUMA system. CPU cores access memory through a coherent cache hierarchy, while GPUs use weaker consistency models that do not require cache coherence between cores. Similar shared memory NCC-NUMA system architectures have been successfully implemented in the past (e.g., Cray T3E [25]). NVIDIA proposes a similar system architecture in the Echelon project [16]. Moreover, NVIDIA will offer single-board multi-GPU configurations in which GPUs share the memory through a non-coherent interconnect named NVLink, in the Pascal family of GPUs. AMD also implements coherent and non-coherent memory hierarchies in their APU chips [2].

We evaluate AMGE on an existing commercial system based on NVIDIA discrete GPUs (see Figure 1). In this system, GPUs access their local memory (arc *a*) with full-bandwidth. Accesses to CPU memories from the GPU (arc *b*) are routed through the PCI Express (PCIe) interconnect and the CPU memory controller. If the target address resides in a memory connected to a different CPU socket, the inter-CPU interconnect (HyperTransport/QPI) must be traversed, too. GPUs can also access the memory in another GPU through the PCIe interconnect (arc *c*) [1].

While the execution model provided by GPUs can hide large memory latencies, both CPU memory and the inter-GPU interconnects (e.g., PCIe 2.0/3.0) deliver a memory bandwidth which is an order of magnitude lower than the local GPU memory (GDDR5). New interconnects that provide much higher bandwidth have been announced (e.g., NVLink is projected to deliver up to 100 GB/s), but the memory technology will also keep improving, thus maintaining this gap. Therefore, minimizing remote accesses is key to maximize application performance.

## 2.1 GPU Programming Model

GPUs are typically programmed using a Single Program Multiple Data (SPMD) programming model, like NVIDIA CUDA [24] or OpenCL [17]. For simplicity, we adopt the CUDA terminology in the remainder of the paper. A SPMD model lets programmers spawn a large number of threads that execute the same program, although each thread can take a different control flow path. All these threads are organized into a *computation grid* of groups of *thread blocks* (i.e., groups of threads). Each thread block has an identifier and each thread has an identifier within the thread block, which can be used by programmers to map the computation to the data structures. Both CUDA and OpenCL provide weak consistency models: memory updates performed by a thread might not be perceived by others, except for atomic and memory fence instructions.

**Multi-GPU Programming**. In CUDA and OpenCL, GPUs are presented as external devices with their own memories. Programmers typically decompose computation and data so that each GPU only accesses its local memory. If there are regions of data that are accessed by several GPUs, programmers are responsible of replicating and keeping them coherent through explicit memory transfers. CUDA exposes a Unified Virtual Address Space (UVAS), which ensures that virtual memory addresses are unique across all memories in the system. Using UVAS, programmers can map the pages of a memory allocation on different GPU memories. However, CUDA does not provide any means to control how virtual addresses are mapped to physical memory, and allocations are bound to a single GPU.

Oftentimes, data structures need to be accessed by both CPU and GPU code, and programmers are in charge of using separate copies of the data structures and keeping them coherent through explicit memory transfers. This extra code incurs additional development time and harms maintainability. Recently, CUDA introduced UVM (Unified Virtual

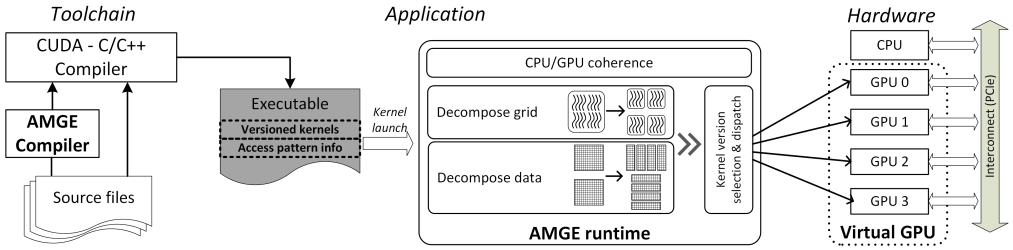



Figure 2: Overview of AMGE components. The compiler extracts array access pattern information and stores it in the program binary. It also generates optimized kernel versions for different array decompositions. The runtime system uses this information to decompose and distribute computation and data across the GPUs in the system.

Memory), which allows allocating memory that can be accessed by CPU and GPU code, but not concurrently. UVM is based on the ADSM model [13], in which the memory that is shared by CPU and GPUs is acquired/released by the GPU at kernel call boundaries. OpenCL 2.0 also exposes a Shared Virtual Memory space [17], but it does not allow programmers to specify in which memory data is actually allocated. AMGE builds on remote memory accesses, UVAS, and ADSM technologies.

# 3. AMGE OVERVIEW

AMGE is a programming framework that decomposes and distributes GPU kernels and data to be collaboratively executed on all the GPUs in the system. We implement AMGE using C++ and CUDA, but it can be extended to other languages. Figure 2 shows the components in AMGE and how they interact with the hardware. AMGE aggregates the GPU resources in the system and presents them as a *single virtual GPU*. Thus, programmers are relieved from the burden of decomposing the problem and explicitly managing several GPUs.

*The AMGE compiler* is a source-to-source compiler, that analyzes the CUDA kernels in the program to detect their array access patterns and store this information in the program executable. It also generates optimized kernel versions for the possible array decompositions. We argue that the utilization of the array dimensionality information is paramount in order to efficiently exploit multi-GPU systems. However, CUDA is an extension of the C/C++ languages, which do not provide data types with such information; programmers typically flatten the multi-dimensional arrays into 1D arrays and linearize the dimension indices in each array reference. It is practically difficult, if not infeasible, for static analysis to reliably recover the dimensionality information once the accesses have been flattened. AMGE provides a new data type for multi-dimensional arrays that makes this information available to the compiler. Details on the implementation of the data type and the generation of optimized kernel versions are discussed in Section 5.

The other key feature of AMGE is the utilization of *remote memory accesses* between GPUs [1]. On each reference to the array, the underlying implementation determines whether the element being referenced is hosted in the memory local to the GPU executing the code or on a different GPU. References from a GPU to parts of the array stored in different GPU memories are handled using remote memory accesses. This approach ensures correctness regardless of the chosen GPU computation and data distribution configuration and removes the requirement for the compiler analysis

```
1  void sgemm(ndarray<float, 2, cmo> C, ndarray<float, 2, cmo> A,
2             ndarray<float, 2> B)
3  {
4     float partial[SGEMM_TILE_N];
5     __shared__ float b_tile_sh[SGEMM_TILE_HEIGHT][SGEMM_TILE_N];
6     for (int i = 0; i < SGEMM_TILE_N; i++) partial[i] = 0.0f;
7
8     int mid = threadIdx.y * blockDim.x + threadIdx.x;
9     int row = blockIdx.x * (SGEMM_TILE_N * SGEMM_TILE_HEIGHT) + mid;
10    int col = blockIdx.y *  SGEMM_TILE_N + threadIdx.x;
11
12    for (int i = 0; i < A.get_dim(1); i += SGEMM_TILE_HEIGHT) {
13       b_tile_sh[threadIdx.y][threadIdx.x] = B(i + threadIdx.y, col);
14       __syncthreads();
15       for (int j = 0; j < SGEMM_TILE_HEIGHT; ++j) {
16          float a = A(row, i + j);
17          for (int k = 0; k < SGEMM_TILE_N; ++k)
18             partial[k] += a * b_tile_sh[j][k];
19       }
20       __syncthreads();
21    }
22    for (int i = 0; i < SGEMM_TILE_N; i++)
23       C(row, i + by * SGEMM_TILE_N) = partial[i];
24 }
```

Listing 1: Multi-GPU `sgemm` GPU code with AMGE. `cmo` means *column major order*.

to unequivocally determine the bounds of the memory range accessed by a computation partition.

However, remote accesses can impose performance overheads and they must be minimized. On each kernel call, *the AMGE runtime* determines the best computation and array decompositions using the information generated by the compiler, and distributes them across all GPUs in the system.

**Memory model**: Arrays are transparently decomposed and/or replicated before each kernel call. Input arrays can be replicated at the cost of additional space and data transfers, but AMGE never replicates output arrays. Replicated output arrays require from additional coherence management to merge partial modifications on different GPUs. Previous works [18, 20] transfer all copies to the host memory for a merging step after every kernel call, imposing a large performance overhead in many workloads. Moreover, replicating output arrays prevents distributing codes with atomics or memory fences. AMGE always distributes output arrays across GPU memories instead, and relies on remote memory accesses to guarantee that they are available to all GPUs.

AMGE implements the ADSM model [13] to allow arrays to be used both by host and GPU code. The runtime transfers arrays between CPU and GPU memories as needed.

## 3.1 An example: matrix multiplication

Code programmed to run on a single GPU requires only minor modifications to use AMGE. Listing 1 shows the GPU code of a single-precision floating point matrix-matrix multiplication computation [12] (i.e., `sgemm`) using AMGE. $A$ and $C$ matrices are stored in *column major* order, and $B$ in *row major* order, for optimal performance. The highlighted text shows the modifications performed to the original code. The

```
1   // Initialize A and B in the host code
2   ndarray<float, 2, cmo> A;
3   ndarray<float, 2>      B;
4
5   read_array("A.dat", A);
6   read_array("B.dat", B);
7
8   ndarray<float, 2, cmo> C(A.get_dim(1), B.get_dim(0));
9   // Computation grid size
10  dim3 block(MATRIXMUL_TILE_N, SGEMM_TILE_HEIGHT);
11  dim3 grid(C.get_dim(1)/(SGEMM_TILE_N * SGEMM_TILE_HEIGHT),
12            C.get_dim(0)/SGEMM_TILE_N);
13  // Kernel launch. A, B and C are used in the GPU code
14  sgemm<<<grid, block>>>(C, A, B);
15  // Write results for C into a file
16  write_array("C.dat", C);
```

Listing 2: Multi-GPU `sgemm` host code with AMGE.

only additional programming requirement for the kernel to be automatically decomposed is the array data type (lines 1-2), and its associated indexing routines (lines 12, 13, 16 and 23). The data type is implemented by the `ndarray<T, Dims, Storage>` C++ class template, where `T` is the type of the elements, `Dims` is the number of dimensions of the array, and `Storage` is an optional parameter that defines the storage type (by default, the C/C++ *row major* order storage convention is used). The kernel uses a 2D computation grid, in which each thread block computes a 2D tile of $C$ by traversing $A$ and $B$ on their $X$ and $Y$ dimensions, respectively. The compiler detects these patterns and stores them in the program executable (*access pattern info* in Figure 2).

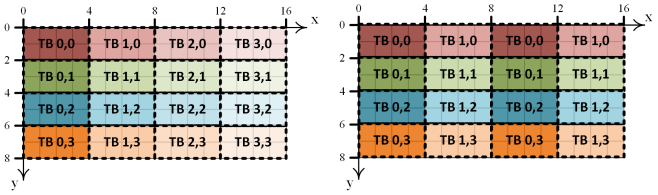
Listing 2 shows the CPU code of the `sgemm` computation. First, `float` input matrices `A` and `B` are declared in lines 2 and 3. The bounds of each dimension of the array are defined at run-time. The AMGE runtime intercepts the kernel call and uses the information generated by the compiler to decompose the matrices, and distribute both computation and data across all the GPUs. Note that `ndarray` objects can be passed both to CPU and GPU routines, making explicit data transfers between host and GPU memories unnecessary.

# 4. COMPUTATION/DATA DISTRIBUTION

AMGE decomposes GPU kernels at thread block boundaries. We choose this decomposition granularity because threads within a thread block share resources (e.g., shared memory), and perform barrier synchronization operations. Hence, all threads within a thread block must be executed in the same compute core of the same GPU. However, the GPU programming model guarantees that there are no data dependences across thread blocks within a kernel and, therefore, they can execute independently.

In CUDA, programmers specify a computation grid that is a multidimensional space $grid_x \times grid_y \times grid_z$ of thread blocks, similar to the iteration space in loop nests. Each thread block has a unique identifier $0 \leq block_x, block_y, block_z < grid_x, grid_y, grid_z$ within the computation grid. The AMGE runtime decomposes the computation grid so that it can be executed on several GPUs. In CUDA and OpenCL, the computation grid is canonical and rectangular. Thus, the computation grid can be uniformly decomposed into partitions along one or several of its dimensions.

AMGE uses compiler analysis to generate array access pattern information for all the GPU kernels in the program. This information is later used by the runtime system to try to place on each GPU the data accessed by the computation partition assigned to it, in order to minimize the number of remote memory accesses.




(a) BLOCK (b) BLOCK-CYCLIC

Figure 3: Computation-to-data mapping examples.

## 4.1 Compiler analysis

The AMGE compiler analyzes all array references in the kernel. Each reference contains one index for each of the dimensions of the array, allowing the AMGE compiler to detect the individual access pattern in each dimension. This is in contrast to previous works [18, 20] that treat all arrays as one dimensional. Kim et al. [18] compute the upper and lower memory addresses of the tiles accessed by each computation partition to distribute the arrays. This may lead to unnecessary replication of large portions of the application dataset, also imposing higher data copy costs. For example, $TB_{0,0}$ in Figure 3a accesses a 2D tile composed of elements from two different rows of a matrix: $\{(0,0)\ldots(0,3)\} \cup \{(1,0)\ldots(1,3)\}$. However, the linear memory address range defined by the upper and lower bounds of the tile also contains elements that belong to the neighboring tiles in the $X$ dimension of the matrix: $\{(0,0)\ldots(0,15)\} \cup \{(1,0)\ldots(1,3)\}$. Moreover, for output arrays this generates additional data merging operations, as tiles might be wrongly classified as overlapping. For example, the tiles for $TB_{0,0}$ and $TB_{1,0}$ do not overlap, but the analysis in previous works uses their linear address ranges, which do overlap. The per-dimension access pattern information allows AMGE to identify multi-dimensional tiles as non-overlapping entities.

We consider three classes of access patterns: (1) as a function of thread block indices, (2) local to a thread block, and (3) data-dependent. The first class, the most commonly found in GPU applications, is produced when programmers access arrays using affine transformations of the block and thread indices. As a result, threads belonging to thread blocks with contiguous identifiers in one dimension access elements that are contiguous in a dimension of the array. In the `sgemm` example (Listing 1), $block_x$ is used to access $A_y$ (line 16) and $C_y$ (line 23), while $block_y$ is used to access $B_x$ (line 13) and $C_x$ (line 23). We use the notation $A_i$ to refer to the $i$th dimension of array $A$. This linear relationship allows us to relate thread blocks with the portions of the arrays accessed by them.

The second access pattern type is produced when array dimensions are traversed through loop induction variables or local thread indexes. In the `sgemm` example (Listing 1), threads traverse $A_x$ using the induction variables $i+j$ of the nested loops (line 16).

The third type of accesses cannot be determined at compile time since the indices are computed with values that are only known at kernel execution time.

Only array dimensions accessed using the first pattern class are eligible for decomposition, since the second class refers to access patterns local to a single thread or thread block, and the third class cannot be determined statically. AMGE uses a novel compiler analysis, for the dimensions that are eligible for decomposition, that identifies the thread block-to-data mappings, and classifies them into the most

common distribution types: `BLOCK`, `CYCLIC` and `BLOCK-CYCLIC` (illustrated in Figure 3 for a 2D array)[14, 19, 6]. This analysis tries to map the index expression used in each dimension of each array reference to the following canonical form:

$$m \times t \times G + t \times B + k \quad (1)$$

where:

- $m$: index of the non-contiguous array tile accessed by a thread block (an induction variable or a constant).
- $t$: array tile size. It is a multiple of the thread block size when threads access different elements or 1 if all threads access the same element of the array's dimension (e.g., the same row in a matrix). The actual thread block size is extracted from the kernel launch parameters at run-time.
- $G$: number of thread blocks in a dimension of the computation grid (e.g., $grid_x$).
- $B$: thread block index (e.g., $block_x$). Expressions not using this term do not belong to the first access pattern class.
- $k$: thread index or a constant. This value does not determine the access patterns across thread blocks.

We find that most GPU array-based computations use this form to distribute the array across thread blocks.

The most common and simple index expression is found when each thread block accesses a single contiguous array tile (i.e., $m = 0$). This expression is classified as `BLOCK`. Another common index expression assigns non-contiguous array tiles to each thread block (i.e., $m > 0$) using a grid-sized stride. This expression is classified as `BLOCK-CYCLIC`. `CYCLIC` is a special case of `BLOCK-CYCLIC`, in which $t = 1$.

**Information generation for the runtime**: The compiler analysis generates a record with the following information for each dimension of each array used in each GPU kernel: (1) the dimension of the computation grid whose thread block index is used to index the array dimension; (2) the access type (`Read`/`Write`); (3) the distribution type (`BLOCK`, `CYCLIC`, `BLOCK-CYCLIC`). This information is built by combining the values from the different references to the array in the kernel: (1) the intersection of the used computation grid dimensions; (2) the union of the access types; (3) the intersection of the distribution types. Thus, if an array dimension is indexed using block indices of different dimensions of the computation grid or different distribution types, an empty record is generated for that dimension. Array dimensions that are not indexed using the first data access pattern class, get empty records, too. Only array dimensions with non-empty records can be distributed.

## 4.2 Run-time distribution

On each kernel execution, the runtime identifies all possible computation grid decompositions. Then, it uses the compiler-provided information to compute the distribution configurations for each potential computation grid decomposition. Since the computation grid is limited to three dimensions, there are, at most, eight potential decompositions. Finally, the runtime ranks each distribution configuration using an analytical model (details in Section 5.3) and distributes the arrays and computation using the highest-ranked configuration.

### 4.2.1 Computation grid distribution

For a specific computation grid decomposition, the AMGE runtime defines a grid of GPUs ($gpu_0 \times gpu_1 \times \ldots gpu_{N-1}$) with as many dimensions as decomposed dimensions in the computation grid. Then, the computation grid is decomposed into as many partitions as the number of GPUs in each dimension of the GPU grid, and each partition is assigned to a GPU. We refer to the mapping of the computation grid to the GPU grid as computation distribution configuration.

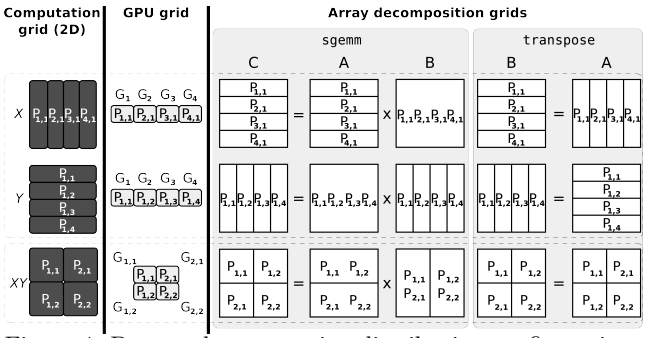



Figure 4: Data and computation distribution configurations for `sgemm` and `transpose` on a 4-GPU system. $A$, $B$, $C$ are the arrays used in the kernels, $P_{i,j}$ is a partition of the computation grid, $G_{i,j}$ is a GPU in the GPU grid.

### 4.2.2 Array distribution

The runtime system uses the compiler-generated access pattern information to determine how arrays must be decomposed for a specific computation distribution configuration. An array is decomposed along those dimensions that are indexed with the thread block indices of the decomposed computation grid dimensions. This creates an $m$-dimensional grid of tiles, where $m$ is the number of decomposed dimensions in the array. Output arrays that are not accessed with any of the thread block indexes of the decomposed computation grid dimensions, are arbitrarily decomposed along their highest-order dimension.

The size and number of tiles depend on the computation distribution configuration. If $block_i$ is used to index $L_j$, and $grid_i$ is mapped on the $k$ dimension of the GPU grid, then $L_j$ is decomposed into as many tiles as GPUs in $gpu_k$ and distributed among them. Figure 4 shows the relationship between the computation grid, the GPU grid and the array decomposition grid in different computation distribution configurations for `sgemm` and `transpose`. In the `sgemm` example, $C_x$, $C_y$, $A_y$ and $B_x$ can be potentially decomposed, as they are indexed with thread block indices. For a configuration that decomposes the computation grid on its $Y$ dimension ($grid_y$, second row of Figure 4), $C_x$ and $B_x$ are decomposed. However, since $block_x$ is used to index $A_y$ but $grid_x$ is not decomposed, $A_y$ is not decomposed either. In `transpose`, both $block_x$ and $block_y$ are used to index the two dimensions of $A$ and $B$ matrices and are always decomposed. The distribution of tiles among the GPU memories follows the mapping of the computation grid partitions on the GPU grid. For example, in the $XY$ computation grid decomposition of `sgemm` (third row of Figure 4), neighboring tiles in $C_x$ are distributed across neighboring GPUs in $gpu_y$.

Input arrays that are not decomposed are replicated in all GPU memories. Moreover, tiles from input arrays that are not indexed using the thread block indexes of all the decomposed dimensions of the computation grid, are *partially* replicated. In this case, the array tiles are only replicated in the memories of those GPUs that belong to the GPU grid

dimensions on which the unused computation grid dimensions are distributed. For example, in `sgemm`, $C$ is indexed with both $block_x$ and $block_y$ and its tiles are not replicated for any computation grid decomposition. However, $A$ is indexed with $block_x$, and $B$ is indexed with $block_y$, only. Thus, $A$ and $B$ are fully replicated in all GPUs for $Y$ and $X$ computation grid decompositions, respectively. In the $XY$ configuration, the tiles in $A$ are distributed across the GPUs in $gpu_x$ and each tile is replicated in all GPUs in $gpu_y$. For example, the upper tile of $A$ is accessed by both $P_{1,1}$ and $P_{1,2}$ computation partitions, and it is replicated in the memories of GPUs $G_{1,1}$ and $G_{1,2}$. Conversely, the tiles in $B$ are distributed across the GPUs in $gpu_y$ and replicated in the GPUs in $gpu_x$.

# 5. IMPLEMENTATION DETAILS

## 5.1 Array data type

We utilize the UVAS support to place different parts of the array in different GPU memories while having a continuous representation of the array in the virtual address space. Hence, decomposed arrays can be referenced by using regular linearization operations on the indexes: $(a_1, \cdots, a_n) \rightarrow \sum_{i=1}^{n} a_i \times \prod_{j=i+1}^{n} D_j$ where $a_i$ is the index and $D_i$ the number of elements in the $i$th dimension of the array. Indexes are ordered from the highest-order to the lowest-order dimension. We refer to this scheme as **VM** implementation.

This implementation decomposes arrays with page-size granularity which, in current GPUs, is in the order of a few kilobytes. Nevertheless, we have experimentally determined that current versions of CUDA impose a 1 MB (instead of page-size) granularity to allocate contiguous virtual memory ranges on different GPUs. This produces data distribution imbalance if partitions cannot be stored in balanced-sized groups of 1 MB chunks, resulting in an increased number of remote memory accesses. One solution to reduce the imbalance is to add padding to the lower order dimensions that are not decomposed. However, achieving perfect balancing using 1 MB chunks can impose a footprint overhead in the order of hundreds of times, especially for arrays' lowest-order dimensions, whose elements are stored contiguously in memory. Another solution is to permute the dimensions of the array to ensure that decomposed dimensions are not contiguous in memory. Unfortunately, this can break memory coalescing [27].

Therefore, we also provide an alternate implementation proposed in [4] that **reshape**s the arrays. In this implementation, each GPU contains a memory allocation that holds all the elements in a partition, and it is padded to the 1MB boundary. The array is reorganized by adding a new dimension for each decomposed one (i.e., strip-mining), that indicates the GPU in which each partition is stored. In each array reference, the original indexes are transformed into a new set of indexes. This approach allows arrays to be decomposed on any dimension as they do not have to be stored contiguously in the virtual address space. However, this flexibility comes at the cost of extra computation. For example, if a 3D volume is decomposed along its highest-order dimension using a `BLOCK` distribution, the index for this dimension $a_1$ is transformed as follows:

$$(a_1, a_2, a_3) \rightarrow (\left\lfloor \frac{a_1}{D'_1} \right\rfloor, a_1 \bmod D'_1, a_2, a_3) \quad (2)$$

where $D'_1 = \left\lceil \frac{D_1}{P_1} \right\rceil$ and $P_1$ is the number of tiles in the first dimension of the array's decomposition grid. The operations needed to compute the location of the element $a_1, a_2, a_3$, are divided into: the computations of the offset of the block in the dimension being decomposed, and the linearization of the index within the block. Therefore, an extra division and modulo operations are performed in each access to the array, compared with the regular index linearization. For `CYCLIC` and `BLOCK-CYCLIC`, similar transformations are performed.

Providing a generic indexing routine that supports all possible array decompositions can impose an unacceptable performance overhead due to the extra operations needed to transform all the indexes. In order to ensure maximum performance, our prototype provides different implementations of the indexing routines optimized for the different array decompositions and distribution types.

## 5.2 Source-to-source transformations

**Kernel versioning**: Using specialized indexing routines for each decomposition requires changes in the kernels, as (1) the kernel code must explicitly call the proper versions of the routines, and (2) the array decomposition to be used is not known at compile time. Thus, AMGE generates different kernel versions for all the possible array decompositions of the arrays used in the kernel. In each version, array references use the indexing routines that are optimized for a chosen decomposition. On each kernel execution, when the runtime system selects the distribution for all the arrays, it uses the specialized kernel version for that configuration.

**Global thread block identifiers**: Since computation partitions are executed independently, CUDA assigns new thread block identifiers in the kernel invocations on each GPU. In order to retain the original identifiers, we store the offsets of each computation partition in the memory of each GPU. The compiler modifies replaces `blockIdx` with code that computes the original indexes using these offsets.

**Memory consistency model**: Distributed kernels must honor the memory consistency of the GPU programming model. Data propagation across thread blocks in the GPU model is only guaranteed for atomic memory operations and memory fences. For atomic operations we exploit the hardware support provided by modern system architectures (e.g., atomic operations in PCIe 3.0). GPU-wide memory fences are translated to system-wide memory fences to ensure correctness. Finally, since a GPU may cache remote array partitions, GPU caches are flushed at kernel exit boundary.

**AMGE toolchain**: AMGE provides a toolchain based on the LLVM framework. The first pass performs the array access pattern analysis introduced in Section 4.1 on the LLVM IR generated by the CUDA compiler, and generates code to pass the information to the runtime system. The second pass handles `blockIdx` and memory fence translations. The third pass generates specialized kernel versions for the possible array decompositions, and code for the runtime system to retrieve the version for a chosen configuration. Generated code is compiled into the program executable.

## 5.3 Run-time distribution selection policy

As explained in Section 4.2, on a kernel call, the runtime system selects the best distribution configuration. Our prototype implements a policy that (1) favors the array implementation that imposes the least overhead, (2) minimizes the number of remote accesses. Thus, the VM implemen-

tation is preferred (since it does not impose any indexing overhead) unless it introduces *too many* remote memory accesses due to data distribution imbalance. Our policy ranks the array decompostions given by the runtime system for both VM and reshape implementations. For VM, the score is calculated using the data distribution imbalance introduced by the coarse allocation granularity. The imbalance is computed analytically by counting the bytes that belong to an array partition that should be stored in a different GPU, with respect to the total array size. The cost of this computation is negligible compared to the kernel execution time. When the imbalance exceeds a threshold, the score becomes zero (effectively choosing the reshape implementation). If several configurations get the same score, decompositions on the highest-order dimension are preferred because they allow for more efficient CPU↔GPU transfers.

# 6. EXPERIMENTAL METHODOLOGY

All experiments were run on a system containing a quad-core Intel i7-3820 at 3.6 GHz with 64 GB of DDR3 RAM memory, and 4 NVIDIA Tesla K40 GPU cards with 12 GB of GDDR5 each, connected through a PCIe 3.0 in x16 mode (containing two PCIe switches like in Figure 1). The machine runs a GNU/Linux system, with Linux kernel 3.12 and NVIDIA driver 340.24. Benchmarks were compiled using GCC 4.8.3 for CPU code and NVIDIA CUDA compiler 6.5 for GPU code. Execution times were measured using the CUPTI profiling library that provides support for sampling and nanosecond timing accuracy. For runs with more than one GPU, graphs show the time for the slowest GPU.

We evaluate AMGE using a number of dense scientific computations that use different computation and array access patterns. The list of benchmarks is summarized in Table 1. Some of them are found in the Parboil benchmark suite [15], some in the NVIDIA SDK, and the rest have been developed in-house. This benchmark selection provide a good variety of access patterns and thus challenges. Both CPU and GPU codes have been modified to use the `ndarray` data type instead of the flat 1D arrays which are commonly used. The benchmarks have been compiled using our toolchain and linked to our runtime system. The `ndarray` implementation has an impact on the register usage count of the kernels (columns 4-7). In the column titles, "Orig" stands for original, "VM" for virtual memory and "Re" for reshape, while "B" stands for `BLOCK`, "C" for `CYCLIC` and "B-C" for `BLOCK-CYCLIC`. `BLOCK` and `CYCLIC` are in the same column ("#Reg Re B/C") as they use the same number of registers.

Columns 8 and 9 show the array decompositions and distributions suggested by AMGE for the possible computation distribution configurations. $A$, $B$, $C$ are the names of the arrays used in the computation. In the case of the kernels in merge sort, a suffix has been added for keys ($A_k$, $B_k$) and values ($A_v$, $B_v$). In the last column, "D" stands for distribution and "R" for replication in the GPU grid. $X$ and $Y$ make reference to the decomposed dimensions of the computation grid. Array decompositions are shown using the notation in HPF [19], but dimensions are ordered (left to right) from highest to lowest order as they are stored in memory. For example, the `sgemv` configuration says that for a computation decomposition on $X$, the $A$ matrix is decomposed on its $Y$ dimension and the $C$ vector is decomposed on its $X$ dimension. Tiles in $A$ and $C$ are distributed across the GPUs in the $X$ dimension of the GPU grid while $B$ vector is replicated.

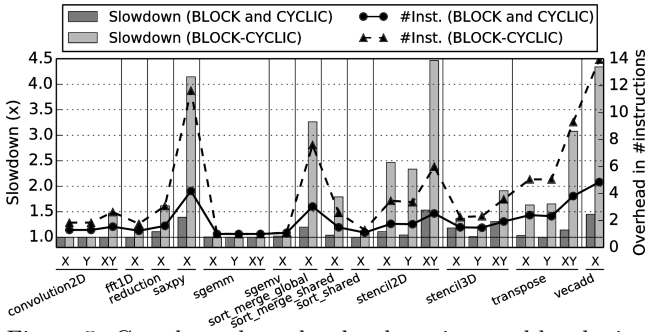



Figure 5: Grey bars show the slowdown imposed by the indexing routines for the *reshape* array implementation compared to the baseline (left axis). Lines indicate the increase in number of executed instructions (right axis).

# 7. PERFORMANCE EVALUATION

## 7.1 Indexing overhead

Table 1 shows that register utilization greatly varies depending on the used `ndarray` implementation. The *VM* implementation only performs the index linearization and the register count is similar to, or even slightly lower than the original version of the benchmarks. *reshape*, on the other hand, uses more registers in most of the kernels, especially in `BLOCK-CYCLIC` decompositions and those configurations in which arrays are decomposed along several dimensions.

Figure 5 shows the overheads imposed by the indexing routines for the *reshape* implementation on a single GPU. While AMGE suggests the utilization of the BLOCK distribution type in all kernels, we study the overhead of the routines for all the distribution types. Grey bars show the slowdown imposed by the indexing overhead of the data distributions for each computation distribution configuration (left axis). Lines represent the increase in number of executed instructions due to the extra operations performed on the indexes (right axis). `BLOCK` and `CYCLIC` are grouped (dark gray bar and solid line) as they perform very similar transformations on the indexes and the performance is virtually the same (±1%). `BLOCK-CYCLIC` (light Gray bar and dashed line) is consistently the slowest implementation (up to 4.48×) and the one that executes more instructions, too. This is caused both by the extra executed instructions and the lower achieved occupancy in the GPU due to the increased number of registers. Slowdowns are large for kernels in which thread blocks perform little work (`convolution2D`, `reduction`, `saxpy`, `sort_merge_*`, `stencil2D`, `transpose` and `vecadd`). Results for *VM* implementation are not shown because performance is within ±5% of the baseline in all kernels.

## 7.2 Multi-GPU performance

Figure 6 shows the speedup achieved by AMGE on our multi-GPU system for all possible distribution configurations. Results are shown for the *VM* and *reshape* implementations of the `ndarray` data type, and for an *ideal* implementation with optimal data distribution (no remote accesses). Bars labeled with "Impl" show the geometric mean for the three implementations of the speedups achieved in each kernel by the best computation distribution configuration. The *reshape* implementation exhibits linear speedups (1.91× and 3.54× on average for 2 and 4 GPUs, respectively) for all kinds of distribution configurations in most kernels. The

| Kernel | Suite | Inputset | #Reg Orig | #Reg VM | #Reg Re B/C | #Reg Re B-C | Array Decompositions | Array Distribution |
|---|---|---|---|---|---|---|---|---|
| convolution2D | Parboil 2 | 16K×16K<br><br>C: 9×9 | 17 | 19 | 19<br>19<br>19 | 22<br>22<br>22 | X→A,B(*,BLOCK) C(*,*)<br>Y→A,B(BLOCK,*) C(*,*)<br>XY→A,B(BLOCK,BLOCK) C(*,*) | A,B:{$D_x$} C:{$R_x$}<br>A,B:{$D_x$} C:{$R_x$}<br>A,B:{$D_{x,y}$} C:{$R_{x,y}$} |
| fft1D | Parboil | 256M | 22 | 24 | 24 | 24 | X→A(*) B(BLOCK) | A:{$R_x$} B:{$D_x$} |
| reduction | SDK | 256M | 12 | 12 | 11 | 18 | X→A,B(BLOCK) | A,B:{$D_x$} |
| saxpy | - | 256M | 8 | 8 | 8 | 17 | X→A,B(BLOCK) | A,B:{$D_x$} |
| sgemm | Parboil 2 | 4K×4K | 38 | 33 | 41<br>40<br>42 | 41<br>40<br>42 | X→A,C(*,BLOCK) B(*,*)<br>Y→A(*,*) B(*,BLOCK) C(BLOCK,*)<br>XY→A,B(*,BLOCK) C(BLOCK,BLOCK) | A,C:{$D_x$} B:{$R_x$}<br>A:{$R_x$} B:{$D_x$} C:{$D_x$}<br>A:{$D_x$,$R_y$} B:{$D_y$,$R_x$} C:{$D_{x,y}$} |
| sgemv | - | 8K×8K | 11 | 11 | 11 | 16 | X→A(BLOCK,*) B(*) C(BLOCK) | A:{$D_x$} B:{$R_x$} C:{$D_x$} |
| sort_merge_global<br>sort_merge_shared<br>sort_shared | SDK | 256M | 17<br>17<br>17 | 16<br>16<br>18 | 18<br>18<br>19 | 28<br>27<br>28 | X→$A_k$,$A_v$,$B_k$,$B_v$(BLOCK)<br>X→$A_k$,$A_v$,$B_k$,$B_v$(BLOCK)<br>X→$A_k$,$A_v$,$B_k$,$B_v$(BLOCK) | $A_k$,$A_v$,$B_k$,$B_v$:{$D_x$}<br>$A_k$,$A_v$,$B_k$,$B_v$:{$D_x$}<br>$A_k$,$A_v$,$B_k$,$B_v$:{$D_x$} |
| stencil2D | - | 16K×16K<br>+<br>halos | 17 | 18 | 17<br>17<br>19 | 23<br>19<br>26 | X→A,B(*,BLOCK)<br>Y→A,B(BLOCK,*)<br>XY→A,B(BLOCK,BLOCK) | A,B:{$D_x$}<br>A,B:{$D_x$}<br>A,B:{$D_{x,y}$} |
| stencil3D | Parboil 2 | 1K×1K×512<br>+<br>halos | 24 | 26 | 30<br>29<br>33 | 34<br>31<br>41 | X→A,B(*,*,BLOCK)<br>Y→A,B(*,BLOCK,*)<br>XY→A,B(*,BLOCK,BLOCK) | A,B:{$D_x$}<br>A,B:{$D_x$}<br>A,B:{$D_{x,y}$} |
| transpose | SDK | 16K×16K | 16 | 14 | 17<br>16<br>18 | 24<br>23<br>26 | X→A(*,BLOCK) B(BLOCK,*)<br>Y→A(BLOCK,*) B(*,BLOCK)<br>XY→A,B(BLOCK,BLOCK) | A,B:{$D_x$}<br>A,B:{$D_x$}<br>A:{$D_{x,y}$} B:{$D_{y,x}$} |
| vecadd | - | 256M | 10 | 10 | 10 | 18 | X→A,B,C(BLOCK) | A,B,C:{$D_x$} |

Table 1: Benchmark description.

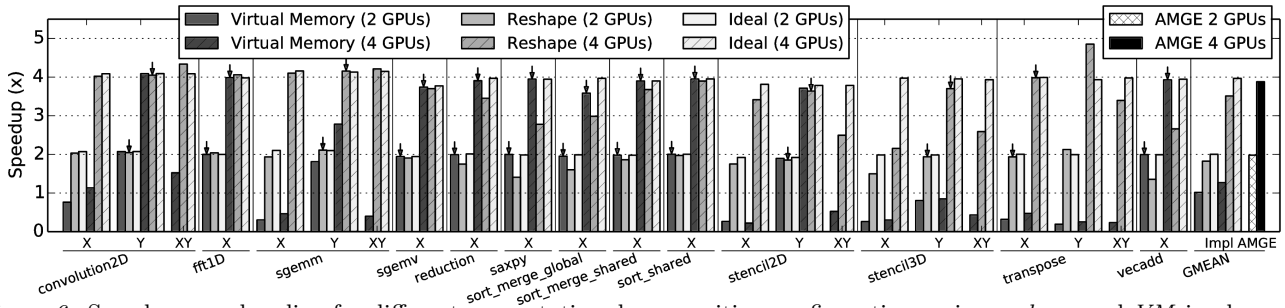
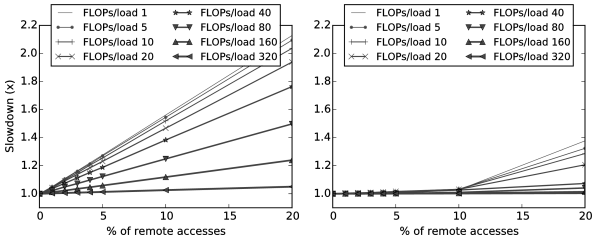



Figure 6: Speedup over baseline for different computation decomposition configurations using *reshape* and *VM* implementations. Arrows point to the configuration chosen by the runtime system for each kernel. Results shown for 2/4 GPUs.

main exceptions are *X* and *XY* configurations in `stencil3D` due to remote memory accesses, and `saxpy`, `vecadd`, `sort_merge_global` and the *XY* configuration in `stencil2D` due to the overhead of the indexing function. The *VM* implementation outperforms *reshape* in some configurations in which the array partitions are large enough not to suffer from imbalance due to the memory allocation granularity imposed by CUDA, but performs very poorly in the other configurations, producing lower speedups on average (1.02× and 1.27× for 2 and 4 GPUs). For example, in `transpose` at least one of the matrices must be decomposed along its *X* dimension, which is contiguous in memory, thus leading to an imbalanced data distribution. Therefore, performance is poor for *VM* in all configurations in `transpose`. Another example is `stencil3D`, for which the *Y* distribution configuration should provide reasonable performance since each plane of the volume can be distributed across GPUs. Nevertheless, the size of each plane still produces an imbalanced distribution. We study this example in more detail in Section 7.3.

On each kernel call, AMGE's runtime system tries to choose the best computation and data distribution configuration. In this evaluation, AMGE implements the policy introduced in Section 5.3, using a 5% ditribution imbalance threshold to choose between *VM* and *reshape* implementations. We choose this number because we have determined experimentally (Figure 7) that the GPUs in our evaluation system can hide the costs of up to 10% remote accesses depending on (1) how they are distributed in the kernel execution (Batch or Spread); and (2) the computational intensity of the kernel (FLOPS/load). We can see that, using this value, the policy correctly selects the best performing configuration for most of the kernels. Figure 6 highlights with an arrow the chosen configurations. The average performance across all the benchmarks (i.e., bars labeled with "AMGE") is 1.98× and 3.89× for 2 and 4 GPUs; very close to *ideal*.




(a) Batch (b) Spread

Figure 7: Performance overhead imposed by remote accesses for different computation intensities; when they are concentrated (Batch) or evenly distributed in time (Spread).

## 7.3 Impact of remote accesses on performance

Figure 8 shows the percentage of accesses to RAM memory that are served by remote GPUs in all the kernels when they are distributed across 4GPUs. *reshape* eliminates the need for remote accesses in most of the configurations. Only kernels in which computation partitions share some data use them (e.g., `convolution2D`, `stencil{2,3}D`). The worst cases are the X and XY decompositions for `stencil3D` in which 17.43% and 7.61% of accesses to memory are remote, respectively. This is the reason why these configurations show poor speedups in Figure 6. In contrast, *VM* introduces a

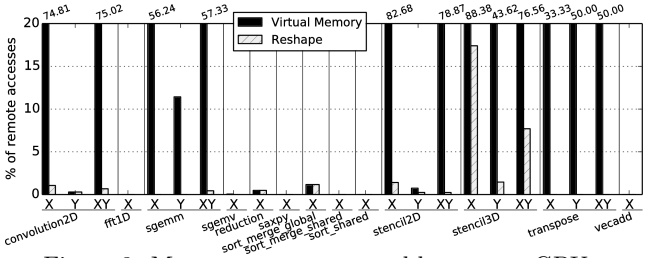
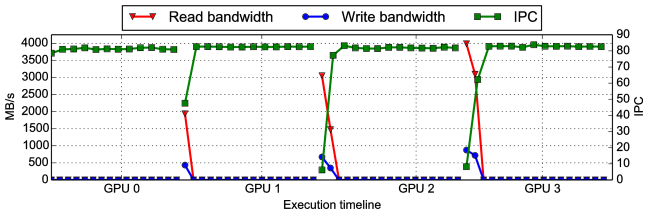
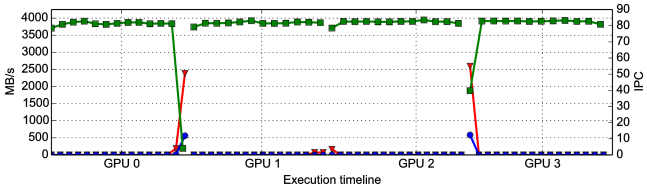


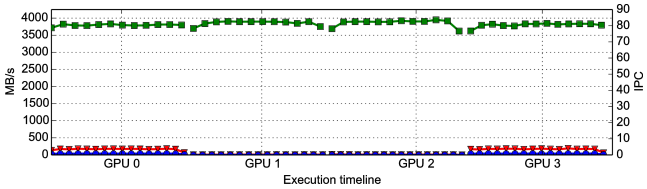

Figure 8: Memory requests served by remote GPUs.




(a) Imbalanced distribution




(b) Balanced distribution




(c) Balanced distribution + transposed thread block scheduling

Figure 9: Execution timeline of `stencil2D` for 4 GPUs.

lot of remote accesses in many configurations due to the memory allocation granularity in CUDA.

**Fighting data distribution imbalance in *VM***: The dimensions of the arrays in `stencil2D` and `stencil3D` kernels make it difficult to evenly split them using 1 MB granularity, causing imbalance and, therefore, remote memory accesses. The computation distribution configuration that we study is *Y*. Using this configuration, the volumes of `stencil3D` are distributed by allocating partitions of each plane alternatively in different GPUs. The size of each plane (1K×1K + halos) produces an imbalanced distribution (2/1/1/1MB for 4GPUs). This results in excessive communication that limits the performance (0.87× and 0.91× for 2 and 4 GPUs). Adding padding to the *X* dimension of the volume to obtain a balanced distribution results in a 127.01× memory footprint increment. Having a 4 KB granularity would reduce this overhead to 1.49×. Using more friendly problem sizes that do not produce imbalance results in much improved performance, reaching linear speedups of 2.08× and 3.95× for 2 and 4 GPUs.

The size (16K×16K + halos) of the plane in `stencil2D` allows for a more balanced data distribution (65/64/64/64MB for 4GPUs). However, there are still some effects on the performance. Figure 9a shows the memory bandwidth consumption due to remote loads and stores (left axis), and the number of instructions per cycle (right axis) during the execution of the `stencil2D` kernel for each GPU in the system. For the sake of clarity, we concatenate the execution timelines of the four GPUs, although they execute in parallel. GPU 0 does not perform any remote memory access and the IPC (instructions per cycle) remains stable during the kernel execution. GPUs 1, 2 and 3 perform remote memory accesses to the previous GPU memories (note that the imbalance increases with the GPU identifier). Remote accesses degrade the performance of the GPU as reflected in the lower IPC. Using padding to obtain a fully balanced array distribution would increase the memory footprint by 7.99×. Instead, we reduce the imbalance by offsetting the beginning of the array so that GPUs 0 and 3, and GPUs 1 and 2 perform the same amount of remote memory accesses (Figure 9b). The improved balancing lowers the remote memory bandwidth consumption (2.5 GBps vs 4 GBps), and the period in which remote accesses are performed is shorter. This results in a 8.2% execution speedup on the slowest GPU.

| Benchmark | Conf | Kim [18] | Lee [20] | AMGE |
|---|---|---|---|---|
| convolution2D | X | 38.7K×38.7K | 38.7K×38.7K | 77.4K×77.4K |
| | Y | 38.7K×38.7K | 38.7K×38.7K | 77.4K×77.4K |
| | XY | 38.7K×38.7K | 38.7K×38.7K | 77.4K×77.4K |
| fft1D | X | 750M | 750M | 3G |
| reduction | X | 2.99G | 2.99G | 11.99G |
| saxpy | X | 6G | 6G | 6G |
| sgemm | X | 31.6K×31.6K | 31.6K×31.6K | 44.7K×44.7K |
| | Y | 44.7K×44.7K | 31.6K×31.6K | 44.7K×44.7K |
| | XY | 38.7K×38.7K | 31.6K×31.6K | 48.9K×48.9K |
| sgemv | X | 77.4K×77.4K | 38.7K×38.7K | 77.4×77.4 |
| sort | X | 750M | 750M | 3G |
| stencil2D | X | 38.7K×38.7K | 38.7K×38.7K | 77.4K×77.4K |
| | Y | 48.9K×48.9K | 38.7K×38.7K | 77.4K×77.4K |
| | XY | 44.7K×44.7K | 38.7K×38.7K | 77.4K×77.4K |
| stencil3D | X | $1.1K^3$ | $1.1K^3$ | $1.8K^3$ |
| | Y | $1.3K^3$ | $1.1K^3$ | $1.8K^3$ |
| | XY | $1.2K^3$ | $1.1K^3$ | $1.8K^3$ |
| transpose | X | 48.9K×48.9K | 38.7K×38.7K | 77.4K×77.4K |
| | X | 48.9K×48.9K | 38.7K×38.7K | 77.4K×77.4K |
| | XY | 54.7K×54.7K | 38.7K×38.7K | 77.4K×77.4K |
| vecadd | X | 4G | 4G | 4G |

Table 2: Maximum problem size for a 4-GPU system in AMGE and in the related work.

**Reducing instantaneous bandwidth demands**: In `stencil2D`, memory accesses concentrate at the beginning/end of the kernel execution because the default thread block scheduler in the GPU issues thread blocks that are contiguous in the *X* dimension in order. Thus, thread blocks that access the boundaries of the matrix partitions tend to execute concurrently, increasing the instantaneous bandwidth demands and reducing the achieved IPC. We evaluate the performance of a thread block scheduler that issues thread blocks that are contiguous in the *Y* dimension instead. We emulate it by transposing the mapping of thread block identifiers on the matrices. Figure 9c shows that, using this scheduler, remote accesses are distributed throughout all kernel execution, thus reducing the instantaneous bandwidth demands to 200MBps. Now the IPC is not affected, since the cost of remote accesses is hidden with the execution of other thread blocks that only perform local accesses, reducing execution time by a 5.8%.

## 7.4 Comparison with previous works

**Memory footprint overhead**. AMGE performs more space-efficient data decompositions than previous works. We quantify the benefits of AMGE over Kim et al. [18] and Lee et al. [20] by comparing the maximum problem size that can be executed by the three solutions on our 4-GPU system.

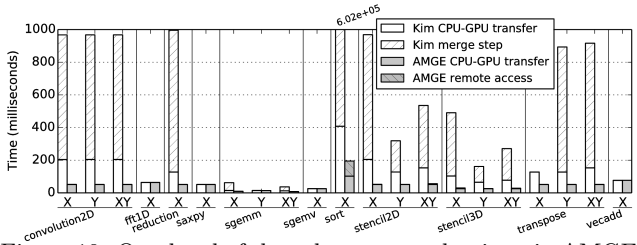


Figure 10: Overhead of the coherence mechanisms in AMGE and in the related work [18].

Table 2 shows that AMGE is able to run bigger problem sizes than previous works for most benchmarks, which is one of the major motivations for using multiple GPUs, especially on those that work on multi-dimensional arrays.

**Coherence overhead**: AMGE ensures coherence by not replicating output arrays and using remote memory accesses when needed. The related work relies on replication and a merge step after kernel execution. Besides, data needs to be copied from CPU to GPU memories before kernel execution. Figure 10 shows the overhead of both solutions. AMGE shows lower CPU/GPU transfer times in most cases (CPU-GPU transfer), because of the more efficient partitioning of multi-dimensional data structures, which requires smaller regions of the arrays to be resident in each GPU memory. The overhead of the merge step is even larger compared with the cost of remote accesses. The most extreme case is the `sort` benchmark, since it executes kernels iteratively and the merge step needs to be performed after each kernel call.

# 8. RELATED WORK

**Auto-parallelization in shared memory NUMA systems**: High Performance Fortran[19] provides primitives to distribute data and implements the *owner-computes rule* [8], that schedules loop iterations in such a way that communication is minimized. AMGE relieves programmers from specifying the distribution of data. Performance degradation due to remote access is much larger in GPUs than in CPU NUMA systems, and bad programmer choices might lead to large slowdowns. Therefore, providing a system that accomplishes this without programmer intervention is key for GPUs. Other proposals exploit architectural mechanisms to implement dynamic memory distribution policies (e.g., *first touch* placement) or data migration [7, 23, 21]. Nevertheless, current GPUs do not provide the necessary mechanisms (e.g., user-managed memory protection or appropriate performance counters) to implement these proposals. We rely on compiler analysis to minimize inter-GPU communication.

**Compiler-based transparent multi-GPU execution**: Kim et al. [18] introduce an OpenCL framework that combines multiple GPUs and treats them as a single compute device. In order to split data, they compute the array ranges accessed by computation partitions by performing sampling runs of the kernels on the CPU. The runtime system chooses the computation decomposition that minimizes the size of the data transfers between CPU and GPU memories. However, this only works for kernels in which array references are affine functions of the thread and thread block identifiers. Otherwise they fall back to replication. But even in cases where data can be decomposed, any array decomposition not performed on its highest-order dimension will produce tiles whose memory address ranges overlap, thus replicating big portions of the array in all memories. Array regions that are potentially modified from different computation partitions need to be merged after kernel execution. Lee et al. [20] extend the same idea to heterogeneous systems with CPUs and GPUs. They do not use the sampling runs on the CPU and generate a merge kernel that is more efficient than in Kim et al.; although both solutions require the merge step to be executed on the CPU, thus increasing the CPU↔GPU traffic. AMGE uses a similar approach but the compiler analysis uses the array dimensionality information by the `ndarray` type. Thanks to this information and the utilization of remote memory accesses, AMGE avoids replication in most cases, enabling bigger problem sizes and minimizing unnecessary CPU↔GPU communication. Moreover, AMGE adds support for cyclic data distributions, and exploits hardware support required by codes that use atomic operations and global memory fences.

**Language/Library-based transparent multi-GPU execution**: GlobalArrays [22] (GA) is a library that allows execution of computations across a distributed system using an array-like interface, and GA-GPU [29] extends GA to GPUs. The memory model in GA-GPU allows memory accesses to be ordered and, hence, it does not fit into bulk synchronous SPMD programming models such as CUDA or OpenCL. Therefore, GA-GPU recommends the utilization of GA data-parallel primitives (at the cost of lower performance due to the overhead of launching one kernel for each of the primitive operations). X10 [10] and Habanero [9] present the programmer with a single partitioned global address space. The compiler and the runtime system transparently redirect remote memory accesses to the proper memory. Sequoia [11] tries to address the problem of programming systems with different memory topologies/hierarchies. Programs are composed of two parts: (a) an algorithmic representation of the computation using a C-like programming language that decomposes data structures and defines how to map the computation on them; and (b) a mapping of the algorithm to the specific system using a declarative language. PGAS languages require a complete rewrite of the program, while AMGE requires minor code modifications.

# 9. CONCLUSIONS AND FUTURE WORK

Modern GPUs provide mechanisms that enable efficient auto-parallelization. In this paper we introduce AMGE, a programming interface, compiler support and runtime system that enables multi-GPU execution of computations written for a single GPU. Thanks to remote memory accesses, AMGE imposes much lower memory footprint and coherence management overheads than previous works. We also demonstrate that transparent data distribution can be efficiently implemented on current GPUs using the UVAS and compiler/runtime-assisted code versioning. Using the array data type provided in AMGE results in shorter and cleaner code, too. AMGE achieves almost linear speedups for most of the benchmarks on a real 4-GPU system with an interconnect with moderate bandwidth. Further performance improvements can be achieved by reducing the virtual memory mapping granularity exposed by CUDA and by allowing programmers to tune the thread block scheduling policy.

We believe that AMGE could be used in future systems such as NVIDIA Pascal boards to automatically scale the performance of GPU kernels to multiple GPUs. We plan to extend our evaluation to irregular computation patterns.

## Acknowledgments

We would like to thank the anonymous reviewers, on their comments and help improving our work. This work is supported by NVIDIA through the UPC/BSC CUDA Center of Excellence, the Spanish Government through Programa Severo Ochoa (SEV-2011-0067), and the Spanish Ministry of Science and Technology through the TIN2012-34557 project.

## 10. REFERENCES

[1] NVIDIA GPUDirect. `https://developer.nvidia.com/gpudirect`, 2012.

[2] APU 101: All about AMD Fusion Accelerated Processing Units, 2013.

[3] An Inside Look at Summit and Sierra Supercomputers. `http://info.nvidianews.com/CoralWhitepapers.html`, 2014.

[4] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. PPoPP '95, 1995.

[5] M. Araya-Polo, J. Cabezas, M. Hanzich, M. Pericas, F. Rubio, I. Gelado, M. Shafiq, E. Morancho, N. Navarro, E. Ayguadé, J. Cela, and M. Valero. Assessing Accelerator-Based HPC Reverse Time Migration. *TPDS*, 22(1), 2011.

[6] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance. SC '96, 1996.

[7] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA policies and their relation to memory architecture. ASPLOS IV, 1991.

[8] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2), 1988.

[9] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. PPPJ '11, 2011.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. OOPSLA '05, 2005.

[11] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. SC '06, 2006.

[12] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4), July 2008.

[13] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. ASPLOS XV, 2010.

[14] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *TPDS*, 3(2), Mar. 1992.

[15] IMPACT Group. Parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`, 2012.

[16] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *Micro, IEEE*, 31(5), 2011.

[17] The Khronos Group Inc. *The OpenCL Specification*, 2013.

[18] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. PPoPP '11, 2011.

[19] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The high performance Fortran handbook*. 1994.

[20] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. PACT '13, 2013.

[21] J. Marathe, V. Thakkar, and F. Mueller. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *Journal of Parallel and Distributed Computing*, 70(12), 2010.

[22] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A portable "shared-memory" programming model for distributed memory computers. SC '94, 1994.

[23] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. ICPP 2000, 2000.

[24] NVIDIA Corporation. *CUDA C Programming Guide*, 2015. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[25] S. L. Scott. Synchronization and communication in the T3E multiprocessor. ASPLOS VII, 1996.

[26] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling*, 29(2), 2010.

[27] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. D. Liu, W. mei W. Hwu, and N. Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *IEEE Computer*, 45(8), 2012.

[28] I. Tanasic, L. Vilanova, M. Jordà, J. Cabezas, I. Gelado, N. Navarro, and W.-m. W. Hwu. Comparison based sorting for systems with multiple GPUs. GPGPU-6, 2013.

[29] V. Tipparaju and J. S. Vetter. GA-GPU: extending a library-based global address space programming model for scalable heterogeneous computing systems. CF '12, 2012.

[30] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by NVIDIA. *Computing and Visualization in Science*, 13(1), 2010.