Automatic execution of single-GPU computations across multiple GPUs

[Extended Abstract]

Javier Cabezas[§], Lluís Vilanova[§], Isaac Gelado[⌽], Thomas B. Jablin[*]
Nacho Navarro[§‡], Wen-mei Hwu[*]
[§]Barcelona Supercomputing Center, [⌽]NVIDIA Corporation, [*]University of Illinois,
[‡]Universitat Politècnica de Catalunya
{first.last}@bsc.es, isaac@nvidia.com, nacho@ac.upc.edu, {jablin,w-hwu}@illinois.edu

ABSTRACT

We present AMGE, a programming framework and runtime system to decompose data and GPU kernels and execute them on multiple GPUs concurrently. AMGE exploits the remote memory access capability of recent GPUs to guarantee data accessibility regardless of its physical location, thus allowing AMGE to safely decompose and distribute arrays across GPU memories. AMGE also includes a compiler analysis to detect array access patterns in GPU kernels. The runtime uses this information to automatically choose the best computation and data distribution configuration. Through effective use of GPU caches, AMGE achieves good scalability in spite of the limited interconnect bandwidth between GPUs. Results show 1.95× and 3.73× 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. AMGE OVERVIEW

AMGE is a C++ programming framework to distribute data and GPU kernels to be collaboratively executed on all the GPUs in the system. This greatly simplifies the task of programming multi-GPU systems.

Figure 1 shows the components in AMGE and how they interact with the hardware. AMGE aggregates the GPU resources in the system and exposes them as a *single virtual GPU*. Thus, programmers are relieved from the burden of decomposing the problem and managing several GPUs.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.
ACM 978-1-4503-2809-8/14/08.
http://dx.doi.org/10.1145/2628071.2628109.

The AMGE compiler is a source-to-source compiler, based on the LLVM framework, that analyzes all CUDA kernels, detects all array access patterns and stores this information in the program executable. Array information is key for the compiler pass to produce meaningful information. However, C/C++ forces programmers to flatten dynamically-allocated multi-dimensional arrays into 1D arrays. This harms code readability and makes almost impossible for static analysis to extract the dimensionality and the array access patterns information. Thus, AMGE provides a new data type for multi-dimensional arrays that is used by the compiler analysis.

A key feature in 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 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. While this is a costly operation, GPUs' latency tolerance gives AMGE limited ability to absorb its overhead. This approach ensures that computation can always be decomposed and executed across multiple GPUs regardless the chosen data distribution. Moreover, this removes the requirement for the compiler analysis to precisely determine the bounds of the memory ranges accessed by a computation partition, and only provide information about the detected array memory access patterns in each GPU kernel.

On each kernel call, *the AMGE runtime* transparently determines the best computation grid and array decompositions using the compiler-provided access pattern information and distributes them across all GPUs.

Memory model: Arrays are decomposed and/or replicated before each kernel call. Input arrays can be safely replicated, but replication of output arrays requires special handling. After a kernel call, partial modifications in each copy need to be merged to provide a consistent view of the array, before using it in another kernel or in the host code. Previous solutions [4, 5] transfer all copies back to the CPU memory for a merge step, which imposes a large performance overhead in many workloads. We avoid this problem by not allowing output arrays to be replicated. Instead, output arrays are always distributed across GPU memories, and accessed through remote memory accesses if necessary.

AMGE implements the Asymmetric Distributed Shared Memory (ADSM [3]) model to allow arrays to be used both

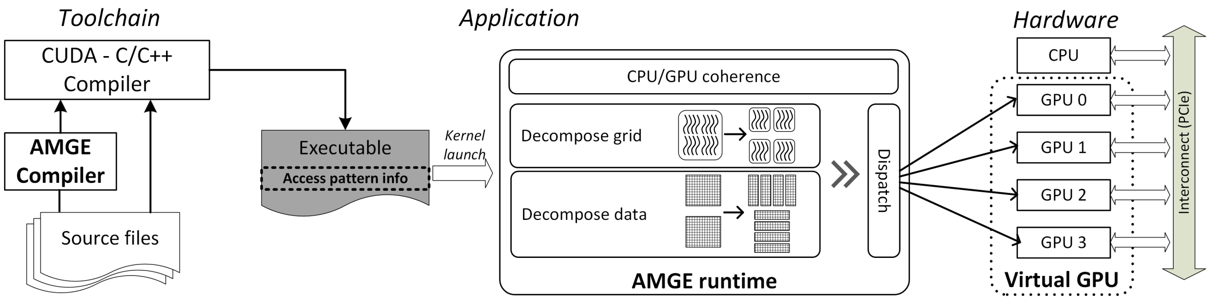

Figure 1: Overview of AMGE components. The compiler extracts array access pattern information and stores it in the program binary. The runtime system uses this information to decompose and distribute computation and data across the GPUs in the system.

```
 1  void sgemm(ndarray<float, 2, storage::cmo> C, ndarray<float, 2, storage::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: `sgemm` CUDA code using AMGE.

```
 1  // Initialize A and B in the host code
 2  ndarray<float, 2, storage::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, storage::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: `sgemm` host code using AMGE.

by host and GPU code. ADSM assumes a release consistency model in which allocations belong to the CPU code by default and are implicitly acquired/released by the GPU on kernel call boundaries. The runtime transparently transfers arrays between CPU and GPU memories as needed.

2. EXAMPLE: MATRIX MULTIPLICATION

Kernel code written for 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 (i.e., `sgemm`) computation using AMGE. This kernel is based on an efficient algorithm by Volkov and Demmel [2] that requires A and C matrices to be stored in *column major* order, and B in *row major* order. The grey background highlights the modifications to the original code. The only requirement for the kernel to be automatically decomposed is the utilization of 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 (*row major* by default). Indexing routines removes the need for index linearization on array accesses, resulting a more readable and compiler analysis friendly source code. 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 this pattern and stores it in the program executable so that it is accessible later by the runtime.

Listing 2 shows the CPU code of the `sgemm` computation. Input matrices `A` and `B` are declared in lines 2 and 3. Objects created from the `ndarray` template can be passed both to CPU and GPU routines. For example, they are passed by reference to the `read_array` routine (lines 5 and 6) to be initialized with the contents of a file. The `C` output matrix is declared and initialized using the sizes of the read matrices (line 8). These very same objects (i.e., matrices) are passed to the kernel call function. The AMGE runtime intercepts the kernel call, decomposes the computation grid and matrices using the information registered by the compiler, and distributes both computation and data across all the GPUs in the system. Finally, `C` is passed to the `write_array` routine by reference to be written to a file. It is worth noting that there are no explicit data transfers between host and GPU memories. The runtime transparently detects when data is used and needs to be transferred between them.

3. REFERENCES

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

[2] 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.

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

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

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