# Declarative Parallel Programming for GPUs

Eric HOLK William BYRD Nilesh MAHAJAN Jeremiah WILLCOCK
Arun CHAUHAN and Andrew LUMSDAINE
School of Informatics and Computing, Indiana University, Bloomington, Indiana, USA
{eholk,webyrd,nnmahaja,jewilco,achauhan,lums}@indiana.edu

**Abstract.** The recent rise in the popularity of Graphics Processing Units (GPUs) has been fueled by software frameworks, such as NVIDIA's Compute Unified Device Architecture (CUDA) and Khronos Group's OpenCL that make GPUs available for general purpose computing. However, CUDA and OpenCL are still low-level approaches that require users to handle details about data layout and movement across levels of memory hierarchy. We propose a *declarative* approach to coordinating computation and data movement between CPU and GPU, through a domain-specific language that we called *Harlan*. Not only does a declarative language obviate the need for the programmer to write low-level error-prone boilerplate code, by raising the abstraction of specifying GPU computation it also allows the compiler to optimize data movement and overlap between CPU and GPU computation. By focusing on the "what", and not the "how", of data layout, data movement, and computation scheduling, the language eliminates the sources of many programming errors related to correctness and performance.

**Keywords.** GPGPUs, declarative parallel programming, compilers

## Introduction

One of the most important developments in computing in the past few years has been the rise of graphics processing units (GPUs) for general purpose computing (also known as "GPGPU"). Driven by the demand for high-quality real-time graphics for video games, GPU performance has been increasing faster than that of conventional CPUs, including multi-core CPUs. A high-end GPU card can perform over a trillion floating-point operations per second (1 tera FLOP)—faster than the fastest supercomputer in 1997. More recently, "hybrid" clusters of CPUs and GPUs have become popular in the realm of high-performance computing (HPC). According to the November 2010 Top500 results, the fastest supercomputer in the world is Tianhe-1A, a Chinese hybrid cluster with a mixture of Intel CPUs and NVIDIA GPUs; the number three supercomputer is also a Chinese hybrid machine [18]. The popularity of hybrid architectures is likely to increase, since the primary design consideration for the next generation of supercomputers is energy efficiency, and since GPUs are much more energy efficient than CPUs (as measured in FLOPS per watt).

The introduction of NVIDIA's Compute Unified Device Architecture (CUDA) [14], and more recently the Khronos Group's OpenCL framework [13], along with GPU hard-

ware advances to better support non-image-based computing, has made GPGPU programming much easier than before. However, CUDA and OpenCL are still relatively low-level approaches to GPGPU programming. Programmers are required to write boilerplate code, handle low-level details of data layout and memory movement, determine how many blocks and threads are required for a computation, and so forth. Programming hybrid clusters is even harder, since cluster architectures complicate data movement, and introduce additional levels of memory hierarchy and computational granularity.

We propose a declarative approach to coordinating computation and data movement. It builds on our earlier declarative language to specify communication [11].

## 1. Declarative Foundations

GPGPU programming, and especially hybrid cluster programming, is complicated by many factors. For best performance, programmers must carefully manage low-level details of memory movement, strided memory access [12], and thread synchronization and management. When GPGPUs are used in hybrid clusters, programming becomes even more cumbersome. In order to fully leverage the power of GPUs, and especially hybrid clusters, a different approach is needed.

We advocate a declarative approach to programming hybrid clusters and GPUs. Our declarative approach provides the user with a straightforward mechanism for expressing the semantics the user wants for the data layout, memory movement, and computation coordination in her programs. The user can express the "what" but can leave the "how" to the tool developers, avoiding the myriad details described in the previous section.

The two pillars of our approach are the following:

- Development and analysis of a declarative "computational kernel" language as an approach for coordinating computation, data layout, and memory movement within a single machine containing GPUs; and
- Integration of the kernel language within Kanor, our declarative language for cluster programming.

Our approach leverages our current work on Kanor, a declarative language for specifying communication on distributed-memory clusters. Kanor is unusual in that the programmer declaratively, but explicitly, specifies the essence of the communication pattern. The programmer lets the implementation handle the details when appropriate, but retains the option to hand-encode communications when necessary, providing a balance between declarativeness and performance predictability and tunability.

Similarly, our computational kernel language (named "Harlan") allows the user to declaratively, but explicitly, describe (potentially asynchronous) computational kernels and to coordinate computation, data layout, and memory movement. As with Kanor, this approach gives the programmer enough control to write efficient code, while abstracting over the low-level details that make GPU programming so difficult. Integrating Harlan into Kanor results in a unified, high-level, flexible language suitable for efficiently programming hybrid clusters, traditional (CPU-based) clusters, and GPUs on a single machine.

Declaratively specifying data layout, memory movement, and computation coordination requirements results in a system with well-defined semantics. Thus, for instance,

interactions between data movement and computation can be automatically verified. Common GPU programming mistakes (e.g., deadlock through incorrect use of synchronization constructs [3,17]) can thereby be avoided.

Moreover, a declarative approach with well-defined semantics provides opportunities for sophisticated optimizations, analyses, and tools. For example, the implementation could use a combination of heuristics and autotuning to determine how many stages of a reduction (if any) should be performed on the CPU rather than on the GPU, depending on the specific machine's hardware. An example of another optimization would be double or triple buffering to hide direct memory access (DMA) latency when moving data from the GPU to main memory—this optimization could be performed automatically by the compiler, perhaps using guidance from programmer declarations.

It is important to emphasize at this point that we are not proposing a "silver bullet" or "magic compiler" that will somehow make GPGPU or hybrid cluster programming easy. Rather, we are seeking to abstract away many of the low-level details that make GPU/-cluster programming difficult, while still giving the programmer enough control over data arrangement and computation coordination to write high-performance programs.

## 2. A Declarative Language

As described in Section 1, a CUDA or OpenCL programmer must handle a variety of low-level details that have nothing to do with the problem domain. For example, consider the CUDA code in Figure 1, which sums two vectors; this code is verbose, containing boiler-plate code for moving data to and from the GPU, calculating thread indices, and so forth. Ideally, the programmer could ignore these details and write something like the code in the top part of Figure 2, which expresses the desired kernel computation and data movement much more succinctly. This code snippet indicates that the expression `z = x + y` runs on the GPU for each `x`, `y` and `z` in the vectors `X`, `Y` and `Z`. The end result is that `Z` contains the sum of vectors `X` and `Y`. This level of expression allows the compiler to automatically perform transformations, such as pipelining, depending on the size of the vectors.

The compiler translates the specification in Figure 2 into low-level code in CUDA or OpenCL. In this section we describe our approach to Harlan's design, implementation, and optimization.

### *2.1. Design*

Harlan enables the programmer to specify sections of code to run on the GPU or other accelerator over certain ranges of data. This level of expression gives the programmer control of where the computation takes place, and implicitly defines what data must move and when such movement must occur. However, the compiler and runtime maintain a great deal of flexibility to perform data layout transformations or optimizations such as pipelining.

One key issue in programming GPUs is managing whether data is resident on the device or host memory. This is especially difficult in CUDA, as the language does not make any distinction at a language level between data on the GPU and data on the CPU—both are represented as pointers. Our kernel blocks, on the other hand, indicate syntactically

which portions of code should run on the GPU if possible, and thus imply what data must be on the GPU. In the most naïve sense, all the data needed by a kernel is moved to the GPU upon entering a kernel expression and the data is moved back afterwards. In practice, however, the compiler may use dataflow analysis to eliminate unnecessary data movement. Freeing the programmer from worrying about these details also lowers the potential for errors that arise from, for example, dereferencing a device pointer from the host or vice-versa.

We have designed kernels so that they are expressions that return values. This decision improves expressiveness and compositionality. Using kernels as expressions allows rewriting the first example in Figure 2 as: Z = **kernel**(x : X, y : Y) { x + y }. Kernel expressions are similar to the map operator in functional programming languages. We also provide support for reductions, enabling programming styles such as MapReduce [8]. For instance: Z = +/**kernel**(x : X, y : Y) { x * y }. Here, the values returned by the kernel are to be summed and stored in `z`. This approach gives the programmer more control and is more natural when nesting reductions. Making this reduction information explicit provides more information for the compiler to use in optimizations.

Our goal is to avoid restrictions on what code is allowed within a kernel block. We allow arbitrary procedure calls within kernels, including recursive calls. This implies the ability for kernels to be nested, since even if we tried to restrict this, kernels might still call functions that use kernels. This eliminates the cognitive burden caused by constructs

```
__global__ void add_kernel(int size, float *X, float *Y, float *Z)
{
  int i = threadIdx.x;
  if(i < size) { Z[i] = X[i] + Y[i]; }
}

void vector_add(int size, float *X, float *Y, float *Z)
{
  float *dX, *dY, *dZ;
  cudaMalloc(&dX, size * sizeof(float));
  cudaMalloc(&dY, size * sizeof(float));
  cudaMalloc(&dZ, size * sizeof(float));

  cudaMemcpy(dX, X, size * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(dY, Y, size * sizeof(float), cudaMemcpyHostToDevice);

  add_kernel<<<1, size>>>(size, dX, dY, dZ);

  cudaMemcpy(Z, dZ, size * sizeof(float), cudaMemcpyDeviceToHost);

  cudaFree(dX);
  cudaFree(dY);
  cudaFree(dZ);
}
```

**Figure 1.** CUDA code for adding two vectors.

```
void vector_add(vector<float> X, vector<float> Y, vector<float> Z)
{ kernel(x : X, y : Y, z : Z) { z = x + y; } }
```

---

```
total = +/kernel(row : Rows) { +/kernel(x : row); };
```

---

```
handle = async kernel(x : X, y : Y) { x * y };
/* Other concurrent kernels or program code go here. */
z = +/wait(handle);
```

---

```
kernel(x : X, y : Y, z : Z) { z = x * y; }
@communicate {Y[i]@r <<= Z[i]@((r + 1) % NUM_NODES),
  where r in world, i in 0...length(Y)}
kernel(x : X, y : Y, z : Z) { z = x * y; }
```

**Figure 2.** Harlan code for adding two vectors, sum reduction using nested kernels, asynchronous kernels, and kernels interspersed with communication across a cluster.

such as CUDA's local and global functions and allows programs such as the second example in Figure 2.

In the most basic form, a **`kernel`** block provides language support for data parallelism. Allowing multiple kernels to run concurrently can provide more flexibility in expressing algorithms and gives more freedom to the compiler for scheduling. Modern GPUs, such as those using NVIDIA's Fermi architecture, support running multiple kernels in parallel [15]. Harlan allows an optional **`async`** annotation on kernel expressions. The expression returns a handle that the program could then wait on. Figure 2 shows an example of using an **`async kernel`**. In this example, the kernel may run asynchronously from the rest of the main program. The **`wait`**`(handle)` expression blocks until the kernel completes and returns the value that would have been returned by the kernel were it executed synchronously. This added flexibility can simplify the expression of task parallelism.

### *2.2. Implementation*

Perhaps the biggest language design challenge is determining how to effectively map our language features onto hardware. One difficulty is that GPUs currently have more limited control flow options than CPUs. In effect, they support only small branches and looping constructs; richer concepts like recursion natively supported. Many languages address this by limiting the expressiveness of GPU kernels. We intend to avoid these restrictions. Kernels should be allowed to run arbitrary computations, including recursive procedure calls and spawning additional kernels. Launching kernels from within kernels leads to nested data parallelism, which has been the subject of much existing research [2,16,6]. NESL [2,1] implements nested parallelism primarily through flattening. More recent work argues that except in the most unbalanced workloads, leaving the program in its original nested form provides more opportunities to exploit hierarchies in modern machines [5]. More evaluation is clearly necessary, especially in the deeper hierarchies present in hybrid GPU cluster computers.

Existing work has demonstrated that it is possible to support richer control flow on a GPU [9]. The approach is to write a SIMD interpreter that interprets different programs as data. Naturally, this incurs a certain amount of overhead, although the results suggested that this overhead will be acceptable in many cases. We propose a hybrid approach, generating native GPU code when possible, and using the interpreter approach for particularly difficult kernels. We expect that future GPUs will continue to relax the restrictions on program control flow.

Data movement within a hybrid cluster introduces more opportunities. Soon it will be common for GPUs to be able to interact with network hardware directly, without involvement from the CPU. While a naïve implementation of Harlan with Kanor `@communicate` blocks would always copy data from the GPU to the CPU before doing network transfers, this is unnecessary. The dataflow analysis that is used to optimize data movement between the CPU and GPU memory can also inform communication code generation to produce direct GPU to GPU transfers even across nodes in a cluster. The last example in Figure 2 shows communication code between two kernels. The program first performs a computation, then all nodes exchange data, and the computation continues. None of the variables `X`, `Y` and `Z` are accessed off of the GPU between the two kernels, so there is no reason to move them off of the GPU. Instead, the communication can directly transfer between the GPU and network.

### *2.2.1. Optimizations*

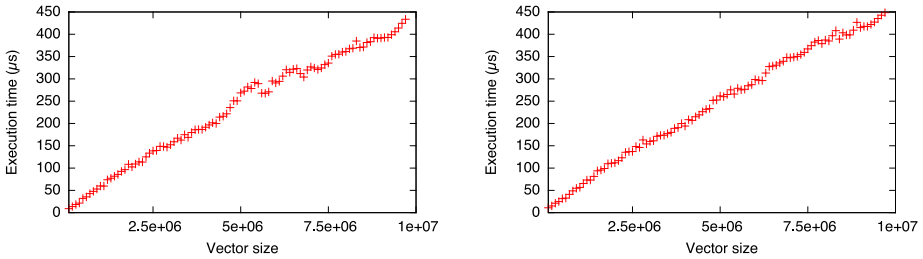
*Data Movement* Since data movement between CPU and GPU is implicit, the compiler must infer when data need to be copied. Data that are not live at the end of a kernel need not be copied back into the CPU. Similarly, data that would be used only by a subsequent kernel may be kept on the GPU. However, this must be balanced against the GPU memory footprint of the application.

A second class of data locality optimizations relate to various memory types on contemporary GPUs, as mentioned earlier. For instance, the compiler can identify read-only data that are live and expected to be used soon, to allocate those in the faster constant memory.

*Splitting Kernels* Kernels defined in Harlan may often map directly to kernels in CUDA or OpenCL, but they are not required to. CUDA and OpenCL kernels have restrictions on control flow and procedure calls, and lack synchronization capabilities. Thus, the compiler may need to split a kernel in Harlan in order to implement it using CUDA kernels [4]. Multiple splits may be possible, so the compiler must select the one that is likely to minimize performance penalty.

*Scheduling Concurrent Kernels* Carefully scheduling the kernels has the potential to dramatically improve data locality and avoid unnecessary CPU-GPU data movement. A data dependence graph between kernels, with edge weights representing the amount of shared data, can provide the compiler the necessary information to create a schedule.

*Generating Code for Reduction Operations* Reduction operations may be more effectively done on the CPU, even if the partial results need to be transferred from the GPU, because low occupancy in the later parts of a reduction operation can result in suboptimal GPU utilization. However, the data movement cost might dominate if the results are to be used back on the GPUs. The compiler will need to consider this tradeoff and




(a) Vector addition (b) Dot product

**Figure 3.** Performance of code generated by prototype Harlan compiler.

may choose to generate code that employs a hybrid strategy of using both CPU and GPU for completing a reduction operation.

## 3. Evaluation

As proof of concept, we studied three benchmarks in Harlan, vector addition, vector dot product, and Mandelbrot set generation. We evaluated their performance on 2.8 GHz Quad-Core Intel Xeon with 8 GB 1066 MHz DDR3 RAM and ATI Radeon HD 5770 graphics processors with 1024 MB memory, running Mac OS X Lion 10.7.1. Figure 3 shows the GPU running times of OpenCL generated by our compiler for two of the benchmarks, vector addition and dot product, for increasing vector sizes. Figure 4 shows running times of compiler-generated OpenCL code for Mandelbrot on CPU and GPU. Unsurprisingly, running OpenCL on GPUs is faster than running the OpenCL on CPU. The slight discontinuity of Mandelbrot times on the GPU seems to be an artefact of the ATI Radeon's memory-hierarchy optimization called *fastpath*.

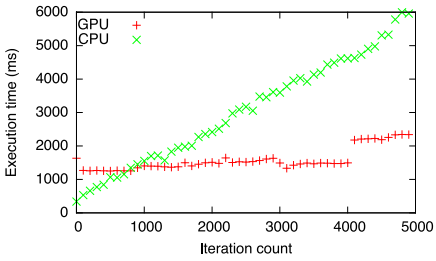



Figure 4.: Mandelbrot on CPU and GPU.

## 4. Conclusion

General-purpose GPU programming is arduous on multiple fronts, requiring programmers to manually program data layout, memory accesses, and boiler-plate code, among other details. Even with state-of-the-art technologies such as CUDA or OpenCL, attention to such detail is necessary to enable GPUs to achieve their performance potential. At the same time, data orchestrations necessary for optimal GPU access may not be human-friendly, or even directly related to the problem being solved. For example, strided memory accesses may incur an order-of-magnitude performance penalty compared to unit strides. Similarly, data movement from the GPU to main memory may require double or triple buffering to hide DMA latency. Ideally, such data access issues should be managed automatically by the compiler, assisted by programmer declarations.

In this paper we introduced Harlan, which affords a *declarative approach to GPGPU programming*, allowing users to specify the "what" not the "how" of data layout, data movement, and computation scheduling and coordination. Consider the normally difficult task of setting up blocks of threads on a GPU for optimal efficiency. Using a declarative approach, the programmer can specify what computation needs to be performed, but let the language and runtime determine "how" the computation should be broken into thread blocks (perhaps using a combination of autotuning and heuristics).

The declarative approach is especially promising for hybrid CPU/GPU clusters, in which movement of data between compute nodes adds even more complexity. Our end goal is a system that removes much of the "accidental" or "artifactual" burden of programming large-scale hybrid resources (such as Roadrunner or Tianhe-1A), without sacrificing performance.

## References


[1] G. E. Blelloch. Nesl: A nested data-parallel language (version 3.1). Technical report, Pittsburgh, PA, USA, 1995.

[2] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

[3] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems*, Apr. 2008.

[4] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for GPGPU. In *Proceedings of the 6th ACM Symposium on Computing Frontiers*, pages 147–150, 2009.

[5] B. C. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In C. Cascaval and P.-C. Yew, editors, *PPOPP*, pages 47–56. ACM, 2011.

[6] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal - nested data parallelism in haskell. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 524–534, London, UK, 2001. Springer-Verlag.

[7] K. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. *Research Directions in Concurrent Object-Oriented Programming*, Jan 1993.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] H. Dietz and B. Young. Mimd interpretation on a GPU. In G. Gao, L. Pollock, J. Cavazos, and X. Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 65–79. Springer Berlin / Heidelberg, 2010.

[10] I. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems (TOPLAS*, 18(4), Jul 1996.

[11] E. Holk, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine. Kanor – A Declarative Language for Explicit Communication. In *Thirteenth International Symposium on Practical Aspects of Declarative Languages (PADL'11)*, Austin, Texas, Jan. 2011.

[12] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22:105–118, 2011.

[13] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1*, September 2010.

[14] NVIDIA Corporation. *NVIDIA CUDA Reference Manual, version 3.2 Beta*, August 2010.

[15] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, version 4.0 edition, Feb. 2011. Included with CUDA 4.0 SDK release candidate.

[16] S. Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, July 2010.

[18] Top500.org. Top500 list, November 2010. http://www.top500.org/lists/2010/11.