LA-UR- 11-00593

Approved for public release;
distribution is unlimited.

Title: Scout: High-Performance Heterogeneous Computing Made Simple

Author(s): James Jablin, Patrick McCormick, Maurice Herlihy

Intended for: IPDPS 2011 PhD Forum

Los Alamos
NATIONAL LABORATORY
EST.1943

Scout: High-Performance Heterogeneous Computing Made Simple

Grad: James A. Jablin, 3^{rd} year
Brown University
jjablin@cs.brown.edu

Advisor: Patrick McCormick
Los Alamos National Laboratory
pat@lanl.gov

Advisor: Maurice Herlihy
Brown University
mph@cs.brown.edu

***Abstract*—Researchers must often write their own simulation and analysis software. During this process they simultaneously confront both computational and scientific problems. Current strategies for aiding the generation of performance-oriented programs do not abstract the software development from the science. Furthermore, the problem is becoming increasingly complex and pressing with the continued development of many-core and heterogeneous (CPU-GPU) architectures. To achieve high performance, scientists must expertly navigate both software and hardware. Co-design between computer scientists and research scientists can alleviate but not solve this problem. The science community requires better tools for developing, optimizing, and future-proofing codes, allowing scientists to focus on their research while still achieving high computational performance.**

Scout is a parallel programming language and extensible compiler framework targeting heterogeneous architectures. It provides the abstraction required to buffer scientists from the constantly-shifting details of hardware while still realizing high-performance by encapsulating software and hardware optimization within a compiler framework.

***Keywords*-GPU; high-performance; parallel programming language; compilers**

I. Introduction

Advances in hardware trends of multicore and heterogeneous architectures have outpaced software development. Previously, programmers could expect increasing performance without intervention based on regular increases in clock frequency. Clock frequency increases eventually stopped due to concern about heat dissipation while Moore's Law continued unabated. Consequently, chip manufacturers used the extra transistors to create multiple cores without increasing overall clock speed rather than a single faster core. The advent of multicore architectures drastically changed the architectural landscape. The number of cores has increased while clock frequency has either remained constant or decreased.

These radical architectural changes are supported by improved CPU performance. Results achieved on the GPU are equally impressive [1]. Unfortunately, high performance on multicore and heterogeneous architectures often demands expert manual optimization. Programs written in current programming languages are a challenge for automatic parallelization and provide no hardware abstraction. Static compiler optimizations produce conservative results. Without hardware abstraction the programmer must acquire target architecture knowledge to aid optimization. Programmers require better programming abstractions for developing, optimizing, and future-proofing codes.

The most common programming languages for scientific workloads are C/C++ and Fortran. Invented in a time when single-core CPUs dominated, these languages contain constructs to improve single-core performance but foil autoparallelization techniques. For example in C/C++, pointer manipulation and subversive typecasting thwart alias analysis creating missed opportunities for optimization. Inadequate programmer education about parallel processing compounds the problem, yielding programs that defy static optimization.

A programming language should reflect the ease of use and intuitiveness of sequential programming and still admit directed compiler optimization and parallelization. Domain-specific languages have limited use cases. Parallel programming languages deliver a highly parallel and scalable programming model but not one has gained widespread appeal. Sequential languages with parallel extensions better exploit concurrency but still contain sequential language pitfalls.

For developing GPU codes, programmers use CUDA C [10] or OpenCL [4]. Both require explicit management of data between CPU and GPU, a tedious and error-prone process. Management of linked-data structures poses an even greater challenge than arrays, further limiting utility. CUDA C and OpenCL are extensions of C. As such the programmer must balance between avoiding C's problems and exploiting the performance potential of the GPU.

This paper presents Scout, a parallel language and extensible compiler framework. It shares similar concepts and the same name with its predecessor [8]. New contributions focus on a refactored programming language and completely redesigned compiler framework and optimization techniques. Scout combines the effective strategy of explicitly identifying concurrency with program directives with an extensible compiler framework for profiling, statically optimizing, and tuning. These new modifications enable Scout to provide a high-level programming abstraction and still generate efficient executable code for heterogeneous architectures.

II. Scout Programming Language

The Scout parallel programming language facilitates analysis and simulation of computationally intensive data sets by providing constructs for describing concurrency while abstracting architectural details and data management. It is

Listing 1: CUDA 2D Heat Simulation

```
■ __global__ void heat2d(unsigned N, float heatIn[N][N],
■                                   float heatOut[N][N]) {
■   int row = blockIdx.x * blockDim.x + threadIdx.x;
■   int col  = blockIdx.y * blockDim.y + threadIdx.y;
■   if(row < N && col < N) {
■     float delta = 0.0;
■     if(row > 0)    delta += heatIn[row    ][col+1];
■     if(col < N-1)  delta += heatIn[row    ][col-1];
■     if(row < N-1) delta += heatIn[row+1][col    ];
■     if(col > 0)    delta += heatIn[row-1][col    ];
■     heatOut[row][col] = heatIn[row][col] + 0.25 * (delta -
■                                   4 * heatIn[row][col]);
■   }
■ }
    __host__ void main(unsigned COUNT) {
      /* Declare 2D array of data structure of 1,048,576 elements */
      float heatIn[1024][1024], heatOut[1024][1024];

      /* Initialize heatIn and heatOut*/
      ...

      /* Malloc memory on the GPU */
      float **d_heatIn, **d_heatOut;
      int const N_BYTES = 1024 * 1024 * sizeof(float);
⧄     cudaMalloc(d_heatIn, 1024);
⧄     cudaMalloc(d_heatOut, 1024);

      /* Copy 2D arrays from the CPU to the GPU */
⊠     cudaMemcpy(heatIn, d_heatIn, N_BYTES,
⊠                     cudaMemcpyHostToDevice);
⊠     cudaMemcpy(heatOut, d_heatOut, N_BYTES,
⊠                     cudaMemcpyHostToDevice);

      for(unsigned i = 0; i < COUNT; ++i) {
        /* Define grid and block size to generate 1,048,576 threads */
        dim3 grid(64, 64, 0), block(16, 16, 0);

        /* Perform 2D heat simulation */
■       heat2d<<< grid, block >>>(1024, d_heatIn, d_heatOut);

        d_heatIn = d_heatOut;
      }

      /* Copy the 2D array back from GPU to CPU */
⊠     cudaMemcpy(d_heatOut, heatOut, N_BYTES,
⊠                     cudaMemcpyDeviceToHost);
      /* Free the 2D arrays */
⧄     cudaFree(d_heatIn);
⧄     cudaFree(d_heatOut);
    }
```

■ GPU kernel ⊠ CPU-GPU communication ⧄ Memory (de)allocation

Listing 2: Scout 2D Heat Simulation

```
    void main(unsigned COUNT) {
      /* Declare 2D array of data structure of 1,048,576 elements*/
      uniform grid Grid[1024,1024];
      /* Declare a 2D variables of type float */
      float@cell Grid:Grid heatIn, heatOut;

      /* Initialize heatIn and heatOut*/
      ...

      for(unsigned i = 0; i < COUNT; ++i) {
        /* Perform 2D heat simulation */
■       forall cells in Grid
■         heatOut = heatIn + 0.25 * (north(heatIn, 0.0) +
■                                     south(heatIn, 0.0) +
■                                      east(heatIn, 0.0) +
■                                      west(heatIn, 0.0) -4 * heatIn);
        heatIn = heatOut;
      }
    }
```

■ GPU kernel ⊠ CPU-GPU communication ⧄ Memory (de)allocation

an imperative language providing strong type information, an abstract data layout, and explicit concurrency constructs.

Listing 1 is an example program written in CUDA and Listing 2 is an example of the same program written in Scout. The code in the listings simulates two-dimensional heat transfer of a point source. When compiled, both codes execute on the GPU with equivalent performance. Comparing listings highlights Scout's ability to hide architectural detail behind an abstract yet descriptive language. Scout programs are easier to program, debug, and maintain without loss of performance.

In Listing 1, memory for GPU variables must be *malloc*'ed. Next, there is code to copy the variables from CPU to GPU. The number of GPU threads per grid and per block is declared. Then GPU kernel *heat2d* is executed. Finally, the results are copied from GPU to CPU and GPU memory is freed.

The Scout version in Listing 2 is noticeably more concise. Scout abstracts the tasks related to GPU initialization and CPU-GPU communication. Initially, a Scout **grid** type is defined. Scout **grid** types define a multidimensional statically allocated data structure. The rank of each dimension need not be the same. Next, the type of each grid element is declared. The **forall** keyword identifies an explicitly concurrent block of code. The *Grid* type beside **forall** indicates that lines nested within **forall** execute on all 1,048,576 elements.

In addition, the Scout GPU kernel is slimmer and more abstract compared to the CUDA GPU kernel. The GPU kernel of each listing is marked with black squares. Notice the declaration of GPU threads in the CUDA GPU kernel and the absence of thread declarations in the Scout example. In the CUDA example, perimeter elements must be explicitly handled. By contrast, Scout's GPU kernel implicitly handles GPU threads, and the corner cases of the perimeter elements are defined with the intrinsic stencil operations: *north*, *south*, *east*, and *west*. Stencil operations perform a query of a neighbor element's value without explicit reference to data structure. The second parameter represents the value returned if an element beyond the dimension's size is accessed. Scout also includes a circular version of handling out-of-bounds access. Abstracting data reference enables optimization of data structure to target architecture.

Scout leverages an efficient and powerful programming abstraction to allow the programmer to easily write programs for heterogeneous architectures. The following section details the work performed in the compiler to transform a Scout program into a high-performing executable.

III. Scout Toolchain

Scout bootstraps from the LLVM [5] compiler infrastructure. LLVM provides common compiler parts as modules assembled based upon user discretion. A schematic diagram of Scout's toolchain appears in Figure 1. The current status

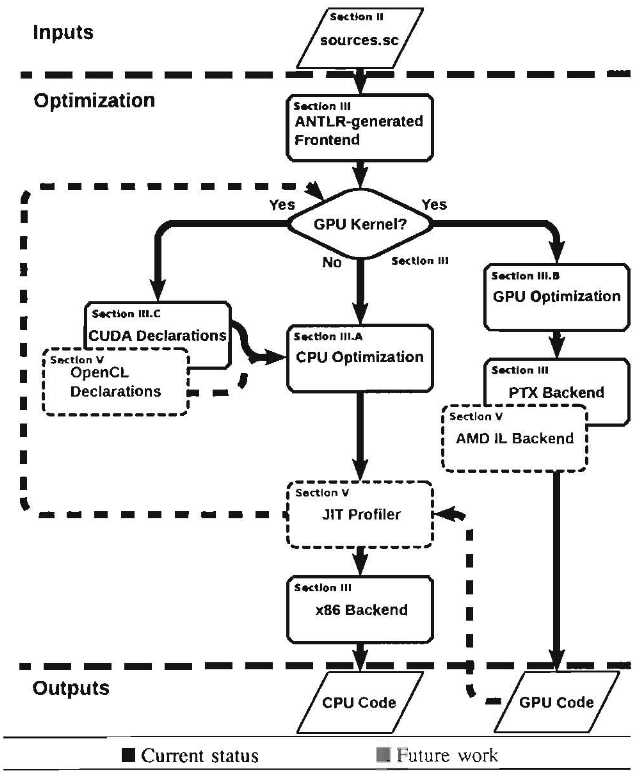

Figure 1. Scout toolchain

of the toolchain is marked in solid black, and future work is marked in dashed grey. Implementing Scout in LLVM facilitates portability. LLVM maintains many CPU architecture backends. Future work on Scout will expand the number of GPU backends.

The Scout compiler takes a Scout source program, like Listing 2, as input and outputs executable CPU and GPU code. The frontend parses and translates the Scout program into LLVM assembly language, the input and output of all LLVM modules. Sections of concurrent code in a Scout program amenable to GPU execution are identified within the frontend.

Kernel identification partitions LLVM assembly into GPU-bound and CPU-bound codes. Each code receives target-specific optimization. For GPU kernels CUDA declarations are automatically inserted based on each GPU kernel's characteristics. After GPU optimization, LLVM assembly is lowered to PTX by Scout's PTX backend, a branch of Rhodin's backend [11]. PTX is the intermediate assembly language for NVIDIA GPUs. Likewise, after CPU optimization, LLVM assembly is lowered to the target CPU architecture.

A. CPU Optimization

Common compiler optimizations are applied to the CPU code. Because the bus connecting CPU and GPU has characteristics of high bandwidth and high latency, GPU kernels tend to dominate runtime, and the work of the CPU is often restricted to copying data and starting GPU kernels. Accordingly, minimizing the number of CPU-GPU communications, rather than the volume of data communicated, will achieve best performance. Consequently, improvements to performance for heterogeneous architectures reduce the CPU workload to CPU-GPU communication and synchronization.

B. GPU Optimization

The important factors for maximizing GPU kernel performance are coalescing memory operations, removing control flow, and distributing work to all GPU cores.

All these factors accentuate the GPU's single-instruction, multiple-thread (SIMT) architecture and are more fully discussed in "CUDA C Best Practices" [9]. The compiler supports the different kinds of GPU memory. The data layout of the GPU kernel's arguments is first optimized for coalesced memory access for the GPU. Coalesced memory operations achieve maximum throughput for the same latency as a single memory access. This optimization is possible by transforming the variable's data footprint in memory. In C/C++, changing the data layout of variables and enforcing pointer arithmetic semantics is impractical.

The GPU's SIMT architecture forces every GPU thread to execute in lock step. Control flow within a GPU kernel will degrade performance by introducing branch divergence. Only threads that succeed the conditional will execute while the rest idle until the successful threads finish the branch. Removing control flow from within a kernel prevents branch divergence.

Distributing work between GPU cores hides latency by overlapping processes stalled on memory accesses with processes able to compute. Uncoalesced memory operations and under-utilization of GPU cores have severe performance implications for a memory-bound kernel.

Finally, variables with high dimensionality present a difficult problem for efficient memory access. Scout has a special optimization for this case. The variable is tiled into two-dimensional blocks. This process has the added benefit of enabling better cache coherence.

C. CUDA Declarations

The CUDA declarations facilitate GPU kernel management. As shown in Listing 1, CUDA declarations allocate and deallocate GPU memory and provide routines for CPU-GPU communication.

Inefficient CPU-GPU communication patterns result in poor overall performance. Where possible the compiler optimizes communication to minimize redundant copies. Also, the compiler transforms cyclic CPU-GPU communication patterns into acyclic ones.

CUDA declarations set the number of threads per grid and per block. Poor choices of grid and block size decrease GPU utilization and consequently decrease performance. The compiler sets the number of threads based upon kernel arguments. Programmers have difficulty choosing an appropriate thread count because the choice depends on code and GPU architecture. Automatically defining thread count relieves this burden from the programmer. As shown in Listing 1, branch statements must be manually calculated and inserted in the kernel to prevent out-of-bound errors. The Scout compiler automates this process.

IV. Results

Collaborations are underway with several groups from Los Alamos National Laboratory to use Scout to aid in analysis and visualization of large-scale data sets. With the Physics and Chemistry of Materials group, short-range molecular dynamics codes are being adapted to run on a GPU cluster. Additionally, with the cooperation of the Space Science and Applications, Nuclear and Particle Physics, and Astrophysics and Cosmology groups work continues on cosmology codes.

The Rodinia Benchmark Suite [3] is composed of programs with parallelized versions using OpenMP and CUDA for multicore CPUs and GPUs respectively. Work continues manually porting the CUDA versions to Scout and comparing performance. Results are encouraging.

V. Future Work

Scout eases the transition for programmers to leverage GPUs for high-performance. Future work will continue to improve performance and portability. Additions to Scout's toolchain include an AMD IL backend, OpenCL declarations, and a JIT profiler. An AMD IL backend and OpenCL declarations would enable Scout to target AMD GPUs. AMD IL is analogous to PTX but for AMD GPUs. It is the intermediate assembly language for AMD GPUs.

LLVM has a module for JIT compilation of LLVM assembly. As depicted in the Scout toolchain diagram in Figure 1, the JIT could be used to profile and tune GPU kernels for performance based on the number of threads allocated per grid and per block. Profiling could also identify CPU codes sandwiched between GPU codes causing a long latency chain. Lowering these sandwiched CPU codes to the GPU will improve performance by removing communication. To further enhance performance, kernel fusion and fission optimizations would improve cache coherence via locality of reference.

Currently, the Scout compiler does not statically analyze programs for parallelizable loops, relying solely on Scout's concurrent annotations. Prior work to automatically detect parallelizable loops in programs complements Scout's annotations and could further improve performance.

Besides high computational performance, data visualization is also important. The Scout programming language will integrate performance and visualization directives, providing facilities for analysis and simulation of computationally intensive data sets.

VI. Related Work

Prior work on automatic parallelization for GPUs focuses on improving codes written in a mixture of C and CUDA. No prior work abstracts details about concurrency or memory management. For programs written in CUDA, CUDA-lite [13] improves GPU kernel performance by optimizing GPU memory access. Using the polyhedral model "C-to-CUDA for Affine Programs" [2] and "A mapping path for GPGPU" [7] optimize C codes into efficient CUDA C. For a less portable solution, "OpenMP to GPGPU" [6] translates programs annotated with OpenMP pragmas to CUDA C. The PGI Fortran and C compiler [12] advertises semi-automatic GPU parallelization. Users must mark loops and the optimization is not tolerant to general pointer arithmetic.

VII. Conclusion

Current programming languages and tools divert attention away from science to software development. Scout showcases how a mix of language abstraction and compiler automation result in high-performance without the burden of manual optimization. Using an explicitly parallel language with few universal primitives results in a less computationally intensive analysis and broader application of parallelization techniques. Future work focuses on optimization opportunities not targeted by the current framework and improvements to portability.

References

[1] CUDA Community Showcase. http://www.nvidia.com/object/cuda_showcase_html.html.

[2] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In R. Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*. Springer, 2010.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, pages 44–54, October 2009.

[4] Khronos OpenCL Working Group. *The OpenCL Specification*, September 2010.

[5] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[6] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2009.

[7] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61, New York, NY, USA, 2010.

[8] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Comput.*, 33:648–662, November 2007.

[9] NVIDIA Corporation. *CUDA C Best Practices Guide 3.2*, 2010.

[10] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide 3.2*, Nov. 2010.

[11] H. Rhodin. LLVM PTX Backend. http://sourceforge.net/projects/llvmptxbackend.

[12] The Portland Group. PGI Fortran & C Accelator Programming Model. White Paper, 2010.

[13] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu. Languages and compilers for parallel computing. chapter CUDA-Lite: Reducing GPU Programming Complexity, pages 1–15. Springer-Verlag, Berlin, Heidelberg, 2008.