Dynamic Instrumentation and Optimization for GPU Applications

Naila Farooqui
Georgia Institute of Technology

Christopher J. Rossbach
Microsoft Research

Yuan Yu
Microsoft Research

ABSTRACT

Parallel architectures like GPUs are a tantalizing compute fabric for performance-hungry developers. While GPUs enable order-of-magnitude performance increases in many data-parallel application domains, writing efficient codes that can actually manifest those increases is a non-trivial endeavor, typically requiring developers to exercise specialized architectural features exposed directly in the programming model. Achieving good performance on GPUs involves effort-intensive tuning, typically requiring the programmer to manually evaluate multiple code versions in search of an optimal combination of problem decomposition with architecture- and runtime-specific parameters. For developers struggling to apply GPUs to more general-purpose computing problems, the introduction of irregular data structures and access patterns serves only to exacerbate these challenges, and only increases the level of effort required.

This paper proposes to automate much of this effort using dynamic instrumentation to inform dynamic, profile-driven optimizations. In this vision, the programmer expresses the application using higher-level front-end programming abstractions such as Dandelion [13], allowing the system, rather than the programmer, to explore the implementation and optimization space. We argue that such a system is both feasible and urgently needed. We present the design for such a framework, called Leo. For a range of benchmarks, we demonstrate that a system implementing our design can achieve from 1.12 to 27x speedup in kernel runtimes, which translates to 9-40% improvement for end-to-end performance.

1. INTRODUCTION

Parallel hardware, such as general-purpose GPUs, can enable high throughput in a variety of application domains, including data-intensive scientific applications, physical simulations, financial applications, and more recently, big-data applications [18, 13]. Evidence that GPUs can improve performance over traditional CPU implementations in these domains is abundant, but manifesting such improvements for individual applications remains effort intensive, and generally requires considerable programmer expertise.

Programmer-facing architectural features are a hallmark of GPU programming. Front end GPU programming frameworks support language-level abstractions to manipulate and manage specialized memories, caches, and thread geometries because exploiting the underlying architectural features is almost always required for best-case performance, and because tools that can effectively automate their use remain elusive. Consequently, optimizing GPU workloads typically requires the programmer to implement and compare multiple code versions that exercise different combinations of those features.

GPU hardware is designed to take advantage of regularity in workloads that feature minimal synchronization, high arithmetic intensity, and predictable memory access patterns. Unfortunately, the wide availability of GPUs continues to entice programmers to apply them in data-parallel application domains where such pronounced regularity is not the common case, such as graph traversal, data mining, and scientific simulations. While GPU acceleration can be performance profitable for such irregular data parallel workloads [4, 15], they feature data-dependent control flow and memory access patterns that are difficult to predict statically, which translates to a significant cost in additional programmer effort. In particular, the efficacy of code-transforming optimizations is highly data dependent for irregular codes, effectively increasing the dimensionality of the space the programmer must search for optimal combinations. We argue that the current level of manual optimization effort is untenable. We need better, automated approaches to GPU optimization.

The recent emergence of higher-level programming front-ends for GPUs such as Dandelion [13], Copperhead [5], DSLs coded to Delite [3], and others [2, 6, 11, 9] represent an additional challenge, as well as an opportunity. Such frameworks are attractive for the degree to which they insulate the programmer from low level architectural details, yet the extent to which they can reliably and predictably exploit those features in service of performance is often limited. However, because these frameworks generate or cross-compile code to produce GPU implementations, they provide a natural interface at which a compiler and runtime can collaborate to instrument, measure, and improve generated implementations, automatically exercising code transformations commonly used in GPU optimization efforts.

In this paper, we describe the design of Leo, a profile-driven dynamic optimization framework for GPU applications. Motivated by an emerging abundance of unstructured GPU applications that exhibit highly data-dependent memory and control-flow patterns that cannot be determined statically, Leo dynamically profiles the behavior of GPU applications using binary instrumentation, and uses the runtime characteristics of the applications to drive GPU-specific code optimizations such as memory layout transformations. The class of applications Leo targets, therefore, are streaming workloads that iteratively perform the same computations on large amounts of data, a model suitable for today's data-parallel architectures. In particular, Leo employs iterative information flow analysis and data structure transformations to improve the memory behavior of such applications. It measures an application's runtime behavior and selectively applies optimizations during the execution of the application. Leo achieves this by integrating two existing systems: Dandelion [13] and GPU Lynx [8]. Dandelion provides the compiler framework for code transformations, and GPU Lynx provides the dynamic instrumentation framework to identify optimiza-

tion strategies.

The primary contributions of this paper, therefore, are:

- The preliminary design and implemenation of Leo, a dynamic instrumentation and optimization framework that automatically explores code-transformation optimizations for GPUs.
- Experimental results from our prototype demonstrating the necessity, feasibility, and potential performance-profitability of such systems.

2. BACKGROUND AND MOTIVATION

2.1 GPU Computing

This paper uses NVIDIA GPU devices and CUDA as the target platform, and so we describe the GPU execution model in that context. Note, however, that the same concept and technology can be applied to OpenCL and its supported devices. A CUDA program is composed of a series of multi-threaded *kernels*. Computations are performed by a tiered hierarchy of threads. At the lowest level, collections of threads are mapped to a single stream multiprocessor (SM) and executed concurrently. Each SM includes an L1 data cache, a shared scratch-pad memory for exchanging data between threads, and a SIMD array of functional units. This collection of threads is known as a cooperative thread array (CTA), and kernels are typically launched with tens or hundreds of CTAs which are oversubscribed to the set of available SMs. A work scheduler on the GPU maps CTAs onto individual SMs for execution, and the programming model forbids global synchronization between SMs except on kernel boundaries.

Global memory is used to buffer data between CUDA kernels as well as to communicate between the CPU and GPU. CTAs execute in SIMD chunks called *warps*; hardware warp and thread scheduling hide memory and pipeline latencies. Effective utilization of the memory subsystem is also critical to achieving good performance.

2.2 GPU Metric: Memory Efficiency

Memory efficiency is a warp-level metric that characterizes the spatial locality of memory operations to global memory. Global memory is the largest block of memory in the GPU memory hierarchy and also has the highest latency. To alleviate this latency cost, the GPU memory model enables coalescing of global memory accesses for threads of a half-warp into one or two transactions, depending on the width of the address bus. However, scatter operations, in which threads in a half-warp access memory that is not sequentially aligned, result in a separate transaction for each element requested, greatly reducing memory bandwidth utilization. Memory efficiency is defined as the ratio of dynamic warps executing each global memory dynamic instruction to the number of memory transactions needed to complete these instructions. If all the memory operations are sequentially aligned, we will have 100% memory efficiency.

2.3 A Motivating Example

In this section, we motivate the need for a profile-driven dynamic optimization framework using a concrete application, *SkyServer*. The SkyServer application takes in large collections of astronomical, digital data in the form of photo objects and neighbors, and filters them to find related objects. The SkyServer workload is, in essence a series of relational equi-join operations and filtering over the two collections (see Figure 2). Differing input data distributions can yield very different selectivity for the join predicates, which in turn has a profound impact on dynamic memory access and control flow patterns in the GPU code implementing the join. We use two distinct set of inputs to demonstrate the challenges arising from the irregular memory access patterns exhibited by this application, and discuss the solutions to this problem.

The input sets (1 and 2, detailed in Section 4) both work on the same total number of photo objects and neighbors, but the data distribution in set 1 yields very low selectivity for the join predicate: very few photo objects actually match neighbor objects. For set 2, the majority of the neighbor objects match the join predicate. Both photo objects and neighbor objects are defined as structures with multiple fields. A simple layout of these objects (direct mapping) generates an Array-of-Structures (AoS) data layout in GPU memory. Since each GPU hardware thread works on an individual object, the AoS layout prevents coalesced reads and writes as the members of the data structure are placed contiguously in memory, forcing different threads to access scattered memory locations. A well-known optimization to improve memory efficiency is to transform the AoS layout to a Structure-of-Arrays (SoA) layout. This results in a sequential access pattern for all threads in the same warp, improving memory efficiency. In general, the AoS-to-SoA transformation achieves significant improvements in performance on the GPU due to better utilization of the global memory bandwidth.

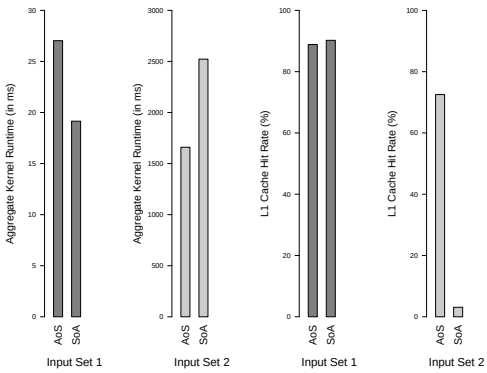

Figure 1: SkyServer runtimes and cache hit rates: (a) runtime on set 1, (b) runtime on set 2, (c) cache hit rates for set 1, and (d) cache hit rates for set 2.

This AoS-to-SoA optimization on the SkyServer application increases the memory efficiency by a factor of two for most of its GPU kernels. However, the SoA version does not always improve the overall performance. As shown in Figure 1, while the SoA version improves the performance for the first input set, when very few objects match the join predicate, it has a negative performance impact on the second input set, when there are a large number of matches. Effective optimization for this workload needs to take into account dynamic information to deal with input-dependent performance.

The negative correlation between memory efficiency and performance for set 2 demonstrates the complexity of the GPU global memory/cache model. An SoA transformation moves members of a given object farther apart, by a factor of the array size. So when members of an object are likely to be accessed sequentially, it can lead to very high L1 cache misses when the array is large (as is the case for input set 2). Therefore, although the SoA optimization results in better global memory bandwidth utilization on the GPU, it results in poor spatial locality for members of the same object. For set 1, L1 hit rates are unaffected by the optimization because low selectivity of the join enables most intermediate data to fit in cache, with the side-effect that failure to coalesce memory transactions has no real performance impact. However, for

set 2, the L1 cache hit rate declines from 72%, for the AoS version, to only 3%, for the SoA version.

The example shows that the memory efficiency optimization does not always correlate positively with runtime performance, and its overall benefits depend on the complex interactions of GPU memory hierarchy induced by the inputs. It highlights the need for a dynamic optimization framework that not only measures an application's memory efficiency at runtime, but also evaluates the impact of a particular code transformation and makes the optimal decision at runtime. Our proposed framework, Leo, addresses precisely this need.

3. DESIGN

In this section, we present the design and our preliminary implementation of Leo. Although we envision such an auto-optimizing engine to be a part of any GPU high-level runtime infrastructure, the current design of Leo is achieved by the integration of the GPU Lynx dynamic instrumentation library into the Dandelion compiler/runtime infrastructure.

3.1 System Components

As a dynamic optimization framework, Leo orchestrates the identification and selection of the optimal code and data layout transformation during the application's execution. It consists of the following two main components:

- A compilation engine that generates GPU kernel code and data layout on-the-fly from higher-level language source code.
- A JIT-based profiling engine that enables dynamic instrumentation and profiling of GPU code at runtime.

This section gives a high-level overview of these components.

3.1.1 Code Generation Framework

Leo leverages Dandelion to run LINQ applications on GPU. Leo needs to perform code and data layout transformations that are not supported by Dandelion, and we are in the process of adding the support into Dandelion.

The Dandelion system enables the execution of Language-Integrated Query (LINQ) on GPUs. LINQ introduces a set of declarative operators, which perform transformations on .NET data collections. LINQ applications are computations formed by composing these operators. Most LINQ operators are common relational algebra operators, including projection (Select), filters (Where), grouping (GroupBy), aggregation (Aggregate) and join (Join). The Dandelion compiler automatically compiles a LINQ query into a data-flow graph and any user-defined .NET code into GPU kernels. The Dandelion runtime automatically manages the execution of the data-flow graph on GPUs and the data transfer between CPU and GPU. For example, the SkyServer application is essentially a Join followed by a filtering. Figure 2 shows the data-flow graph generated by the Dandelion compiler. The nodes of the graph represent GPU kernels that are cross-compiled from their .NET functions.

3.1.2 Instrumentation Engine

We use GPU Lynx for dynamic profiling of GPU code. We improved Lynx significantly with a static information flow analysis so that it could be used to identify the candidate data structures for optimization.

Lynx allows the creation of customized, user-defined instrumentation routines that can be applied transparently at runtime for a variety of purposes, including performance debugging, correctness checking, and profile-driven optimizations. When built as a library, Lynx can be linked with any runtime. In the Leo framework, Lynx is integrated with the Dandelion compiler/runtime to support the execution of CUDA kernels representing the LINQ relational algebra primitives, on NVIDIA GPU devices. Lynx provides a parser and immediate representation (IR) abstraction for extracting and generating NVIDIA's parallel thread execution (PTX) from the compiled CUDA fat binary. An essential feature of GPU Lynx is its flexibility and extensibility, enabling both the specification of user-defined instrumentations using its C-based API, and the creation of sophisticated control-flow and data-flow analyses from its IR abstraction.

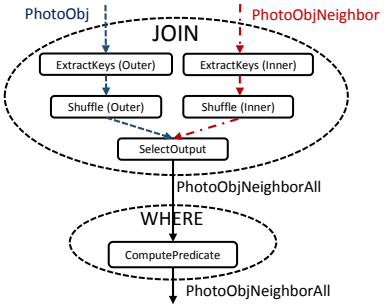

Figure 2: Simplified data-flow graph for SkyServer

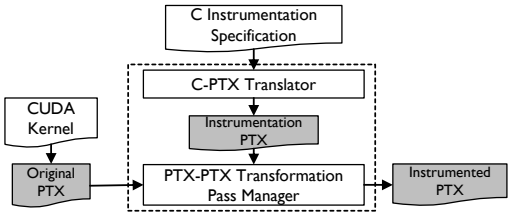

Figure 3: GPU Lynx Instrumentation Engine

An overview of the GPU Lynx instrumentation engine is shown in Figure 3. A C-based instrumentation is provided to the framework in addition to the original GPU kernel. The specification defines where and what to instrument. Lynx allows instrumentations to be defined at the kernel level, basic block level, or the instruction level. The Lynx engine generates the final instrumented PTX kernel from the C specification and the original PTX kernel, by enlisting the C-PTX Translator and the PTX-PTX Transformation Pass Manager.

3.2 System Overview

The Leo runtime orchestrates the identification and selection of the optimal code transformations and data layouts for GPU kernels. The computation model we support is based on streaming, i.e., the input is divided into chunks and chunks are transferred to GPU concurrently with the GPU execution. This model enables Leo to make optimization decisions based on the execution of preceding chunks. In the current design, Leo runs the Lynx instrumented code for the first chunk to determine possible candidate kernels for optimization. This allows Leo to generate the optimized version of the code with the necessary code and data layout transformations. We then run the second and third chunks with and without the optimizations respectively, and compare the total elapsed running times to determine which version of the code to use for the subsequent chunks.

Figure 4 presents a high-level overview of the design of the Leo framework, highlighting the steps the runtime takes in order to apply profile-driven optimizations to LINQ applications. In Step 1, we use Dandelion to generate the Dandelion

version of the GPU code. In Step 2, we apply Lynx to generate an instrumented version of the GPU code. In Steps 3 and 4, the instrumented code is executed and profiling information is used to identify the candidate data structures for optimization. In Step 5, we apply the code and data layout transformations to the candidate data structures to generate an optimized version of the GPU code, which could be used to execute subsequent data chunks to achieve better performance.

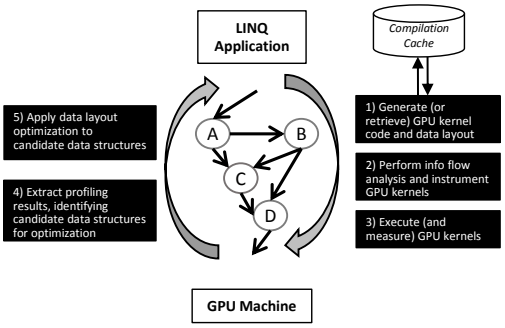

Figure 4: High-level overview of Leo.

4. PRELIMINARY RESULTS

In this section, we evaluate our techniques on two applications, K-Means and SkyServer. For each application, we study the performance impact on the set of most compute-intensive kernels. Our experimental environment is detailed in Table 1; applications and inputs are listed in Table 2.

CPU	Intel Xeon E5504 @ 2.00 GHz
GPU	Tesla M2075, 448 CUDA cores
Operating System	Windows Server 2008 (SP1)
CUDA Version	5.5

Table 1: System Configuration

4.1 K-Means

K-Means is a classical clustering algorithm which partitions M N-dimensional points (or vectors) into k clusters by repeatedly mapping each point to its nearest center, and then recomputing the cluster centers by averaging the points mapped to each center. The cross-compiled Dandelion code will rely on a number of lower-level primitives to implement the relational algebra, but we focus on the corresponding `NearestCenter` GPU kernel as its execution overwhelmingly dominates end-to-end performance for the workload.

The primary data structure is a collection of N-dimensional points, whose most obvious in-memory representation arranges the dimensions of each point in contiguous memory locations. If the number of dimensions is large, such a layout yields poor memory efficiency on the GPU. Figure 5(a) shows the memory efficiency for the original and the AoS-SoA transformed versions for this code, with varying dimensions, and Figure 5(b) shows the corresponding kernel speedup. With $N = 1$, the layouts and memory efficiency are predictably equivalent. As N increases, the original version's memory efficiency degrades, while the optimized version's memory efficiency remains high.[1] Memory efficiency has a first-order impact on performance for this workload as N increases, providing $27\times$ improvement at $N = 32$.

[1]The K-Means `NearestCenter` kernel takes two vectors as input: one for the points and one for the cluster centers. The number of centers is generally much smaller than the number of points, and fits in L1 cache for all input sizes we consider, so our optimization focuses on the points collection.

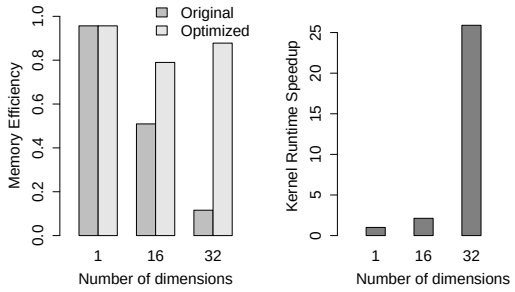

Figure 5: K-Means (a) memory efficiency and (b) kernel runtime speedup for varying dimensions. Memory efficiency has a direct correlation on kernel runtime performance for this workload.

4.2 SkyServer

SkyServer takes as input collections of digitized astrological images (encoded as "photo objects") and the relative locations of images (encoded as "photo neighbors"). The workload filters these data according to criterion that enables the identification of related astrological objects. It is expressed in LINQ as a series of `Join` operations over the objects and neighbors collections.

Dandelion implements the underlying relational algebra on GPUs using techniques fundamentally similar to hash-join [7], decomposed into a number of GPU kernels. The approach first identifies items in the input relations matching the join predicate, then shuffles matching items per-relation into contiguous positions, then computing the final join output as the cross-product of items in each (matching) contiguous block. While a deep understanding of the implementation is not required to here (we refer the interested reader to [13]), some details play an important role in the profitability of the optimizations performed by Leo: to first order, four kernels corresponding to the steps above dominate the performance of SkyServer end-to-end performance, called `ExtractKeys`, `Shuffle`, `SelectOutput`, and `ComputePredicate`, so our evaluation effort focuses on Leo's ability to reduce compute latency for these four kernels.

We evaluate SkyServer with two input sets (1 and 2), corresponding to different levels of selectivity for the join predicate. In set 1, very few of the neighbors (up to 200) match the predicate against photo objects, and vice versa for set 2, where almost all of the photo neighbors are matches. Consequently, for set 1, the `SelectOutput` kernel is the least significant contributor to the overall computation, but forms the largest component for set 2.

Figure 6 presents (a) memory efficiencies of the original and optimized kernels, (b) individual kernel speedups achieved by the AoS-SoA optimization, and (c) the computation breakdown of the kernels for the two input sets. The transformation improves memory efficiency for all of SkyServer's kernels, but the improvement is not as significant for `SelectOutput`. Although the transformation ensures that the loading of members of the SkyServer's input data structures are coalesced, sequentially indexed threads may not necessarily be accessing contiguously located elements in the array due to the data skew introduced by the hash function and the join predicate.

`ExtractKeys` and `ComputePredicate` benefit from the AoS-

Application	Description	Input Configuraions
K-Means	M N-dim points, k clusters	M=1x10^6, N=32, k=40
SkyServer	O objects, N neighbors	Input 1: O=2048, N=2x10^7, up to 200 matching neighbors Input 2: O=2048, N=2x10^7, up to 2x10^7 matching neighbors

Table 2: Applications

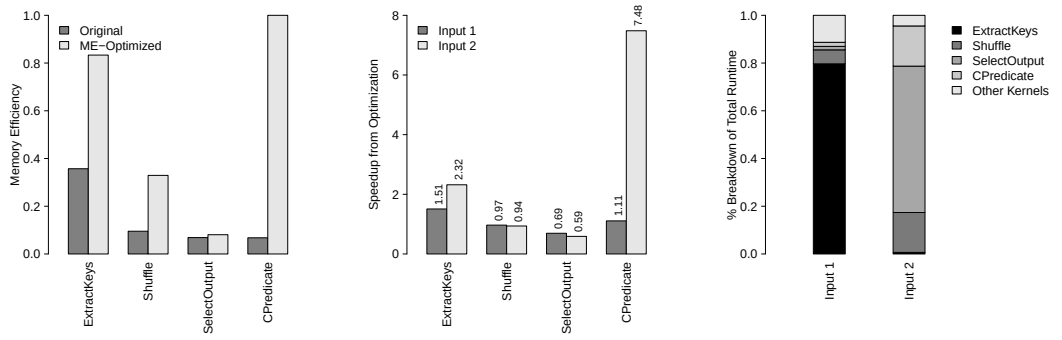

Figure 6: SkyServer (a) memory efficiency of original and optimized kernels, (b) individual kernel runtime speedups and (c) computation breakdown of all the kernels for the two distinct input sets.

SoA transformation for both input sets, whereas `Shuffle` and `SelectOutput` are negatively impacted in both cases. This is due to the tension between memory coalescing and cache performance discussed in Section 2.3. The data structures used in SkyServer comprise 10 long integer and floating point fields, so high selectivity of the join predicate (many matches) results in high L1 miss rates under the SoA layout for `Shuffle` and `SelectOutput` as they must collect data spread across many cache lines to shuffle individual records into a logically contiguous arrangement. When the join predicate has low selectivity, `Shuffle` and `SelectOutput` constitute only about 7% of the entire computation, so the negative impact of the transformation on those kernels is masked by the benefit enjoyed by `ExtractKeys` and `ComputePredicate`. We conclude that the profitability of this optimization is input-dependent for SkyServer, highlighting the need for a dynamic framework to select the best code transformations at runtime.

4.3 End-to-End Performance

In this section, we consider the end-to-end impact of memory efficiency optimizations on the performance We run eight iterations of each workload to allow the compiler and runtime to iteratively improve the generated code. In the first iteration, we instrument GPU code and measure its memory efficiency, using that to guide the decision about whether to generate a different optimized version. The second and third iterations are used to measure performance for optimized and unoptimized versions (without instrumentation), enabling the runtime to select the better of the two, which is used for the remainder of the iterations.

Table 3 compares the speedup over unoptimized code of profile-guided optimization against the speedup that would be attained if a perfect oracle in the compiler could select the most performant code version with no overhead. The *Profile-Guided* column presents the speedup that we observed with our framework, which must amortize overheads of instrumentation, measurement, optimization application, and reloading the GPU device when code versions are changed. The data assume that the compilation cache hides the overheads of JIT compilation: compile overheads are elided. The *Oracle* column shows the speedup attained when the optimal version is selected with no overhead.

For K-Means, the potential benefit is modest (9%), and

Application	Profile-Guided	Oracle
K-Means	**1.09**	1.10
SkyServer Input 1	**1.34**	1.52
SkyServer Input 2	**1.42**	1.53

Table 3: End-to-End Performance for Profile-Guided optimizations of K-Means and SkyServer. The *Profile-Guided* column shows the speedup of many iterations of the optimized code over unoptimized code including overheads to instrument, measure and regenerate code, while the *Oracle* column shows the speedup over unoptimized code that would be attained if a perfect oracle selects the optimal code version.

the profile-guided runs achieve speedups very close to those obtained with an oracle. In contrast, SkyServer sees significant potential benefit. SkyServer has complex data structures and complex generated code, relative to the other workloads. As a result, instrumentation overheads are more significant as well. The difference in maximum profitability of optimization across these workloads is expected. K-Means is compute-bound, so its lower memory intensity translates to fewer sites at which instrumentation code is inserted, yielding lower runtime overhead relative to SkyServer.

The data also show that additional overheads in the end-to-end scenario (notably CPU-GPU data transfers) attenuate the speedups observed in when GPU kernels are measured in isolation. Emerging integrated CPU-GPU architectures and tools that maximize asynchrony and/or eliminate unnecessary data movement can lessen this impact, enabling our framework to deliver higher gains in future systems.

5. RELATED WORK

Profile-Guided Optimization. Adaptive, dynamic, and profile-guided optimization techniques have enjoyed much research attention over the past few decades [1]. Leo draws from basic techniques described in the literature, and we claim no contribution in this domain other than synthesizing known techniques in a new context.

Higher-Level Language Front-ends. Much research has been devoted to higher-level programming front-ends for GPUs [13, 5, 3], with the goal of insulating the programmer from complexities induced by the architecture. The Delite compiler and runtime framework [3] performs domain-specific optimizations on DSL applications for execution on multiple heterogeneous backends. Delite shares many common features with our framework: increasing programmer productivity without sacrificing performance, graph-based representation of computation to enable runtime scheduling and optimizations, and heterogeneous code generation for both CPUs and GPUs. Delite's extensibility enables compiler optimizations that are aware of the semantics of operations within the domain, while Leo searches for low-level input-dependent optimization opportunities that are inaccessible to an optimizer with a static view, however semantically rich that view may be. Copperhead [5] is a high-level data parallel language embedded in Python. Leo's novelty is integration of a dynamic instrumentation engine, GPU Lynx, with a cross-compilation runtime, Dandelion, to enable transparent profile-driven optimizations.

GPU Optimizations. GPU-specific optimizations, such as the AoS-SoA transformation, have been studied extensively in previous works as well [12, 16, 19, 17]. In [12], Rompf et al. show how the AoS-SoA data structure optimization can be performed via internal compiler passes. G-Streamline [19] removes dynamic irregularities in GPU applications on-the-fly, such as those resulting from irregular memory accesses and data-dependent control. Leo's goal is to automatically determine when a particular optimization is useful, and respond to the application behaviors dynamically. As such, libraries such as DL and G-Streamline are complementary to our work, and can be linked with our framework to provide the data layout re-ordering mechanisms for optimizing irregular memory and control-flow accesses.

Irregular Workloads. Burtscher et al. [4] observe the memory efficiency-cache tension that we discussed in our workloads as well. In general, GPU acceleration for irregular data parallel workloads [4, 10, 14, 15], has been studied extensively, and augments the potential value of our proposed framework for current and future heterogeneous systems.

6. CONCLUSION

While higher-level front-end languages for GPUs insulate the programmer from complexity and low-level architectural detail, but will only become widely used if compilers and runtimes are able to effectively take advantage of low-level architectural features and (potentially input-dependent) optimizations that are currently the programmer's responsibility. Leo is a dynamic optimization framework that automatically searches the implementation and optimization space formerly searched by hand. The current implementation focuses on the memory efficiency optimization. Leo is a research prototype: the limited results presented here give reason to be optimistic that such a framework can be realized.

7. REFERENCES

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proceedings of the IEEE, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation*, 2005.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *SC 2012*.

[3] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, and K. OLUKOTUN. A heterogeneous parallel framework for domain-specific languages. In *PACT 2011*.

[4] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. IISWC 2012.

[5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. PPoPP 2011.

[6] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. SC 2011.

[7] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. SIGMOD 1984.

[8] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures. In *ISPASS 2012*. `http://code.google.com/p/gpulynx/`.

[9] M. D. Linderman, J. D. Collins, H. W. 0003, and T. H. Y. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS 2008*.

[10] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. PPoPP 2012.

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI 2013*.

[12] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. POPL 2013.

[13] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. SOSP 2013.

[14] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: integrating file systems with gpus. ASPLOS 2013.

[15] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *ICML 2011*.

[16] I. Sung, G. Liu, and W. Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, page 1âĂŞ11. IEEE, 2012.

[17] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. PPoPP '13.

[18] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. MICRO-45, 2012.

[19] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS 2011*.