

Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications

Naila Farooqui
Georgia Institute of Technology
naila@cc.gatech.edu

Christopher J. Rossbach
Microsoft Research
crossbac@microsoft.com

Yuan Yu
Microsoft Research
yuanbyu@microsoft.com

Karsten Schwan
Georgia Institute of Technology
schwan@cc.gatech.edu

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 [18], 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 7-40% 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 [1], physical simulations [13], financial appli-

cations [15], and more recently, big-data applications [23, 18]. 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.

We argue that the current level of manual optimization effort is untenable if parallel architectures are to become more broadly applicable. GPUs have become ubiquitous in modern computing environments, resulting in more demand for generality. Performance-hungry programmers use GPUs in increasingly complex applications where algorithms rely fundamentally on unstructured or irregular control and data access patterns. GPU hardware is designed explicitly to take advantage of the regularity characterized by workloads with minimal synchronization, high arithmetic intensity, and predictable memory access patterns. However, such pronounced regularity is not the common case for many highly data-parallel applications: graph traversal, data mining, and scientific simulations, for example, feature abundant parallelism while exhibiting data-dependent control flow and memory access patterns that are difficult to predict statically. GPU acceleration can be performance profitable for irregular data parallel workloads [5, 12, 19, 20], but typically at a significant cost in additional programmer effort. Moreover, the efficacy of code-transforming optimizations is highly data dependent for irregular codes:

the need for better, automated approaches to GPU optimization is pronounced.

The recent emergence of higher-level programming front-ends for GPUs such as Dandelion [18], Copperhead [6], DSLs coded to Delite [4], and others [3, 7, 16, 11] 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 [18] and GPU Lynx [9]. Dandelion provides the compiler framework for code transformations, and GPU Lynx provides the dynamic instrumentation framework to identify optimization strategies.

The primary contributions of this paper, therefore, are:

- An in-depth study of the memory efficiency optimization and its runtime implications on applications.
- The design and preliminary implementation of a dynamic instrumentation and optimization framework that automatically explores code-transformation optimizations for GPUs.
- Experimental results that demonstrate the necessity, feasibility, and potential performance-profitability of such systems.

The remainder of this paper is organized as follows. Section 2 provides the necessary background and motivates with a real workload. Section 3 describes the design and implementation of Leo. In Section 4, we evaluate the effectiveness of Leo using several applications. Section 5 discusses related work and Section 6 concludes.

2 Background and Motivation

2.1 GPU Computing

Although this paper uses NVIDIA GPU devices and CUDA as the target platform, the same concept and technology can be applied to OpenCL supported devices. A CUDA program is composed of a series of multi-threaded *kernels*, which can be thought of as computational functions that are executed on the GPU. 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*. A *warp* is the most basic unit of scheduling on the GPU, which processes a single instruction over all of the threads in it at the same time, in lock-step fashion. Hardware warp and thread scheduling hide memory and pipeline latencies.

2.2 GPU Metrics

The two most well-known performance limiters on the GPU are memory bandwidth utilization and thread divergence, captured via *memory efficiency* and *activity factor* [10]. These two metrics, therefore, drive the dynamic optimizations in Leo.

2.2.1 Memory Efficiency

Memory efficiency is a warp-level metric that characterizes the spatial locality of memory operations to global memory, the block with the highest latency in the GPU memory hierarchy. 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. Effective utilization of the GPU memory subsystem is, therefore critical to achieving good performance.

2.2.2 Activity Factor

Thread divergence is a common performance issue with GPU code. When threads within a warp diverge, taking different control paths, the warp serially executes each branch path taken, disabling threads that are not on that path, so non-uniform control flow entails a significant performance penalty. Activity factor (AF) characterizes how well an application utilizes the GPU SIMD parallel execution model, effectively by measuring thread divergence. Applications with completely uniform control-flow, or no thread divergence, exhibit a 100% activity factor. In contrast, applications with low AF exhibit a higher degree of control-flow irregularity.

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 sets 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.

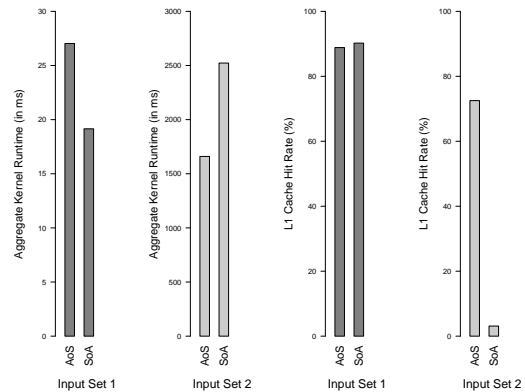


Figure 1: SkyServer runtimes and cache hit rates: (a) runtime on set 1, (b) runtime on set 2, (c) cache hit rates for set 1, and (d) cache hit rates for set 2.

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

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.

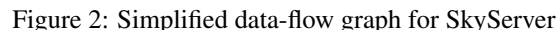
In this section, we present the design and our preliminary implementation of Leo. Although we envision such an auto-optimizing engine to be a part of any GPU high-level runtime infrastructure, the current design of Leo is achieved by the integration of the GPU Lynx dynamic instrumentation library into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

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:

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

Leo leverages Dandelion to run LINQ applications on GPU. We extended Dandelion to add the necessary support to perform code and data layout transformations required by Leo. See Section 3.3.1 for more detail.

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.



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. See Section 3.3.2 for more detail.

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

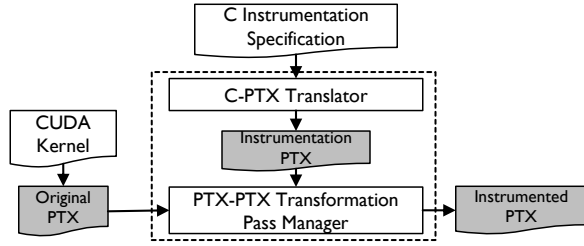


Figure 3: GPU Lynx Instrumentation Engine

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. This profiling is repeated at continuous intervals to detect time-varying runtime behaviors and relevant application phase changes.

Figure 4 presents a high-level overview of the design of the Leo framework, depicting the general steps the runtime takes in order to apply profile-driven optimizations to LINQ applications. We describe these steps more concretely in the context of the *SkyServer* application introduced in Section 2.3.

In Step 1, we use Dandelion to generate the original version of the *SkyServer* GPU code, which results in an AoS data layout in memory. In Step 2, we apply Lynx to generate an instrumented version of the GPU code. For the data layout transformation, the code is instrumented with the memory efficiency metric to characterize the spatial locality of global memory accesses. In Steps 3 and 4, the instrumented code is executed and the profiling information is collected. The original GPU kernels in the *SkyServer* application exhibit a low average memory efficiency (less than 40%, as shown in Figure 10(a)). The instrumented results capture global memory load and store accesses for all possible data sources in the code. As

one might expect, not all data sources may necessarily exhibit poor memory efficiency. As a result, Leo applies information flow analysis to link each global memory load/store access in the GPU kernel code to its corresponding data source. This enables Leo to precisely identify the data structures that need to be transformed. The AoS-SoA code and data layout transformations are applied to the candidate data structures in Step 5 to generate an optimized version of the GPU code. However, as discussed in Section 2.3, the *SkyServer* workload exhibits input-dependent performance with the AoS-SoA optimization. The optimization is effective when the join predicate has low selectivity (input set 1) but degrades performance when the join predicate has high selectivity (input set 2) due to the L1 cache effects. To deal with this input-dependent behavior, Leo uses the execution of the second and third chunks, with and without the optimization, respectively, to select the optimal data layout for all subsequent chunks.

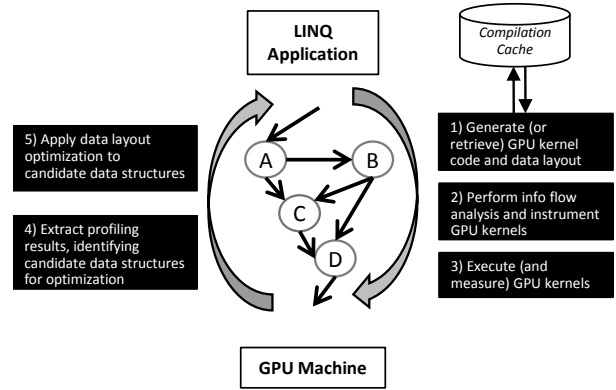


Figure 4: High-level overview of Leo.

3.3 Implementation Details

We now provide a more detailed description of the new and important features of Leo.

3.3.1 Code and Data Layout Transformation

As described before, the Dandelion compiler compiles a LINQ application to a data-flow graph, where the nodes represent fragments of the computation and the edges represent communication channels. For execution on the GPU, each node translates to a primitive relational algebra operation, such as a *Select*, *Join*, or *GroupBy*. The generated CUDA primitives are generic: the input and output data types, as well as the user-defined functions are template parameters of the primitives. The Dandelion compiler instantiates the primitives using the generated GPU data-types and code.

```

Compute Function (Original/AoS)
__device__ Compute(PhotoObj& p,
PhotoObjNeighbor& n){
    struct PhotoObjNeighborAll pn;
    pn.p= p;
    pn.n= n;
    return pn;
}

Compute Function (Optimized/SoA)
__device__ Compute(PhotoObj*p,
PhotoObjNeighbor*n, int pLen, int pIdx,
int nLen, int nIdx){
    struct PhotoObjNeighborAll pn;
    int s = 0;
    pn.p.a= *((long*)
    (p+s*pLen+sizeof(long)*pIdx);
    s += sizeof(long);
    pn.p.b= *((int*)
    (p+s*pLen+sizeof(int)*pIdx);
    s += sizeof(int);
    pn.p.c= *((int*)
    (p+s*pLen*sizeof(int)*pIdx);
    //...
    s = 0;
    pn.n.a= *((long*)
    (n+s*nLen+sizeof(long)*nIdx);
    s += sizeof(long);
    pn.n.b= *((long*)
    (n+s*nLen+sizeof(long)*nIdx);
    //...
    return pn;
}

```

Figure 5: Original (AoS) and optimized (SoA) code versions for an example function in SkyServer.

In addition to the code generation, Dandelion also handles the runtime management of data buffers across input and output channels, seamlessly allocating buffers on-the-fly, without programmer intervention. The layout of a particular data structure, therefore, can be modified by the Dandelion compiler dynamically, across the various input/output channels. This implies that a given node is capable of receiving data in one form but the subsequent node can receive the same data in another form. All generated CUDA kernels are stored in the compilation cache to avoid runtime JIT compilation overheads for subsequent data chunks and application runs.

For the data structures identified for optimization, Leo performs an AoS-SoA data layout transformation on these candidates. The transformation involves modifications to both the generated CUDA code and the data layout. Leo invokes the Dandelion compiler to generate the SoA versions of the CUDA code and switches from the row-major to the column-major data layout for the identified data structures.

Figure 5 shows the original (AoS) and optimized

```

for basic block bb in reversed cfg do
    for instruction inst in reversed bb do
        if inst = global mem load or store then
            registerSet s = {address operand of inst}
            flowMap[inst] = s
        end
        else
            for [instruction m, registerSet s] in flowMap
                do
                    for register r in s do
                        if dst reg operand of inst = r then
                            if inst = param mem load then
                                flowMap[m] = flowMap[m] +
                                (src operand of inst) - r
                            end
                            else
                                flowMap[m] = flowMap[m] U
                                ({src reg operands of inst} -
                                r)
                            end
                        end
                    end
                end
            end
        end
    end
end

```

Figure 6: Information Flow Analysis Algorithm

(SoA) versions of an example *auto-generated* function, *Compute*, in the *SkyServer* application. The *Compute* function performs the necessary data accesses for the photo object and photo object neighbor structures, and is invoked by the *SelectOutput* kernel (shown in Figure 7). In the original *Compute* function, an AoS memory layout is assumed, allowing the code to directly access the individual members of the two structures. The transformed version of this function assumes an SoA memory layout, requiring additional parameters to index the individual members of the structures appropriately in column-major order. Since the *Compute* function is automatically generated at runtime by the Dandelion compiler, we have the flexibility to modify the function prototype as needed.

3.3.2 Information Flow Analysis and Dynamic Instrumentation

Leo uses a combination of information flow analysis and dynamic instrumentation to identify data structures that exhibit irregular global memory access patterns at runtime. The goal of information flow analysis is to link each individual global memory load or store instruction

in the GPU kernel code to its corresponding data source. Since not all data structures may need to be transformed, it is important to precisely know which data structures are candidates for optimization. This mechanism enables just that.

Leo uses a form of information flow analysis, known as taint analysis [14], to track causal dependencies between global memory operations, and kernel parameters that identify the data sources for those memory operations. In this analysis, all addressable variables in global memory instructions, and corresponding registers that participate in the calculation for those addressable variables, are marked as tainted. Using a backward data-flow analysis, each addressable variable’s data source is tracked back to its corresponding kernel parameter. In the CUDA programming model, all kernel parameters are stored in the GPU parameter memory. The parameter must be loaded into a register from the parameter memory before being used. This information is used to determine each global memory operation’s data source.

The information flow analysis algorithm is presented in Figure 6, and a visual depiction for an example GPU kernel, the `SelectOutput` kernel from the `SkyServer` application, is shown in Figure 7. Specifically, Figure 7 shows how global memory addressable variables `r41` and `r44` are linked back to their corresponding kernel input parameters, `arg0` and `arg4`. Note that `arg0` maps to the outer relation (`pOuterRelation`) and `arg4` maps to the inner relation (`pInnerRelation`) of the cross product join operation in the original `SelectOutput` CUDA kernel.

```

Memory Efficiency Metric
ulong threadId = blockDim();
ulong warpId = (blockId() *
    blockDim() + threadId) >> 5;
ON_INSTRUCTION:
MEM_READ:
MEM_WRITE:
GLOBAL: {
    sharedMem[threadId] =
        computeBaseAddress();
    if(leastActiveThreadInWarp()) {
        uint offset = (memOps() * warpId +
            memOpId())*2;
        globalMem[offset] +=
            uniqueElementCount(sharedMem);
        globalMem[offset+1] += 1;
    }
}

```

Figure 8: C specification for memory efficiency metric.

A mapping for each global memory operation to its corresponding input data source is constructed. Each global memory operation is identified by an index that determines its position in the static PTX kernel code. This index is returned by GPU Lynx’s instrumentation

API function, `memOpId()`. For the `SelectOutput` kernel code snippet in Figure 7, the mapping would look like the following:

```

memOpId: 0 --> ....
memOpId: 1 --> pOuterRelation
memOpId: 2 --> pInnerRelation

```

Once such a mapping is established, the kernel is instrumented with the memory efficiency metric. The memory efficiency instrumentation is defined as follows.

For every global load or store instruction, each thread within a thread block computes the base memory address and stores it in shared memory. For NVIDIA GPUs, a half-warp, or 16 threads, can coordinate global memory accesses into a single transaction. This implies that if the base address is the same for all threads belonging to a half-warp, then the memory accesses will be coalesced. A single active thread within a warp, the one with the smallest index, is selected to perform an online reduction of the base addresses written to the shared buffer. It maintains a count of unique base addresses within a half-warp, to determine the total number of memory transactions required for a particular memory operation. The number of transactions required for a particular memory operation by a given half-warp is stored in a global memory instrumentation buffer, indexed by the memory operation index. After the kernel completes execution, the memory operation index is used to determine the individual memory efficiency for each kernel parameter. The C-based instrumentation specification for the memory efficiency metric is shown in Figure 8.

3.3.3 Extraction of Profiling Data

In order to determine if a particular code transformation is beneficial, the kernel runtimes of the generated original and optimized CUDA kernels are measured via GPU Lynx’s kernel runtime instrumentation [9]. The kernel runtime instrumentation polls hardware counters on the GPU device, exposed as PTX instructions, which capture precise measurements of multiple events within the execution of a single kernel without including latencies of PCI bus, driver stack, and system memory.

GPU Lynx exposes an API to Dandelion for retrieving profiling results. The profiling results are stored in a multi-dimensional map (JSON), identified by each kernel’s name at the first level, and the profiling metric at the second level. The profiling results include the necessary meta-data to identify the candidate input data structures for optimization as well. After the execution of the instrumented kernel code, the profiling results of the run are retrieved by the Dandelion compiler. The Dandelion compiler uses this information to selectively apply code transformations to the candidate data structures.

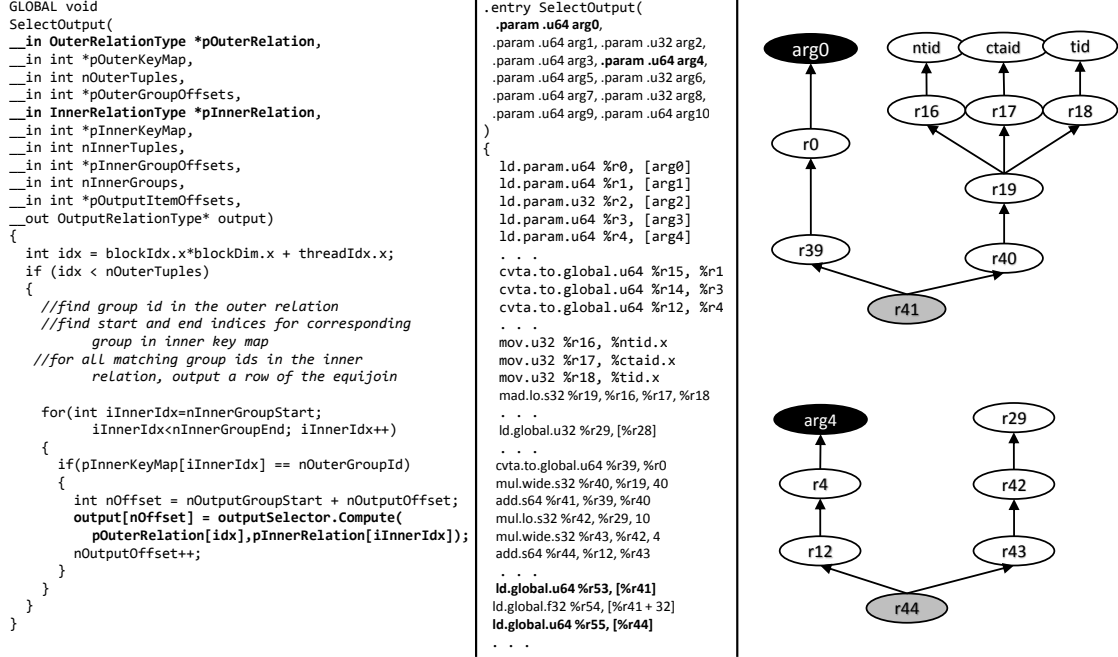


Figure 7: Information flow analysis: (a) CUDA code for the *SelectOutput* kernel, (b) PTX snippet of the same kernel code, and (c) visual depiction of the information flow analysis algorithm, showing the mapping of two global memory addressable variables to their respective data sources, identified as kernel input parameters.

The decision model to select appropriate data structure candidates for optimization can incorporate various pieces of information, such as a threshold on the memory efficiencies of the data structures, the size of the data structures, and so on. For instance, all data structures that have a memory efficiency lower than a specified value may be selected as candidates for optimization. In the current implementation, we try to aggressively optimize all data structures that have a memory efficiency lower than 100%, and input sizes greater than the L1 cache size.

4 Evaluation

We evaluate our design using a preliminary prototype of Leo using four applications from different domains: Black-Scholes, K-Means, SkyServer [24], and Bellman-Ford’s Single-Source-Shortest Path algorithm. The current Leo prototype is not a full end-to-end implementation: while it integrates dynamic instrumentation and measurement for activity factor and memory efficiency, along the necessary support at the generic primitive library targeted by the Dandelion compiler, manual intervention is still required to code alternate versions of anonymous functions in the cross compilation from C# to CUDA. While incomplete, we hope that our preliminary results offer useful insights on the potential profitability,

feasibility and necessity of a dynamically adaptive, auto-optimizing GPU framework.

All benchmarks are coded using .NET LINQ as described in Section 3.1.1. As a result, the cross-compiles GPU code for each benchmark relies on a number of separate kernels. We evaluate our profile-guided optimizations by considering their impact on the performance of the the most compute-intensive (and consequently, longest-running) of those kernels in isolation, and by considering their impact on end-to-end performance for a subset of the benchmarks: Black-Scholes, K-Means, and SkyServer. Details of the evaluation platform and benchmarks are provided in Tables 1 and 2 respectively.

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 BlackScholes

The Black-Scholes algorithm estimates option prices based on a number of parameters including constant price variation, strike price, time-to-expiry, etc: coded in LINQ, the workload is a single *Select* statement us-

Application	Description	Input Configurations
BlackScholes	N options	$N=1 \times 10^7$
K-Means	M N -dim points, k clusters	$M=1 \times 10^6$, $N=32$, $k=40$
SkyServer	O objects, N neighbors	Input 1: $O=2048$, $N=2 \times 10^7$, up to 200 matching neighbors Input 2: $O=2048$, $N=2 \times 10^7$, up to 2×10^7 matching neighbors
Single-Source-Shortest-Path	N nodes, M edges	USA road network: $N=24 \times 10^6$, $M=58 \times 10^6$

Table 2: Applications

ing a lambda function to encode the algorithm. Changing the BlackScholes data layout from an AoS to an SoA representation, we improve the memory efficiency from **18% to 100%**. The workload is largely compute-bound: a low ratio of memory accesses to compute operations limits the corresponding kernel speedup to just $1.12\times$.

4.2 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 workload is coded to LINQ as a GroupBy followed by a Select, the former of which uses a method called `NearestCenter` as the key extractor function. 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 9(a) shows the memory efficiency for the original and the AoS-SoA transformed versions for this code, with varying dimensions, and Figure 9(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 $N = 32$.

¹The 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.

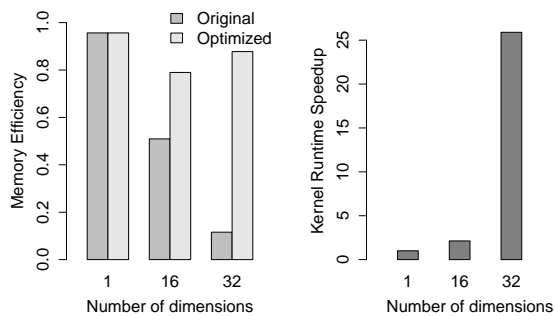


Figure 9: K-Means (a) memory efficiency and (b) kernel runtime speedup for varying dimensions. As the number of dimensions grows, the memory efficiency of the originalAoS code version degrades while the optimizedSoA version’s remains high. Memory efficiency has a direct correlation on kernel runtime performance for this workload.

4.3 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 [8], 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 [18]), 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 dom-

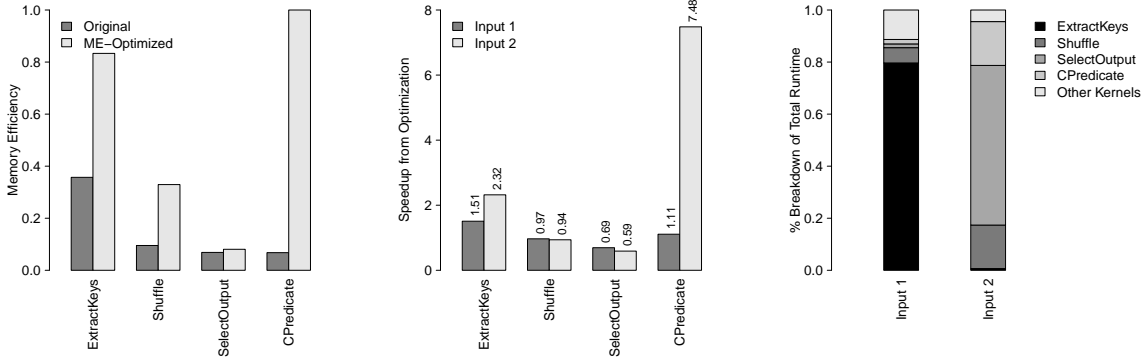


Figure 10: SkyServer (a) memory efficiency of original and optimized kernels, (b) individual kernel runtime speedups and (c) computation breakdown of all the kernels for the two distinct input sets.

inate 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 10 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-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 log-

ically 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.

4.3.1 Exploring Activity Factor

Recall from Section 2.2.2 that Activity factor (AF) essentially characterizes the level of thread divergence. In future work, we plan to incorporate AF to drive dynamic optimizations. As step towards that goal, we use SkyServer to demonstrate that AF can be a useful predictor for determining hash table sizes on-the-fly.²

Recall that for the *join* operation, keys are inserted into a lock-free hash table using a CAS operation. As a result, the distribution of the keys, the hash function, and the size of the bucket list array can yield variance in the length of the bucket lists, which in turn causes thread divergence as threads traverse different length lists concurrently. Smaller bucket list arrays exacerbate this effect by forcing longer bucket lists, while over-provisioning the bucket list array wastes memory, which is particularly scarce in big-data workloads such SkyServer. Dynamic resize is an unattractive option in this context primarily because it is an inherently sequential operation whose additional can be worse for performance than the thread divergence incurred by tolerating a poorly sized table. Consequently, hash-table bucket array size is a parameter managed by the Dandelion compiler. Because many applications have performance-sensitivity to the hash-table

²Many relational algebra primitives in Dandelion such as *Join* and *GroupBy* rely on efficient hash table performance.

size, Dandelion currently provides an interface for programmer hints for this parameter, but even when values are well-chosen the situation is not optimal, as the hash-table size remains constant even if the application’s needs change dynamically.

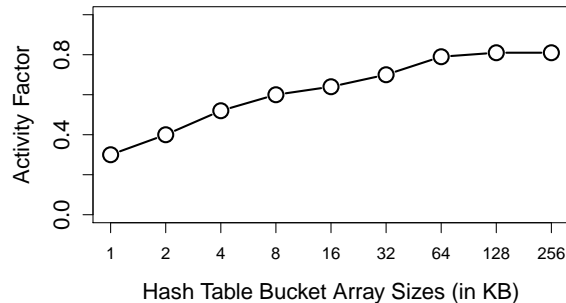


Figure 11: SkyServer configured with input set 2: activity factor for varying hash table bucket array sizes.

To demonstrate how activity factor can guide dynamic selection of the hash table size we measure the AF in SkyServer for various hash table sizes, shown in Figure 11. The data show that AF improves as the hash table size increases up to a 64KB, leveling out as sizes increase beyond that, suggesting an good choice of hash table size for the given data distribution. The same mechanisms used to communicate and respond to ME in the runtime can be used to select better hash table sizes based on AF.

4.4 Single-Source Shortest Path

Single-Source Shortest Path (SSSP), is Bellman-Ford’s algorithm for finding the shortest path from a source node to every other node in a graph. The algorithm is based on the principle of relaxation, in which iteratively improves an approximation until converging on an optimum. The data structures in our implementation are nodes (an id and distance) and edges (source, destination, weight).

Figure 12 shows the kernel runtimes and L1 hit rates for five iterations of this algorithm on the USA road network graph, for both the original and memory-efficiency optimized code versions. In aggregate, the AoS-SoA transformation improves memory efficiency from **71%** to **100%**, but the data show significant time-varying behavior. In early iterations, the SoA layout version performs better because the join predicate has low selectivity when only a few nodes have been visited. Over time more and more nodes are visited, increasing selectivity which results in the join spending more time shuffling objects in the correct regions. Since the SoA layout spreads individual object members across different cache lines, this in turn causes higher L1 miss rates, and

in later iterations, the original layout is more performant by larger and larger margins. This time-variance can be handled by our periodic re-evaluation scheme, which enables the runtime to detect and respond to such trends in application behavior.

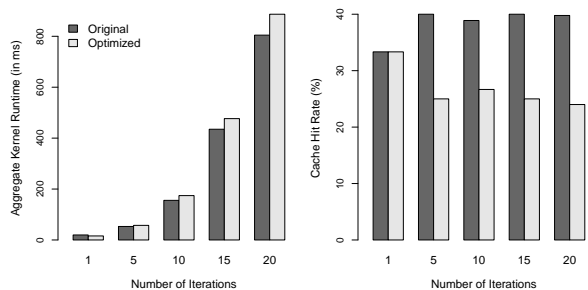


Figure 12: Single-Source-Shortest-Path (a) aggregate kernel runtimes and (b) cache hit rates for five distinct iterations on the same input (USA road network graph)

4.5 End-to-End Performance

In this section, we consider the end-to-end impact of memory efficiency optimizations on the performance of a subset of benchmarks: BlackScholes, K-Means, and SkyServer.³ The goal of this experiment is to demonstrate that Leo is able to amortize the runtime overheads incurred via dynamic instrumentation over a relatively small number of iterations and still provide significant performance gains in some cases. 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 code version represented by the oracle is the hand-optimized, statically performed AoS-SoA transformation applied to kernels that benefit from this optimization. The *Profile-Guided* column presents the speedup that we observed from using our compiler framework,

³We do not include end-to-end data for SSSP because the benchmark uses the Concat LINQ operation. Because Dandelion currently lacks a GPU-side implementation for this operation, data must be moved back and forth to the CPU for that phase of the computation at a performance cost that dominates end-to-end performance. Until GPU-side Concat support is available, performance comparisons at the GPU kernel level are meaningful, but end-to-end performance measurements cannot provide an accurate picture.

Application	Profile-Guided	Oracle
BlackScholes	1.07	1.09
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 Black-Scholes, K-Means, and SkyServer. The **Profile-Guided** column shows the speedup of many iterations of the optimized code of unoptimized code including overheads to instrument, measure and re-generate code, while the **Oracle** column shows the speedup over unoptimized code that would be attained if a perfect oracle selects the optimal code version.

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, so compile overheads are not included. The *Oracle* column shows the speedup attained if the most optimal code version can be selected at the first iteration with no additional overhead.

For BlackScholes and K-Means, the potential benefit is modest (7%-9%), the profile-guided runs achieve speedups very close to those obtained with an oracle. In contrast, for SkyServer the potential benefit is significant. The SkyServer application has the most complex data structures and generated code, compared to the more simpler implementation of the other two workloads. As a result, the instrumentation overheads are more significant as well. We note that the difference in maximum profitability of optimization across these workloads is expected. BlackScholes and K-Means are compute-bound, so their lower memory intensity relative to SkyServer translates to fewer sites at which instrumentation code is inserted and a correspondingly less runtime overhead relative to SkyServer.

The data also show that as one would expect, 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, programming abstractions and runtime tools that maximize asynchrony and/or eliminate unnecessary data migration can lessen this impact, enabling our framework to deliver higher gains in future systems. In the future, we will explore application of finer-grained instrumentation techniques to alleviate instrumentation overheads for memory-bound codes, such as selectively instrumenting only certain kernel segments or certain types of memory operations. This may represent a profitable avenue for lowering overheads if sampling alone is sufficiently pre-

dictive.

5 Related Work

Profile-Guided Optimization. Adaptive, dynamic, and profile-guided optimization techniques have enjoyed much research attention over the past few decades [2]. 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. The research community has focused considerable effort on higher-level programming front-ends for GPUs [18, 6, 4], with the primary goal of insulating the programmer from complexities induced by the architecture. The Delite compiler and runtime framework [4] performs domain-specific optimizations on DSL applications for execution on multiple heterogeneous back ends. Delite shares many common features with our framework: support for programmer productivity without sacrificing performance, representation of applications as execution graphs to enable runtime scheduling and optimizations, and heterogeneous code generation for both CPUs and GPUs. The optimizations explored by the Delite runtime focus on fusion of Delite operators to reduce memory pressure and improve cache behavior, which result in fewer memory allocations and total number of accesses, as well as runtime scheduling decisions. 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 [6] is a high-level data parallel language embedded in Python, enabling the execution of Python applications on parallel architectures. Leo’s novelty lies in its integration of a dynamic instrumentation engine, GPU Lynx, with a cross-compilation runtime, Dandelion, to enable profile-driven optimizations, transparently and seamlessly, based on the application’s runtime behavior.

GPU Optimizations. The GPU-specific optimizations, such as the AoS-SoA transformation, have been studied extensively in previous works as well [17, 21, 25, 22]. In [17], Rompf et al. show how the AoS-SoA data structure optimization can be performed via internal compiler passes. DL [21] is an OpenCL-based runtime library that provides an efficient data layout transformation engine, specifically to perform AoS-SoA type transformations. G-Streamline [25] is a framework which removes dynamic irregularities in GPU applications on-the-fly, such as those resulting from irregular memory accesses and data-dependent control accesses. Leo’s goal is to automatically and transparently determine when a par-

ticular optimization is useful or not, and respond to the given application’s varying runtime 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. The ubiquity of irregular, unstructured applications running on GPUs has made the need for an auto-optimizing framework that reacts to the application’s runtime behavior increasingly urgent. In [5], Burtscher et al. discuss the memory efficiency-cache tension that we discussed in our workloads as well. In general, GPU acceleration for irregular data parallel workloads [5, 12, 19, 20], has been studied extensively, and augments the potential value of our proposed framework for current and future heterogeneous systems.

6 Conclusion

The increasing ubiquity and attractive performance properties of parallel architectures has resulted in increased demand for generality and ease of programming, and driven the emergence of higher-level front-end programming languages and targeting GPUs. While such tools can insulate the programmer from complexity and low-level architectural detail, that insulation shifts the responsibility of efficiently exercising the architecture from the programmer to compilers and runtimes, in some cases sacrificing the goal of achieving best-case performance. Leo is a dynamic optimization framework whose goal is make such sacrifice unnecessary by enabling the system to automatically search the implementation and optimization space formerly searched by hand, by developers.

While the current implementation of Leo relies on the Dandelion compiler to optimize LINQ-based streaming workloads on NVIDIA GPUs, the same techniques and insights can be applied to optimize general data-parallel applications on any of the various GPU back-ends. Additionally, the focus of this work was to explore the challenges arising from memory access irregularity and control flow diversity in GPU codes generated by higher-level programming tools. However, Leo can be extended to perform several other kinds of dynamic optimizations, such as improving shared memory usage and/or bank conflicts, register pressure, and thread-level parallelism. Leo is a research prototype under active development: we hold that the results presented give reason to be optimistic that such a framework can effectively ameliorate many of the effort-intensive development, tuning, and optimization problems that currently characterize GPU programming.

References

- [1] J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.*, 227(10):5342–5359, May 2008.
- [2] 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.
- [3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In J. K. Hollingsworth, editor, *SC*, page 66. IEEE/ACM, 2012.
- [4] K. Brown, A. Sujeeth, H. Lee, T. Rompf, H. Chafi, and K. OLUKOTUN. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.
- [5] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC ’12, pages 141–151, Washington, DC, USA, 2012. IEEE Computer Society.
- [6] B. Catanzaro, M. Garland, and K. Keutzer. Coperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPOPP ’11, pages 47–56, New York, NY, USA, 2011. ACM.
- [7] 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. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.
- [8] 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. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’84, pages 1–8, New York, NY, USA, 1984. ACM.

- [9] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 58–67, april 2012. <http://code.google.com/p/gpulynx/>.
- [10] A. Kerr, G. Damos, and S. Yalamanchili. A characterization and analysis of ptx kernels. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 3–12, oct. 2009.
- [11] M. D. Linderman, J. D. Collins, H. W. 0003, and T. H. Y. Meng. Merge: a programming model for heterogeneous multi-core systems. In S. J. Eggers and J. R. Larus, editors, *ASPLOS*, pages 287–296. ACM, 2008.
- [12] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP ’12*, pages 117–128, New York, NY, USA, 2012. ACM.
- [13] J. Mosegaard and T. Sørensen. Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu. In *Proceedings of Eurographics Workshop on Virtual Environments*, volume 11, pages 105–111, 2005.
- [14] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.
- [15] V. Podlozhnyuk. Black-scholes option pricing. *Part of CUDA SDK documentation*, 2007.
- [16] 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 H.-J. Boehm and C. Flanagan, editors, *PLDI*, pages 519–530. ACM, 2013.
- [17] 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. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13*, pages 497–510, New York, NY, USA, 2013. ACM.
- [18] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP ’13*, pages 49–68, New York, NY, USA, 2013. ACM.
- [19] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: integrating file systems with gpus. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’13. ACM*, 2013.
- [20] 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 L. Getoor and T. Scheffer, editors, *ICML*, pages 609–616. Omnipress, 2011.
- [21] I. Sung, G. Liu, and W. Hwu. DL: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, page 111. IEEE, 2012.
- [22] 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. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP ’13*, pages 57–68, New York, NY, USA, 2013. ACM.
- [23] H. Wu, G. Damos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45*, pages 107–118, Washington, DC, USA, 2012. IEEE Computer Society.
- [24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, J. Currey, F. McSherry, and K. Achan. Some sample programs written in DryadLINQ. Technical Report MSR-TR-2008-74, Microsoft Research, May 2008.
- [25] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems. ACM*, 2011.