ZOne — A Data Parallel Language and Runtime

Abdul Dakkak Carl Pearson Li-Wen Chang
University of Illinois at Urbana-Champaign
{dakkak, pearson, lchang20}@illinois.edu

Abstract

The advent of big data introduces new challenges for GPU computing. Data, which can no longer be assumed to reside in memory, now must be read from disk. Furthermore, since data cannot fit into memory, it cannot be copied (as a whole) to the GPU. Data movement is the main bottle neck in GPU computing, since data copies dominate the program runtime.

We first present a novel asynchronous runtime, ZOne, that builds on prior work by providing a unified view of disk, GPU, and GPU memory. The runtime maintains data and compute dependencies to optimize the schedule of I/O and compute copies. To hide latency, it is able to interleave independent operations — such read from disk and copy to GPU. By providing a simplified programming model, we can improve both programmability and performance.

We also present a new high-level language and compiler. The compiler targets our asynchronous runtime by determining the dependencies in the code.The compiler demonstrates that optimizations can be done on high-level languages to transform it to performant code. We demonstrate up to 8× speedup over CUDA code with simple data transfer management (on an Intel Core i7-2820 and Nvidia Fermi Quadro 5010M) by combining the language and runtime.

1. Introduction

As the programmability and capabilities of GPUs have risen, GPUs have moved beyond their initial use as video accelerators and become a widely-used tool in the computing toolbox. Scientific and engineering programs were the first to see benefit, but with the advent of personal digital devices, big data has become another pressing issue that GPUs can be adapted to tackle.

There have been several successful non-GPU frameworks for processing big data. One of the most successful in terms of deployment is the Map-Reduce [11] programming style (an example is Hadoop [28]) which all big-data companies employ in some way. The problem with this programming style is that many programming patterns cannot be easily expressed through it, and no deployed system makes use of GPUs.

One of the reasons for this is that common GPU programming languages like OpenCL and CUDA only allow programmers to extract peak performance at the cost of considerable effort. This effort is typically focused on efficiently utilizing the GPU's on-chip resources, or carefully avoiding the caveats of massively-parallel programming. Processing "big data" only complicates programming by adding a new problem into view of the programmer - how that data should move to compute that is supposed to handle it? This problem exists in other contexts as well, but due to the scale of big data its relevance is increased.

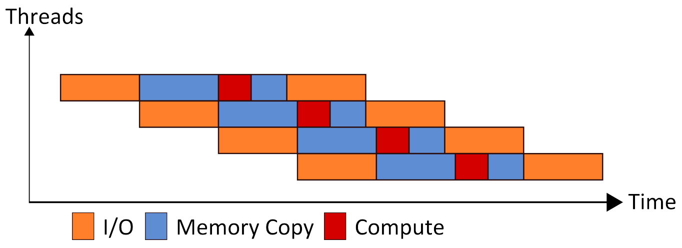

Figure 1: A representation of simple overlap of host I/O, host-to-device data transfer, and kernel execution. In this example, there are no dependencies between kernels and data so management is relateively simple.

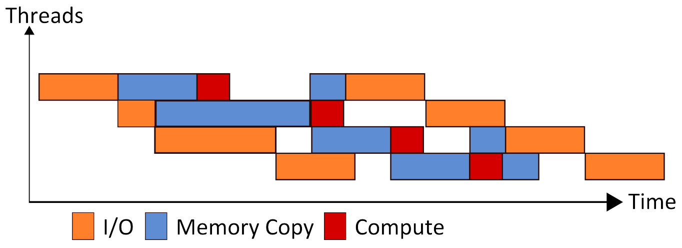

Figure 2: A more complicated oververlap of host I/O, host-to-device data transfer, and kernel execution. Arbitrary data-dependence between kernels and transfers places too large a burden on the programmer to manage properly.

CUDA and OpenCL both provide constructs that allow a certain degree of overlap between compute and data transfer. These constructs are difficult to manage even when data dependencies are simple - as data sets grow larger, the dependencies will only grow more complicated and efficient programs will be more and more difficult to create.

We have developed ZOne (a new programming language) and a parallel runtime to explore the issues of map-reduce and data transfer on GPUs. ZOne's compiler IR include `map` and `reduce` instructions to effectively map computation across GPU cores. Compiler transformations include loop fusion, loop tiling, and autotuning for improved performance on both CPU and GPU architectures. The runtime is designed from the ground up for asynchronous IO so computation may be overlapped with IO.

2. Motivation

When using GPUs to process big data, two major sources of latency are disk I/O and data transfer between the host and device. OpenCL and CUDA provide methods for managing simple patterns of overlap of data transfer and compute, shown in Figure 1. However, this does not assist with disk I/O. Furthermore, in more complicated programs the existing methods are difficult to implement efficiently.

Figure 2 shows an example of overlapping in a more complicated program varied I/O and transfer times. These dependencies are difficult to efficiently manage statically because the latencies may not be known ahead of time. Runtime management of these transfers is necessary to get efficient results.

3. Implementation

ZOne generates CUDA code for NVIDIA GPUs. It consists of a custom runtime that uses Intel Threaded Building Blocks and CUDA Streams to enable concurrency of host and device I/O, data transfer, and compute. The parser uses Parser Combinators for Dart [7], and the custom compiler analysis and transformations are written using the Dart language. This section describes how the combination of language choices, compiler, and runtime are used to hide I/O latency.

3.1 Language

Our statically typed language is inspired by array and data flow programming languages such as Fortran [4], APL [21], and LINQ [19] where one expresses computation based on operations on vectors and arrays. In our language, one can define two vectors of size `100` by

```
n :: Integer = 1000;
as :: []Real = rand.Real(n);
bs :: []Real = rand.Real(n);
```

The syntactic style for ZOne is borrowed from a other languages which we think have good syntax representations. The type declaration position was browed from Go [2], the function declarations from Dart [1], the type notation from Julia [5], and the semantics from array languages such as Fortran [4] or APL [21]. The result is a unique language that is applicable to wide array of applications.

To give you a taste of the language, in this program we approximate `pi` using Monte Carlo integration

```
def f(a :: Real, b :: Real) :: Bool {
    return a*a + b*b < 1;
}
res :: Integer = zip(as, bs).count(f) / n;
```

this would be translated into the map/reduce operations of

```
t1 = map(f, zip(as, bs));
count = 0;
reduce((x) => count += x, t1);
res = count / n;
```

The compiler would lower the above into an IR representation, maintaining the map/reduce operations. This is discussed in more detail later in the report.

The language is purely functional, disallowing any side effects. It is also statically typed with type anotation provided by the user. Due to time constraints, it lacks many features such as conditionals. This limitation can be avoided through simple bit operations.

Using parser combinators, we are able to simplify the definition of the language (compared to traditional `LR(1)` parsers). The parser is expressed within dart through a Dart library and the parser and lexer are in the same file (as compared to two files as traditionally used by `lex/yacc` [17]). The library also provides convenient primitives, such as a definition of a floating point numbers, to simplify our definition. Furthermore, it provides utility functions to declare common patterns in a convenient manner. To express function parameters, for example, first define how one function parameter is to be parsed:

```
param() =>
    identifier + typeIdentifier ^
        (id, type) => new ParameterNode(id, type);
```

Then, we use the built-in utility function `sepBy`, `parens`, and `comma` to define how a collection of parameters are to be parsed

```
params() => parens(param().sepBy(comma));
```

While using parser combinator is cleaner, it does take longer to correct that traditional LR parsers — left recursions, for example, are not allowable in parser combinators and one must restructure the parser to go around this limitation.

3.2 Compiler

We developed a simple compiler for our language. This allows us to analyze and generate low level code from our language. While our compiler borrows many techniques from production compilers, given the time constraints it is not as robust. Errors in the code will manifest themselves as uncaught exceptions in the compiler. Regardless, we think that there are some features in the design of the compiler that are worth mentioning.

3.2.1 Dart

We built our compiler using Dart [1]. Dart is an open-source language developed by Google that is designed for building scalable web apps. Dart is class-based, single-inheritance, object-oriented language with a syntax that is similar to C. It supports interfaces, generics, and abstract classes. Typing is optional - it assists with static analysis and dynamic runtime checking but it does affect the semantics of the code.

Dart was selected as the language because it has good tooling, it allowed us to start writing code without concerning ourselves with `Makefiles`, and it is close to C++ (with the option to ignore types). Our compiler and parser was just `5000` lines of Dart code.

3.2.2 Intermediate Representation

For the MonteCarlo implementation shown previously, the compiler generates the following IR.

```
Instruction(zab, zip(as, bs))
Instruction(t1, Map(f, zab))
Instruction(count, 0)
Function(g, (x) => count += x)
Instruction(r2, Reduce(g, t1))
Instruction(res, divide(count, n))
```

There are a few things to notice about the IR. First, while it is similar to 3-address form, we represent it as an S-expression internally (with a head defining what the instruction is and a body as a list of arguments). Second, `Map` and `Reduce` operations are preserved in the IR, allowing us to keep high level information. Finally, binary operations, such as + and −, are not treated specially and are simply function calls.

The compiler is able to analyze the above code, inline the map operation into the reduce function, and privatize the `count` variable. This results in efficient code that can be parallelized and does not produce unnecessary temporary arrays.

3.2.3 Compiler Passes

The most important factor in parallel computing is how to manage memory transfer. If a node computes a chunk of data and it is used in subsequent instructions, the node should reuse the output rather than send and request the data again. There are two major types of approaches to facilitate this. The first is a runtime approach: Hadoop, for example, dispatches tasks to maximize reuse of local data. This done via the Hadoop scheduler which has a mapping between nodes and data state. The second is a static approach. The compiler does the analysis to figure out dependencies and relationship between the data and the nodes. Our approach follows the static method but the framework can support either method.

Since the final generated code is C/CUDA, we can rely on the backend compiler to perform optimizations which we did not have time to implement. Note that this assumption is not valid for generated JavaScript code, but if high-performance JavaScript is needed some optimizations can be delegated to offline JavaScript optimizers such as the Google Closure [6] compiler.

3.2.3.1 Peephole Optimization

Many temporary variables are generated when the AST is lowered into the IR. They are removed through a peephole optimizer that has removal of temporary variables as a pattern.

3.2.3.2 Def/Use Analysis

The Def/Use analysis are two passes that compute which variables are defined or used at each program point. The passes also define the life span of variables. This is a simple pass, since the compiler considers the IR to be in SSA form at this stage — and while this is a fair assumption considering our language as purely functional, we do not have a pass to enforce this currently.

3.2.3.3 Free Variables

Free variables are ones which are not bound by the current scope. This pass computes the set of variables at each statement using the Def/Use information. At each program point, $OUT(p) = \cup_{e \in pred(p)} In(e)$ and $Out(p) = Out(p) \backslash Kill(p) \cup Gen(p)$ where the *Gen* set are the use set at that point, and the *Kill* set is the set of definitions.

3.2.3.4 Closure Conversion

Since ZOne is a functional language functions are first class objects. This means that some variables inside functions are not bound by the function scope. A compiler pass performs closure conversions (also called lambda lifting) to lift the function to the global scope. This pass uses the free variables pass and is done late, since other passes, such as redundant code elimination, may be able to remove the function statement.

3.2.3.5 Sharing Analysis

Sharing analysis determines the data dependency between input and output arrays. In this work, we focus on only simple lambda functions, which contain only a few statements. Only the coarse sharing pattern, which means one sharing pattern per array per lambda function, is generated. All memory accesses associated one array in a lambda function are merged to its coarse sharing pattern.

Our sharing pattern analysis is similar to array reference-based analysis [20]. However, our analysis is simplified because the feature of map operation in our language. Therefore, instead of recording all array reference in all iterations, we only record reference in a subset of iterations, which get referenced, since we know all iterations of map are independent. In this sense, we dramatically reduce the overhead of analysis.

3.2.4 Loop Fusion

If for example, in the program `map(f, map(g, lst))` a compiler pass can transform this into `map(f.g, lst)` where `f.g` is the composition of `f` and `g`. A simple peephole optimizer can scan for this instruction pattern and perform this transformation. A generalization of this technique for other list primitives is found in the the Haskell vector library. Using a concept called Stream Fusion [10], Haskell fuses most function loops to remove unnecessary temporaries and list traversals. In this project, adopt some aspects of how Haskell performs this transformation when they are applicable in the CUDA programming model. For example, GPU programs tend to be memory bound so reducing the number of temporaries increases performance.

In order to achieve high-performance fusion, a resource estimator and a memory counter are necessary. The resource estimator monitors on-chip resource usage for a current tiling to avoid register spilling and drop of multithreading. The memory counter calculates the potential memory count saved by fusion. Since 1) the on-chip resources of the GPU are limited, and 2) the performance gap between on-chip access and off-chip access on GPU might have less impact than the gap between on-node access and inter-node access on a cluster, the decision of whether to fuse is not trivial. We will use a heuristic function of these two parameters to determine the whether to fuse. A precise resource estimator might be very difficult to implement and may not really match the real vendor compiler. In this project, we will simply build a routine to rely on the feedback of the real vendor compiler after the GPU backend generated the code.

3.2.5 Loop Tiling

Tiling is critical for performant GPU programs. In this project, automatic tiling will be considered as group of several single functions `f`. Considering a function `f` is isolated and that map constructs limit the data dependency among different outputs, an analysis can be done for `f` to track data sharing among multiple outputs. Here, sharing analysis is considered as access pattern analysis of function `f`. Therefore, tiling for data sharing is achievable.

Coalesced access of GPU memory is also important for performance. In this project coalesced access is considered as a specialized type of tiling. Register packing and thread coarsening of GPU are also recognized by applying the same analysis, with different tiling. Therefore, we can potentially perform very aggressive tiling for GPU.

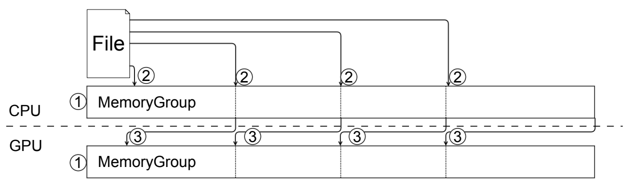

Figure 3: Memory Streaming Design

3.2.6 Autotuning

One of the advantages of compiling from a high-level languages into CUDA is that parameters for loop tiling, unrolling, and fusion can easily be tweaked. A combination of a resource model and autotuning can be used to maximize the performance of the generated code. A brute-force-like algorithm similar to the one employed by ATLAS [26] can be used to explore the parameter space.

3.2.7 Compiler Limitations

Parsing errors will manifest themselves as runtime exceptions in the parsing code and do not generate human readable syntax errors. Though the type system is required by our language sytax is not enforced by the IR, it is just propagated. It is only at the back end level that types are used to generate the appropriate variable definitions and/or function calls. Finally, a lot of functionality (such as branches and most standard math functions) are not supported either because our test cases do not require them, and they would complicate the code base.

3.3 Runtime

The ZOne runtime is primarily written in C/C++, with portions of CUDA and Intel Threaded Building Blocks. The ZOne runtime provides all normal language runtime functions with a focus on hiding I/O latency. The ZOne compiler provides information to the runtime about what tasks can be interleaved. These runtime features are built on top of CUDA streams and Intel Threaded Building Blocks. The following sections describe the implementation of the ZOne runtime. We have included the code for Black-Scholes in the appendix in listing 2 as an example code of runtime code.

3.3.1 Overview

The runtime simplifies our code generator by abstracting common operations into (essentially) a separate project. The runtime's job is to eagerly copy data to the GPU and execute the computation while maintaining the dependencies.

To unify memory, we define a new object called a `zMemoryGroup_t` This object contains multiple `zMemory_ts` which, index into a contiguious array buffer for both CPU and GPU memories. Each `zMemory_t` has a status, such as `unalloacted`, `allocated`, `dirtyDevice`, etc... that allow us to maintain conherence and avoid copies when not necessary. The memory object also contains size and type information, and the runtime operates on `zMemory_t` rather than `void *` types. This abstraction will eases adding support for other languages, such as OpenCL, without changing the code generator.

To eagerly copy memory to the GPU, we use logic similar to what is shown in Figure 3. In (1), we concurrently alloate CPU memory and GPU memory. The file is then opened in (2) and different chunks are read into the CPU memory one it's been allocated (we `mmap` the file with the `PROT_READ` and `MAP_PRIVATE` options to increase performance). Once a chunk of data is read and the GPU memory has been allocated, data is copied to the GPU in (3). Note that while we have an abstracted view of chunks of memory (defined by the datatype `zMemory_t`), memory is in fact contiguious and placed in a `zMemoryGroup_t` object. This makes it possible to resize and free memory efficiently.

3.3.2 Intel Threaded Building Blocks

Intel Threaded Building Blocks [22] (TBB) is a library for scalable parallel programming in C++. It provides templates for common parallel programming patterns and abstracts away the details of synchronization, load-balancing, and cache optimization. Instead of writing threads the programmer specifies tasks, and the library maps them onto threads in an efficient manner.

We use TBB to perform parallel file I/O as well as invoking the `cudaMalloc` call (since it does not have an asynchronus version) in a background thread. Since some parts of the code need to be performed atomically, such as setting error codes or modifying the list of memory objects used, each function takes a `zState_t` object as its first argument. The state object contains all "global" variables and locks needed to safely modify state visible from other threads.

When the state object is created, we create a set of mutexes that are to be reused (a logger, error, timer, etc... mutexes). We then use a macro to allow us to easily write these mutexes:

```
#define zState_mutexed(st, lbl, ...)          \
  do {                                        \
    speculative_spin_mutex::scoped_lock(      \
      zState_getMutex(st, zStateLabel_##lbl)\
    );                                        \
    __VA_ARGS__;                              \
  } while (0)
```

Throughout our runtime code, we use the above macro to update our state — for example, setting the error of the program to `memoryAllocate` is done via `zState_mutexed(st, Error, zState_setError(st, memoryAllocate))`.

`speculative_spin_mutex` are used to decrease overhead due to locks. `speculative_spin_mutex`es are a lock concept in TBB built on top of transactional memory. The locks and unlocks in `speculative_spin_mutex` are marked as eligible for elision, meaning that the lock is not modified in memory and the thread continues until it has reached the unlock. Any changes that the thread makes to memory will not be initially visible to other processors. If no conflicts occur during that time, the changes are atomically committed to memory.

A conflict could either take the form of a memory location being updated, or the lock being taken non-speculatively by another processor. To avoid false sharing, a `speculative_spin_mutex` takes two cache lines. This guarantees that at least one cache line boundary will exist within the object, so the actual lock portion of the class can fill up an entire cache line by itself. Presently, `speculative_spin_mutex` will only execute speculatively on 4th-generation "Haswell" Intel Core parts with TSX enabled.

3.3.3 CUDA Streams

By design, CUDA device kernels execute asycnhronously with respect to the host code. This allows host code to overlap with device code, but does not allow different device operations (kernels and data transfers) to execute concurrently. CUDA exposes device concurrency through CUDA Streams [16].

A CUDA Stream is a sequence of operations that execute in-order on a CUDA device. Separate streams may be interleaved or overlap if possible. In this way, it is possible to overlap computation and memory transfers to hide the latency of some operations. In order to effectively use streams, CUDA allows synchronization operations to occur between arbitrary streams and provides host code that allows the CPU to determine the execution status and progress of different streams.

The flexibility of CUDA streams is limited by the device hardware. For example, the Fermi architecture can manage one queue of kernels, one queue of device-to-host transfers, and one queue of host-to-device transfers. Stream dependencies between queues are maintained, but there are no dependencies within queues - the operations are simply done in the order they are put into the queue. On devices that support more than one concurrent compute operation the amount of concurrency is limited by the execution resources on the device. If both operations are too large to run concurrently they will be serialized.

CUDA streams are used for our copy operations and we register callbacks via the `cudaStreamAddCallback` function to set flags and clear mutexes once the asynchronus requrest completes. These flags and mutexes are checked before a kernel launch or copy to either make sure the data is available before a computation, or that data has already been copied and does not need to be recopied.

3.3.4 Runtime Limitations

One of the main limitations of the runtime is that the sizes of files need to be known beforehand. This is because we start allocating memory before a file is open. A slight modification to the runtime would allow us to read file meta data before starting to allocate without a noticeable impact in performance.

Another limitation is that files are assumed to be encoded as a raw byte array of a specific C type. This avoids us having to write logic to parse the file.

Both of these limitations manifest themselves because we did not want to write a generic file format or file importer, since this is a problem orthogonal to this project. In the future, file formats such as HDF [13] could be used to circumvent these implementation limitations.

The current file form limitation actually provides a performance benefit, since presently files can be parsed in parallel.

4. Evaluation

To demonstrate the performance and correctness of our ZOne implementation, we present three benchmark: binary image segmentation, convolution, Black-Scholes option pricing, and histogram. While all benefit greatly from our I/O latency interleaving, we show that segmentation improves from insight into the problem while histogram also improves from our compiler optimizations.

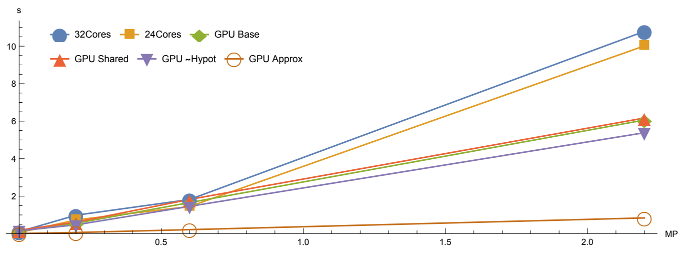

Figure 4: This shows the timing of GrowCut on a 24 and 32 core system. As expected, no difference is noticed between the two configurations due to hitting I/O bandwidth limits. The x axis is the number of megapixels in the image and the y axis is the execution time in seconds. It also shows the effects of different GPU optimizations on time.

4.1 Binary Image Segmentation

Binary image segmentation partitions an image into foreground and background pixels. GrowCut [25] is an algorithm that uses ideas from cellular automaton to perform image segmentation. GrowCut flood neighbors based on an initial seed until it reaches a barrier. One can think of each pixel as a bacteria with an energy function, if a bacteria has higher energy than one of its neighbors, then it will devour them — otherwise it is the victim. The algorithm is iterated until either a fixed point is reached or the maximum number of iterations are reached. A pseudocode of the algorithm is shown in listing 1. For our experiments, we set `MAX_ITERATIONS` to 2048 and use a penalty function $g(x, y) = 1 - \|x - y\|_2$.

We use this algorithm to show that a plateau is reached in terms of scaling. Using a standard segmentation dataset [3], we modified our runtime so it does not use the GPU and use TBB for threading. Figure 4 shows the performance on a multicore machine as well as on a GPU as the input size varies. We see that having 24 core (Intel Xeon X5660) vs 32 core (Intel Xeon X5675) does not make a difference for streaming, since one hits the I/O bandwidth limits. More advanced I/O configurations, such as RAID, would allow us to scale better as the number of cores become large.

Figure 4 shows the effects of different GPU optimizations on a C2070 GPU. The GPU base version performs the computation using an 8×8 work group. The GPU shared version places the *label*, *strength*, and *image* data into shared memory before performing the computation. Since error can be tolerated, the GPU ∼Hypot version approximates the 2-norm using the "α-max plus β-min algorithm" with $\alpha = 1$ and $\beta = 1/2$. The final version, GPU Approx, introduces more error by performing multiple iterations without doing a global synchronization. This shows that beyond program tuning for hardware, one can exploit the error tolerance for this class of algorithms to achieve great speedup.

4.2 Convolution

Convolutions has wide applications in both engineering and mathematics. Depending on the "kernel" (or mask), convolution, or stencil as it is sometimes called, can be used to approximate a differential operator — a $[-\frac{1}{h}, 0, \frac{1}{h}]$ kernel approximates the gradient operator, for example. High-performance CPU convolution implementations involve vectorization and tiling to make full use of cache bandwidth and execution resources. The biggest impact from a GPU point of view, however, is the use of fast scratch pad memory as a user managed cache. We therefore see little per-

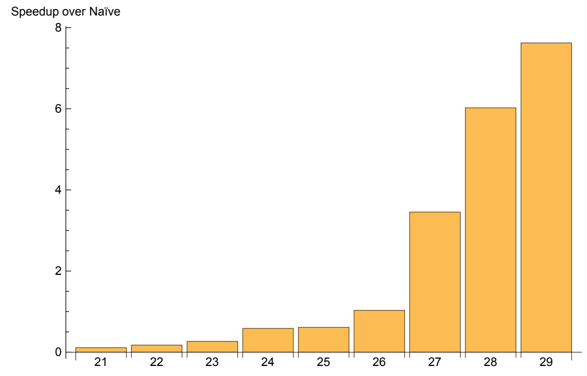

Figure 5: Speedup of convolution over naive CUDA implementation. The x-axis is the convolved vector size in bytes in a $log2$ scale.

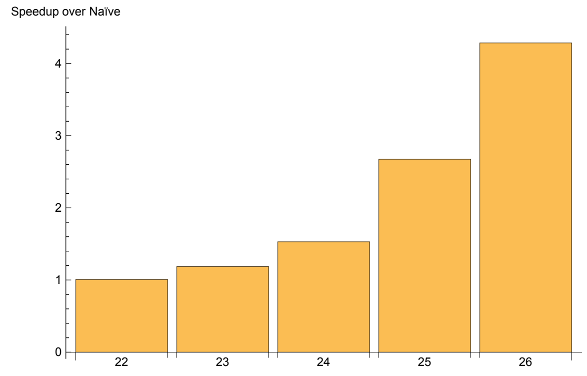

Figure 6: Speedup over CUDA implementation of BlackScholes option-pricing algorithm. The x-axis is the size of the five input datasets in floats with a $log2$ scale.

formance benefits from our compiler passes, and the performance primarily stems from interleaving memory copies. Figure 5 shows the speedup achieved by our runtime for a 1D convolution compared to a base CUDA implementation on Intel Core i7-2820 and Nvidia Fermi Quadro 5010M. At small input sizes, the speedup is less than one due to overhead from the runtime system. When the data size is large enough(2^{26}), the benefit of overlapping I/O and compute is enough to compensate for the overhead. The largest speedup shown is 7.62×. The speedup will flatten out for larger input sizes as the program's compute time becomes dominant or the data transfer becomes limited by the hardware capabilities.

4.3 Black-Scholes Option Pricing

Black-Scholes is a closed form analytic differential equation for option pricing that only considers neighboring datapoints. All data can be made private to a GPU thread, so this program maps onto GPUs with high-performance.

In Figure 6, overhead from the runtime causes the speedup to be less than 1x for datasets smaller than 2^{22} entries. The largest speedup shown is 42.8× for input data sizes of 2^{26} entries. Like Black-Scholes, the speedup will flatten out as the input data sets grow.

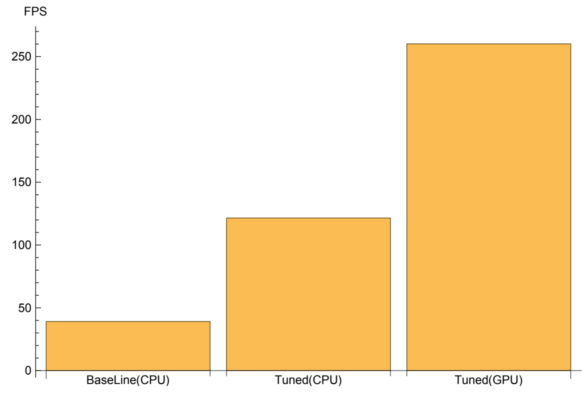

Figure 7: Frames per second achieved for histogram equalization kernel on a 4K video. The figure compares the parallel GPU implementation to a serial CPU and parallel CPU implementations. The parallel GPU implementation uses thread coarsening to achieve high fps.

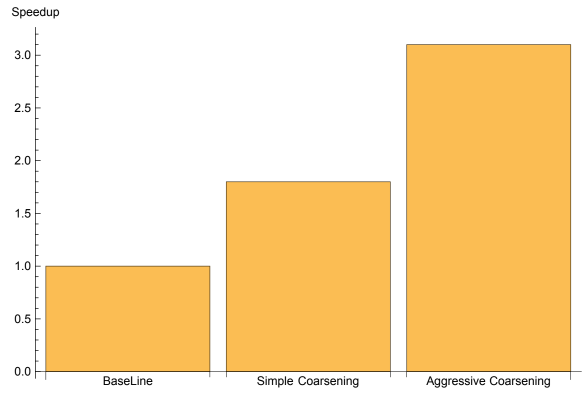

Figure 8: The coarsening level impacts the performance of the histogram kernel on the GPU.

4.4 Histogram

Histograms are a fundamental analysis tool in image and data processing. Although serial histogram implementations are very straightforward, efficient parallel histogram implementations are more involved due to the data-dependent access pattern. Atomic memory operations are typically used to enable potential parallelism for data-dependent stores. Privatization, where individual histograms for portions of the data are computed separately and then compiled together into the overall result, may potentially reduce memory contention and then penalty of serialization. Also, scan or sort operations on input data can potentially replace atomic memory operations and achieve the similar functionality.

In this image histogram equalization benchmark, atomic memory operations are specifically applied. Figure 7 shows the performance difference among serial CPU, parallel CPU, and parallel GPU versions of image histogram equalization on a 4K video. Our GPU image histogram equalization can achieve up to 260.2fps on a Tesla C2050, while serial and parallel CPU versions only have 39.1 and 120.5fps, respectively, on a Intel Xeon E5520 CPU. Tiling and thread coarsening are the two major optimization involved in this evaluation. In the histogram kernel specifically, tiling (baseline in Figure 8) is simply applied for further optimization in order to gain memory efficiency. Beyond tiling, thread coarsening (simple coarsening in Figure 8) can be applied to hide the overhead of kernels and give a 1.78× speedup over the tiling version. An aggressive thread coarsening version (aggressive coarsening in Figure 8), which involves loop fusion of independent histograms, can further have up to 3.10× over the tiling version.

5. Prior Work

Prior work in the IMPACT research group provided insight about memory management and hiding implementation details.

In GMAC [14] the authors hide CPU to GPU copy latency and unify the address space (from the programmer's point of view) of GPU and CPU devices. Our work extends this by also managing file I/O and thus unifying disk, CPU, and GPU memory.

The industry as a whole is also moving in this direction, with CUDA 5 providing a unified virtual address space and AMD's HSA providing true unified memory for APUs.

Triolet [23] is a programming interface and system for distributed-memory clusters. Triolet is able to handle task decomposition, scheduling, and communication from a program description that uses algorithmic skeletons. Triolet is able to capture parallel computation patterns and abstract away the details of the implementation. Triolet presents the programmer with higher-order functions through which expressed parallelism is automatically distributed across a cluster.

While both projects were done in the IMPACT group, this work does not borrow any code from any other project. It did benefit from some discussions with senior IMPACT members, who provided insight into the challenges to overcome. In the next section we describe related work outside the IMPACT group.

6. Related Work

This work that is related to our project emphasizes expression of data-parallelism through higher-order functions and high-level languages. In this section, we present a selection of related works.

DryadLINQ [27] is a high-level language for data-parallel computing. It is designed to handle batch operations on large-scale distributed systems. It combines strongly-typed .NET objects, general-purpose declarative and imperative statements, and LINQ expressions into a sequential program that can be debugged with a standard .NET debugger. The DryadLINQ system automatically transforms the data-parallel portions of the program into a distributed execution plan.

Thrust [15] is a parallel algorithms library for C++ resembling the C++ standard library. Thrust code can use CUDA, Intel TBB, or OpenMP as a final target to enable high performance across a variety of systems.

Copperhead [8] is a data-parallel version of a subset of Python that is interoperable with traditional Python code and Numpy. Copperhead can use CUDA, Intel TBB, and OpenMP to accelerate operations. The Copperhead runtime intercepts annotated function calls and transforms them (and their input data) to the appropriate CUDA, OpenMP, or TBB constructs before executing them.

Accelerate [18] is an embedded array language in Haskell for high-performance computing. It allows computations on multi-dimensional arrays to be expressed through collective higher-order operations such as maps or reductions. It has a CUDA and OpenCL backend.

NOVA [9] is a data-parallel polymorphic, statically-typed functional language. Parallelism is expressed through higher-order functions such as scan, reduce, map, permute, gather, slice, and filter. NOVA has a squential/parallel C backend, and a CUDA backend.

Adaptive Implementation Selection in SkePU [12] is a C++ template library for data-parallel computations on one or more GPUs through CUDA or OpenCL. SkePU programs are expressed through skeletons derived from higher-order functions. Notably, SkePU also implements lazy memory copying to avoid unecessary memory transfers.

GPU MapReduce (GPMR) [24] is a MapReduce library written for multi-GPU clusters. GPMR programs are expressed through the map and reduce higher-order functions. GPMR breaks these programs up into a map stage, a sort stage, and a reduce stage, then uses optimizations that reduce communication at the expense of computation, which is a tradeoff that is well-suited for GPUs. These optimizations are all based off of the data partitioning that the programmer selects.

7. Future Work

While ZOne demonstrates some important features of an effective parallel programming environment, there is plenty of opportunity to improve the applicability and efficiency of the code generation.

Support for OpenCL, a heterogeneous computing framework for writing programs that run on CPUs, GPUs, DSPs, FPGAs, and other available computing resources, would be a logical step for the runtime. OpenCL is inspired by CUDA and has stream-like capabilities, so many of the CUDA constructs that ZOne can generate code for can easily be retargeted to OpenCL to allow ZOne to generate efficient code for devices that are not NVIDIA GPUs.

It is worth mentioning that our aim was not to write an optimized compiler (the compiler can be very slow), our aim was to have a compiler that can generate optimized code. To that end, we have written the compiler in Dart (a Javascript inspired language) that currently generates sequential Javascript and threaded CUDA code from our language. The Javascript generation is primarily meant for debugging purposes. A robust compiler would require more optimizations passes, such as data layout transformation and vectorization. The compiler built for this project is only complicated enough to show that high-level languages can be mapped onto the runtime - a useful compiler would have to be extended with more advanced optimization passes.

References

[1] Dart, Apr. 2014. URL `http://dartlang.org`.

[2] Go, Apr. 2014. URL `http://golang.org`.

[3] Gridcut benchmark dataset, Apr. 2014. URL `http://gridcut.com/downloads.php`.

[4] J. C. Adams, W. S. Brainerd, and C. H. Coldberg. *Programmer's guide to Fortran 90.* Intertext Publications, 1990.

[5] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[6] M. Bolin. *Closure: The Definitive Guide.* " O'Reilly Media, Inc.", 2010.

[7] P. Brauner. Parser combinators for dart, Apr. 2014. URL `http://pub.dartlang.org/packages/parsers`.

[8] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: 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. ISBN 978-1-4503-0119-0. . URL `http://doi.acm.org/10.1145/1941553.1941562`.

[9] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. Nova: A functional language for data parallelism. Technical report, Tech. rep., NVIDIA, 2013.

[10] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ACM SIGPLAN Notices*, volume 42, pages 315–326. ACM, 2007.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`.

[12] J. Enmyren and C. W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.

[13] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[14] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 347–358. ACM, 2010.

[15] J. Hoberock and N. Bell. Thrust: A parallel template library. *Thrust: A Parallel Template Library*, 2009.

[16] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach.* Newnes, 2012.

[17] M. E. Lesk and E. Schmidt. Lex: A lexical analyzer generator, 1975.

[18] T. McDonell. The accelerate package, Nov. 2013. URL `hackage.haskell.org/package/accelerate`.

[19] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[20] C. Nugteren, R. Corvino, and H. Corporaal. Algorithmic species revisited: A program code classification based on array references. In *MuCoCoS '13: International Workshop on Multi-/Many-core Computing Systems, 2013*. IEEE, 2013.

[21] R. P. Polivka and S. Pakin. *APL: The language and its usage.* Prentice Hall Professional Technical Reference, 1975.

[22] J. Reinders. *Intel threaded building blocks.* O'Reilly, 2007.

[23] C. Rodrigues, T. Jablin, A. Dakkak, and W.-M. Hwu. Triolet: a programming system that unifies algorithmic skeleton interfaces for high-performance cluster computing. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 247–258. ACM, 2014.

[24] J. A. Stuart and J. D. Owens. Multi-gpu mapreduce on gpu clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068–1079. IEEE, 2011.

[25] V. Vezhnevets and V. Konouchine. Growcut: Interactive multi-label nd image segmentation by cellular automata. In *Proc. of Graphicon*, pages 150–156, 2005.

[26] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[28] P. Zikopoulos, C. Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data.* McGraw-Hill Osborne Media, 2011.

8. Appendix

Listing 1: Pseudo Code for GrowCut Segmentation

```
Input: Image, Approximate Label
Output: Label Sgementation
-----------------------------
changed = true
do_wile changed && iterations < MAX_ITERATIONS:
  changed = false
  for ii from 0 to height:
    for jj from 0 to width:
      cp = image[ii][jj]
      nl = lp = label[ii][jj]
      ns = sp = strength[ii][jj]
      for ni from -1 to 1:
        for nj from -1 to 1:
          cq = image[ii+ni][jj+nj]
          lq = label[ii+ni][jj+nj]
          sq = strength[ii+ni][jj+nj]
          gc = g(cp, cq)
          if gc * sq > sp:
            nl = lq
            ns = sq * gc
            changed = true
          end
        end
      end
      nextLabel[ii][jj] = nl
      nextStrength[ii][jj] = ns
    end
  end
  iterations++
  label = nextLabel
  strength = nextStrength
end
```

Listing 2: Black-Scholes using ZOne Runtime

```

#include "z.h"

#define BLOCK_DIM_X 64

#define N(x) (erf((x)/sqrt(2.0f))/2+0.5f)

__global__ void gpuBlackScholes(float* call,float* S,
    float* X,float* T,float* r,float* sigma,int len){

  int ii=threadIdx.x+blockDim.x*blockIdx.x;
  if(ii>len){
    return;
  }
  float d1=
  (log(S[ii]/X[ii])+(r[ii]+(sigma[ii]*sigma[ii])/2)*T
      [ii])/(sigma[ii]*sqrt(T[ii]));
  float d2=d1-sigma[ii]*sqrt(T[ii]);

  call[ii]=S[ii]*N(d1)-X[ii]*exp(-r[ii]*T[ii])*N(d2);
}

void BlackSholes(zMemory_t out,zMemory_t S,zMemory_t
    X,zMemory_t T,zMemory_t r,zMemory_t sigma){
  size_t len=zMemory_getFlattenedLength(S);
  dim3 blockDim(BLOCK_DIM_X);
  dim3 gridDim(zCeil(len,blockDim.x));
  zState_t st=zMemory_getState(out);
  cudaStream_t strm=zState_getComputeStream(st,
      zMemory_getId(out));
  gpuBlackScholes<<<gridDim,blockDim,0,strm>>>
    ((float*)zMemory_getDeviceMemory(out),
    (float*)zMemory_getDeviceMemory(S),
    (float*)zMemory_getDeviceMemory(X),
    (float*)zMemory_getDeviceMemory(T),
    (float*)zMemory_getDeviceMemory(r),
    (float*)zMemory_getDeviceMemory(sigma),
    len);
  return;
}

int main(int argc,char* argv[]){
  size_t dim=atoi(argv[1]);
  zMemoryGroup_t S=zReadFloatArray(st,"S",1,&dim);
  zMemoryGroup_t X=zReadFloatArray(st,"X",1,&dim);
  zMemoryGroup_t T=zReadFloatArray(st,"T",1,&dim);
  zMemoryGroup_t r=zReadFloatArray(st,"r",1,&dim);
  zMemoryGroup_t q=zReadFloatArray(st,"q",1,&dim);
  zMemoryGroup_t out=zMemoryGroup_new(st,
      zMemoryType_float,1,&dim);
  zMapGroupFunction_t mapFun=zMapGroupFunction_new(st
      ,"blackScholes",BlackSholes);
  zMap(st,mapFun,out,S,X,T,r,q);
  zWriteFloatArray(st,"out",out);
  return 0;
}
```