ZOne — A Data Parallel Language and Runtime

CS533: Parallel Computer Architectures Final Project

Abdul Dakkak Carl Pearson Li-Wen Chang

University of Illinois at Urbana-Champaign

{dakkak, pearson, lchang20}@illinois.edu

Abstract

add abstract.

Introduction

With the advent of personal digital devices, big data has become an issue faced not only by large companies but by regular users. Processing this data using traditional languages is both inefficient and sometimes not feasible. By their nature, for many tasks, computing on big data is a highly parallel and scalable and a multitude of solutions have been proposed to make writing programs to process big data more manageable. One of the most successful, in terms of deployment, is the Map-Reduce[11] programming style (an example is Hadoop [24]) 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.

The goal of this project is to investigate what is needed from a compiler and architecture perspective to make personal computing with large data both efficient and practical. This investigation targets four areas:

- Language development: What features and restrictions could a programming language have to enable useful parallelism?
- Compiler IR: What information could the IR contain to enable optimization of parallel code?
- Compiler transformations: What transformations are enabled by the langauge and IR choices?
- Runtime: How can the runtime be designed to support computation on large datasets?

We have developed ZOne, a new programming language, to explore these questions. 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.

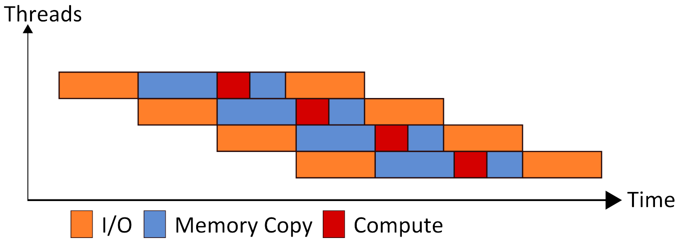

Figure 1: A representation of simple overlap of host I/O, host-to-device data transfer, and kernel execution. In this example, there are no dependencies between kernels and data so management is relateively simple.

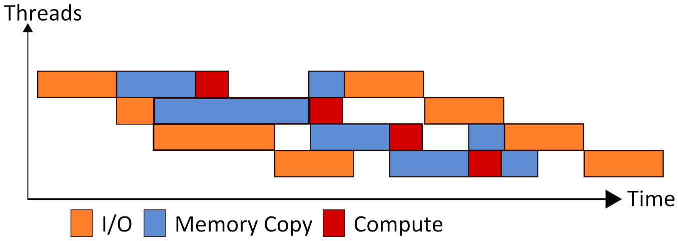

Figure 2: A more complicated oververlap of host I/O, host-to-device data transfer, and kernel execution. Arbitrary data-dependence between kernels and transfers places too large a burden on the programmer to manage properly.

1. Background

Add background

1.1 Overlaping to Hide Latency

In many cases certain overlapings are not possible, either because of hardware restrictions or language semantic restrictions. In CUDA, for example, two copy memory to device operations cannot be overlapped since they both share the same PCI-Express bus. One can overlap the compute with the copy memory to device, however. This results in the programer having to add logic to keep track of which overlaps are occuring at which point — causing the program to be more complicated. This is show in figure 2.

2. 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[8], 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.

2.1 Language

2.1.1 The Zone Language

Our statically typed language is inspired by array and data flow programming languages such as Fortran[5], APL[17], and LINQ[15] 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[3], the function declarations from Dart[2], the type notation from Julia[6], and the semantics from array languages such as Fortran[5] or APL[17]. 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, but one can go around that limitation using simple bit operations.

2.1.2 Parser

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`). The library also provides convinient primitives, such as a definition of a floating point numbers, to simplify our definition. Futhermore, it provides utility functions to declare common patterns in a convinient manner. To express function paramaters, 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.

2.2 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.

2.2.1 Dart

We built our compiler using Dart[2]. 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 were just `5000` lines of Dart code.

2.2.2 Intermediate Representation

For the MoteCarlo implementation shown in 2.1.1, 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.

2.2.3 Compiler Passes

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 deligated to offline JavaScript optimizers such as the Google Closure[7] compiler.

2.2.3.1 Peephole Optimization

When lowering the AST to IR, many temporary variables are generated. They are removed through a peephole optimizer that has removal of temporary variables as a pattern.

2.2.3.2 Def/Use Analysis

finish me.

2.2.3.3 Free Variables

finish me.

2.2.3.4 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.

2.2.3.5 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 [16]. 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.

2.2.4 Loop Fusion

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 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 compiler transformation. This is mainly done via 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, we will adopt some aspects of how Haskell performs this transformation when they are applicable in the CUDA programming model — since GPU programs tend to be memory bound, reducing the number of temporaries increases performance, for example.

In order to achieve high-performance fusion, a resource estimator and a memory counter are necessary. The resource estimator monitor 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 has less impact than the gap between on-node access and inter-node access on cluster, the decision of fusion may not be trivial. Here, 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.

2.2.5 Loop Tiling

Tiling is critical for performant on GPU programs. In this project, automatic tiling will be considered as group several single functions `f`. Considering a function `f` is isolated and map constructs limit data dependency among different outputs, an analysis can be done for fixf 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 is also important for performance. In this project coalesced access is easily 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.

2.2.6 Autotuning

One of the advantages of compiling from a high level languges into CUDA is that you can easily tweak parameters for loop tiling, unrolling, and fusion. A combination of a resource model and autotuning will be used to maximize the performance of the generated code. We will use a brute-force-like algorithm, similar to the one employed by ATLAS[22] to explore the parameter space.

2.3 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 1 as an example code of how one writes code using our runtime.

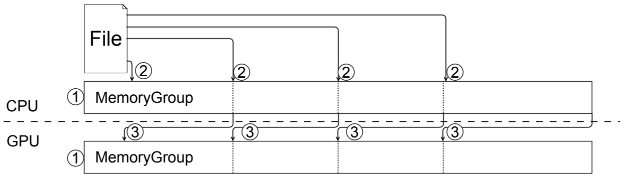

Figure 3: Memory Streaming Design

2.3.1 Overview

The purpose of the runtime is twofold. First, it simplifies our code generation by allowing us to have a unified view of memory and simplifying the code generator.Second, it eagerly copies data to the GPU.

this says simplifying code generation simplifies the code generation.

To unify memory, we define a new object called a `zMemoryGroup_t` This object contains multiple `zMemory_t`s 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, 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 free memory efficiently, and, althought not presently implemented, allows us to resize the chunks.

2.3.2 Intel Threaded Building Blocks

Intel Threaded Building Blocks[18] (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(lbl, ...)              \
  do {                                        \
    speculative_spin_mutex mutex =            \
    zState_getMutex(st, zStateLabel_##lbl);   \
    mutex::scoped_lock();                     \
    { __VA_ARGS__; }                          \
  } while (0)
```

Throughout the code, we use the above macro to update our state. For example, setting an error involves `zState_mutexed(Error, zState_setError(st, memoryAllocation))`.

Based on ...., we use `speculative_spin_mutex` to increase performance. `speculative_spin_mutexes` are a lock class in Intel Threaded Building Blocks built on top of trasactional 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 commited 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 Intel Core parts with TSX enabled. In ....

Why we use `speculative_spin_mutex`

2.3.3 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* [14].

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→host transfers, and one queue of host→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. This allows overlap of data transfer and compute on Fermi GPUs. 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.

2.3.4 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, one can use file formats, such as HDF [13], to circumvent these implementation limitations.

In the case of requiring one to parse a file to extract its data into C datatypes, we expect that we would see more performance improvement since we can perform this parsing in parallel.

more performance improvement than what? -carl

3. Evaluation

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

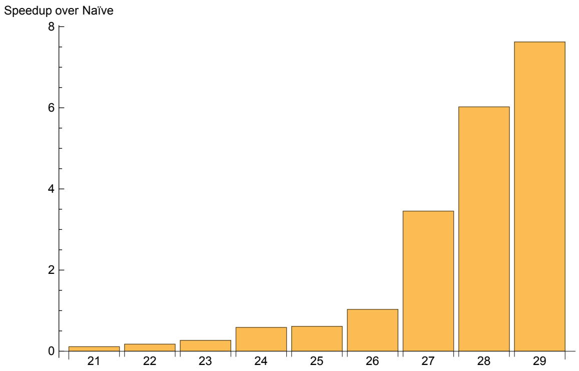

Figure 4: Speedup of convolution over naive CUDA implementation. The x-axis is the convolved vector size in bytes.

3.1 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 performance benefits from our compiler passes, and the performance primarily stems from interleaving memory copies. Figure 4 shows the speedup achieved by our runtime for a 1D convolution compared to a naive CUDA implementation.

3.2 Black-Scholes

Black-Scholes models option pricing by simplifying its formula, and some other stuff....

Coarsening increases register pressure, and degrades performance.

3.3 European Pricing

3.4 Histogram

Histograms are a fundamental analysis tool in image and data processing. Efficient serial CPU histogram implementations are very straightforward, but due to the data-dependent access pattern efficient GPU implementations are more involved. They typically feature privatization, where individual histograms for portions of the data are computed separately and then compiled together into the overall result. This reduces serialization of atomic memory accesses when different threads increment the same bin. Other approaches include sorting the input data and then finding the start index of each bucket, and approaches that use graphics-specific hardware like occlusion queries.

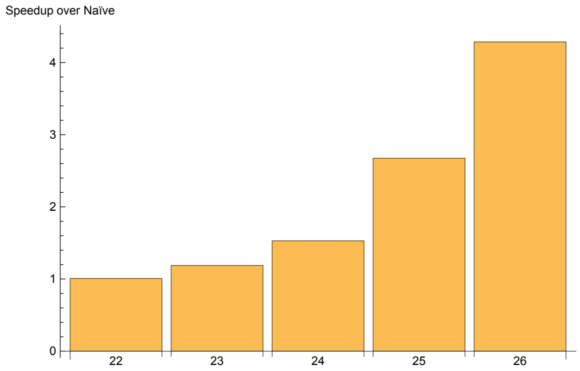

Figure 5: Speedup over CUDA version..todo

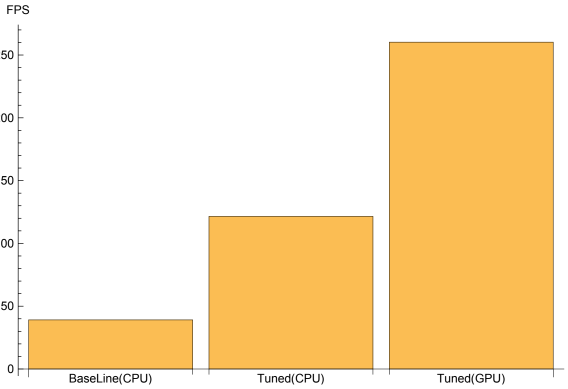

Figure 6: Frames per seconds achieved for histogram equalization kernel on a 4K video. The figure compares the tuned GPU implementation to a naive CPU and tuned CPU implementations. The tuned GPU implementation uses thread coarsening to achieve high fps.

4. Prior Work

As it pretains to prior work, the element memory management is one that has similar implementations in both academia and industry. Prior work was primarily concerned with hiding CPU to GPU copy latency [?] and/or unifying 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 drive, CPU, and GPU memory. In the next section we describe similar programming models to ours.

Discuss related work that we have in the IMPACT group?

5. 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.

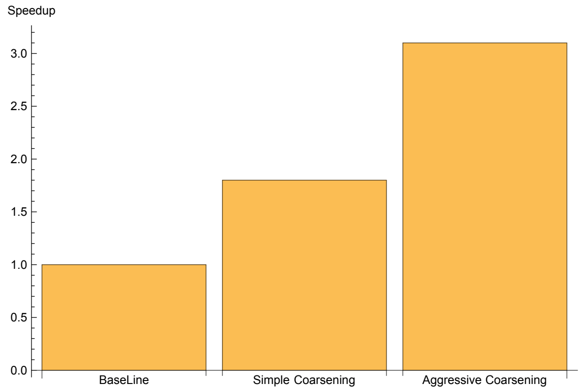

Figure 7: The coarsening parameter impacts the performance of the histogram equalization kernel.

DryadLINQ[23] 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.

Triolet[20] is a programming interface and system for distributed-memory clusters that handles task decomposition, scheduling, and communication. Triolet emphasizes algorithmic skeletons 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.

Thrust[4] 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[19] 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[1] 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)[21] 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.

6. 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.

6.1 Runtime

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.

6.2 Compiler

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.

finish

References

[1] The accelerate package, Apr. 2014. URL `http://copperhead.github.io/`.

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

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

[4] Thrust, Apr. 2014. URL `https://code.google.com/p/thrust/`.

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

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

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

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

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

[15] 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.

[16] 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.

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

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

[19] N. Research. Copperhead, Apr. 2014. URL `http://copperhead.github.io/`.

[20] 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.

[21] 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.

[22] 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`.

[23] 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.

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

Todo list

7. Appendix

Listing 1: 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;
}
```