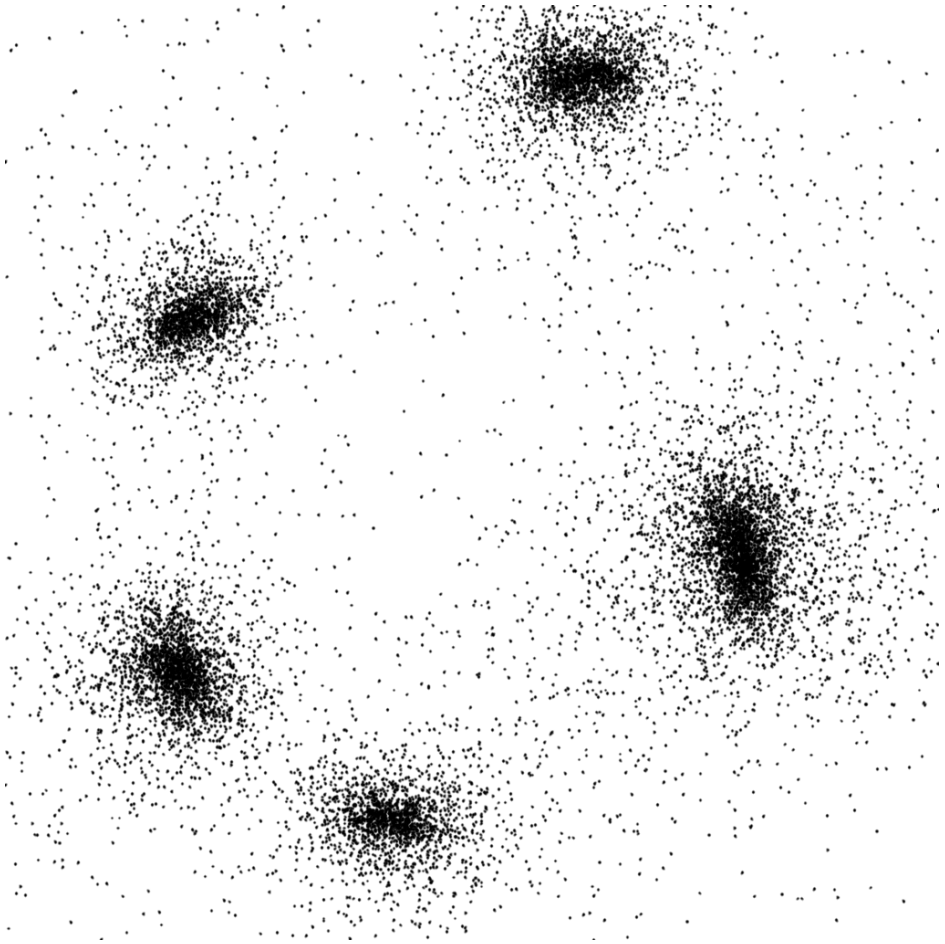
CHALMERS



Increasing programmability of an embedded domain specific language for GPGPU kernels using static analysis

Master's Thesis in Computer Science

NIKLAS ULVINGE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Increasing programmability of an embedded domain specific language for GPGPU kernels using static analysis

Niklas Ulvinge

© Niklas Ulvinge, 2014

Examiner: Mary Sheeran

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Rendering of a frame of a simulation of the n-body problem as described in Section 3.1.5.

Department of Computer Science and Engineering
Göteborg, Sweden October 2014

Abstract

GPGPU (general purpose computing on graphics processing units) programming is one interesting way to increase performance; unfortunately it is not easily done, because extensive knowledge of the GPU's architecture is required to write programs that are faster than CPU programs. Obsidian is an embedded domain specific language for writing GPGPU kernels, which tries to make GPUs more programmable, but it still requires extensive knowledge of the GPU's architecture to write fast kernels.

This thesis demonstrates extensions to Obsidian, which increase the programmability of graphics processors. The methods described in this thesis increase the programmability by providing the programmer with feedback about their code through static analysis regarding possible performance bottlenecks, and common programming mistakes. This thesis also demonstrates how many of the decisions of optimizing kernels can be automated through different code transformations. The resulting domain specific language improves upon Obsidian by requiring less knowledge of GPU programming, making it easier to write correct programs, while still providing programs that are as fast and as expressive.

The different kinds of feedback provided to the programmer using static analysis are many. Out of bounds checking and race condition detection are useful for determining correctness of code. Memory access patterns analysis for determining coalescing and bank conflict issues, divergent branch detection, unnecessary synchronization detection, and a cost model are useful for finding bottlenecks.

The code transformations used are scalar depromotion, unnecessary synchronization removal, and some traditional loop transformations that enable an arbitrarily structured program to be transformed into a kernel efficiently runnable on a GPU.

Acknowledgements

I want to thank Joel Svensson for providing guidance of the implementation of this thesis, letting me borrow hardware to perform benchmarks and being very helpful with the writing of this thesis. I would also like to thank Mary Sheeran for giving valuable feedback.

Niklas Ulvinge

Gothenburg, January 2014

Table of Contents

1

Introduction

1.1 Background

1.1.1 Parallelism

The way to increase performance of processors was until recently based on increasing the clock rate of the processor to make execution faster [Demmel, 2011]. If the performance of a single-core processor was not enough, other methods had to be used. One of the methods used was to increase the parallel execution ability of processors by using multiple processors, implementing vector units and multiple cores per processor. Utilizing these resources requires programs written with parallelism in mind, which created the fields of parallelism and concurrency.

Parallelism is the field of making a computer simultaneously execute tasks on many processors, trying to make programs run faster by using more computing resources. This often involves restructuring problems so they are more parallel. Concurrency on the other hand is the field of coordinating multiple running tasks using, for example, mutexes or message passing.

By using parallelism, the computing capability of processors has continued to advance, close to the rate of Moore's law. Memory bandwidth has, however, not increased as quickly [Demmel, 2011]. At the time of the writing of this thesis, the difference is about one and a half orders of magnitude (127 GFlop/s [Williams, 2010] vs 21 GB/s [Intel, 2013], 4 TFlop/s vs 250 GB/s [NVIDIA, 2012]), and this trend will probably continue to increase that gap.

If the memory cannot feed the processors with data fast enough, when running a particular program, it is said to be *memory bound*, or that its *bottleneck* is memory accesses. On the other hand, if a program spends more time computing than the time it takes to fetch the necessary data, the program is *compute bound* and computation is its bottleneck. These bottleneck concepts are closely related to how much memory bandwidth and computational performance a machine has. In the previous paragraph we concluded that the former increases slower than the latter, and thus compute bound programs will increase in performance faster than memory bound programs (until they too become memory bound). If a program is compute bound (and is not doing any unnecessary computation), the program can be concluded to be optimal. There can be other bottlenecks than computation and memory accesses such as: synchronization overhead and different loads on different computation units. In particular, it is common for multiplication

and addition operations to have more and faster computational units than other floating-point operations, such as trigonometric operations.

When optimizing programs, the best strategy is to try to find the biggest bottleneck and then try to make it smaller. To determine if something is a bottleneck, a common technique is to vary the suspected parameter while maintaining the others fixed and see how much the performance changes [Akenine-Moller et al., 2002]. The size of the change is proportional to how big the bottleneck is. If the performance is directly proportional to the parameter then it is the biggest bottleneck and it can be concluded that the program is bound by that parameter. An example of this is when writing graphics shaders it is often important to determine if the pixel shader is the bottleneck. The graphics pipeline works with the rasterization algorithm. For a more complete description of the graphics pipeline (which may be useful for understanding GPUs), see Akenine-Moller et al. [2002]. In brief, the rasterization algorithm works as follows: first, all triangles' vertices are run through the vertex shader, then these vertices are assembled back into triangles and a pixel is generated for every point that are inside the triangle (rasterization), then a pixel shader is run for each pixel, and the resulting value is displayed on the screen. The resolution of the rendered image is directly proportional to the number of pixel shaders run. If the resolution is varied, while keeping all other parameters the same, and the runtime is measured, it is easy to determine how big a bottleneck the pixel shader is. If the performance is greatly affected by resolution the pixel shader is a big bottleneck. If the performance is directly proportional to the resolution (that is, the runtime is ax for some constant a, and x is the resolution), then the pixel shader is the biggest bottleneck.

Computations can be modeled using a *DAG* (directed acyclic graph), in which nodes symbolize units of execution in tasks, and edges symbolize communication between these executions. Each node can only be executed when the nodes of its in edges have already been executed. This is simple if the DAG is a path (every node has only a single incident edge), then each process can simply execute the next task in the path. If a node has multiple incident edges it is a join or synchronization node that must wait for other threads before starting. The *depth* of a computation can then be formalized as the sum of the execution time for all nodes through the shortest path through the DAG. The *work* of the computation is the sum of the execution time of all nodes in the graph. Both of these concepts are usually simplified to the number of nodes in each case instead of the execution time of the sum of the nodes.

Using these concepts, we derive a couple of useful relations. One is Brent's theorem [Brent, 1974] shown in Equation (1.1) where T_p is the time to execute a program using p processors, D is the depth, and W is the work. From this we can derive the concept of *speedup* which gives us Amdahl's law shown in Equation (1.2), where T_1 is the time taken for a sequential program. From Brent's theorem we can conclude that with a single processor, the execution time of a computation is exactly the work, and with an infinite number of processors, the computation time is exactly the depth (providing in both cases there is no other overhead).

$$T_p = D + \frac{W - D}{p} \tag{1.1}$$

$$S_p = \frac{T_1}{T_p} = \frac{p}{1 + (p-1)\frac{D}{W}} \tag{1.2}$$

Figure 1.1 shows an implementation of a recursive reduction. If we imagine that each call to `sum` spawns a new task, the DAG in Figure 1.2 can be derived, where each node indicating a call to `sum`. From this we can calculate that the work is $O(n \log n)$ and the depth is $O(\log n)$. The total running time would then be $O\left(\log n + \frac{n \log n - \log n}{p}\right)$. Since the algorithm uses at most n processes this simplifies to $O\left(\frac{n \log n}{p}\right)$. The nodes in a DAG can be executed in any order, provided each nodes out-edges has been executed. This can be exploited to get a more parallel program, as shown in Figure 1.3. If the inner loop is performed in parallel, the program will have the same DAG as Figure 1.1 shown in Figure 1.2.

```
int sum(int *a, size_t s) {
  if(s<=1) {
    return a[0];
  } else {
    return sum(a,s/2) + sum(a+s/2,(s+1)/2);
  }
}
```

Figure 1.1: Code example

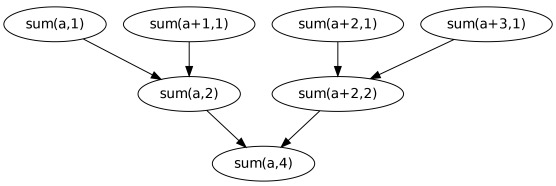


Figure 1.2: DAG of Figure 1.1 with s = 4

```
int sum(int *a, size_t s) {
  for(int i=s/2; j>0; i/=2){
    for(int j=0; j<i; j++){
      a[j] = a[2*j] + a[2*j+1];
    }
  }
}
```

Figure 1.3: Restructured reduction

1.1.2 Hardware

Most modern personal computers, smartphones, and tablets have three components: a *CPU* (central processing unit), some memory, and a *GPU* (graphics processing unit). The CPU has a few very fast cores and is specialized for low latency calculations, while the GPU has a larger number of cores, which are specialized for maintaining a high throughput of calculations. A GPU compared to a CPU can have an order of magnitude more processing power (4 TFlop/s [NVIDIA, 2012] vs 127 GFlop/s [Williams, 2010]), almost two orders of magnitude higher memory bandwidth (250 GB/s [NVIDIA, 2012] vs 21 GB/s [Intel, 2013]) [Intel, 2013; NVIDIA, 2010; Williams, 2010]. This comes at the cost of an order of magnitude more latency of all operations, and most of GPU's architecture is designed to hide this latency by running many computations concurrently.

GPUs were first designed to accelerate rendering graphics (more specifically rasterization), and for a while that was the only thing they could do. At first they could only run a very specific rasterization program. A trend of GPU architectures has been to make them able to run more general rasterization programs. The field of *GPGPU* (general-purpose computing on graphics processing units) tries to improve on this process. GPGPU programming is writing general programs (or parts of programs) for GPUs to exploit the larger processing power and higher memory bandwidth available.

A GPU consists of multiple cores (1-16 cores [NVIDIA, 2010]), which will be referred to as *SMs* (streaming multiprocessors). This is the convention NVIDIA has taken to naming the parts of their product and in describing their framework for GPU programming, and is the convention that will be used

in this thesis. Most GPUs from other manufacturers and frameworks for programming GPUs have a similar setup, but they have different names for these concepts.

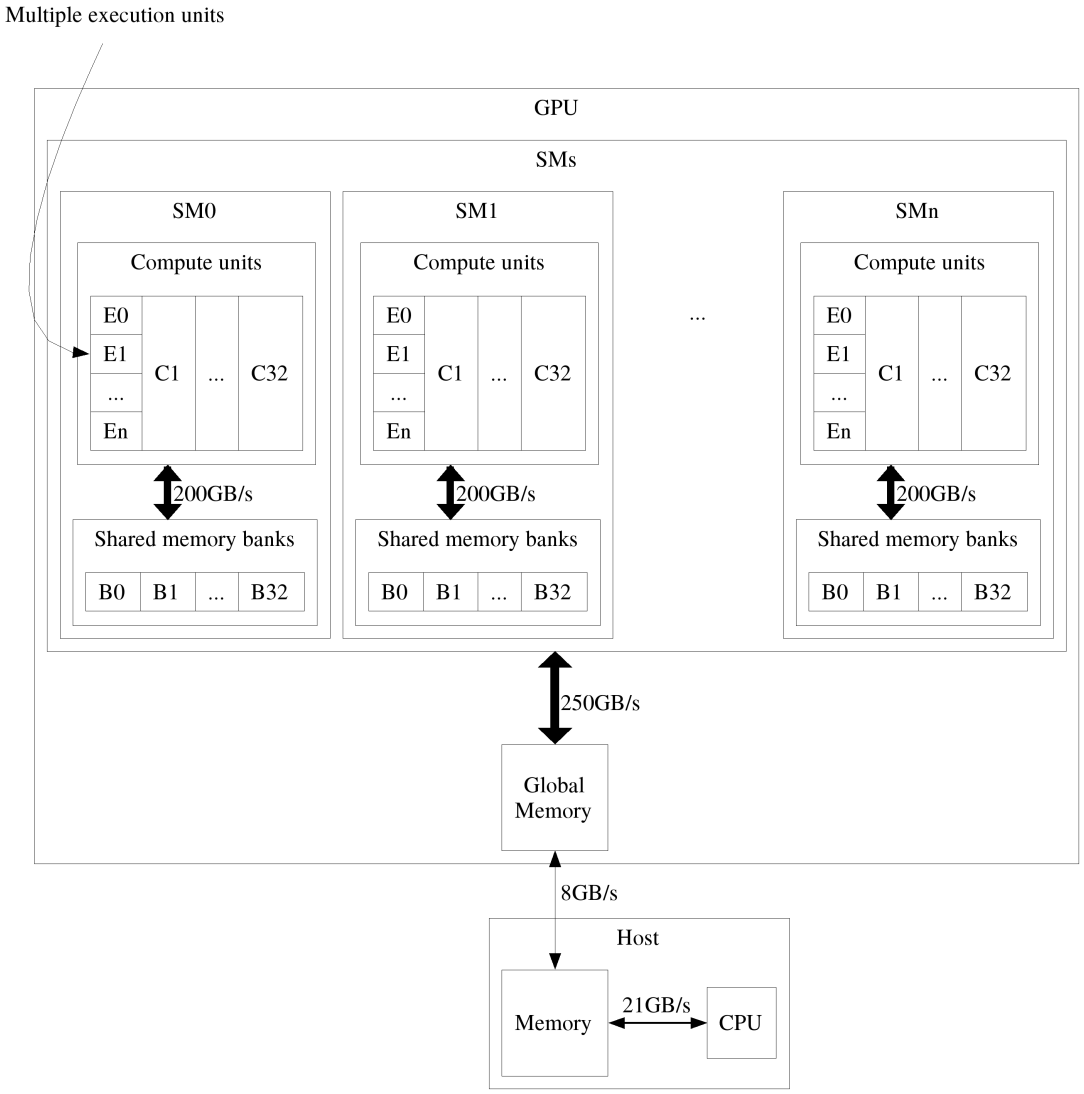


Figure 1.4: The abstraction of a GPU architecture used in this thesis

Most cores of CPUs today have a vector processing unit built into the processor that does *SIMD* (single instruction, multiple data) processing. That is, it can run the same operation on different data concurrently. The cores (SMs) of a GPU on the other hand are *SPMD* (single program, multiple data) processors. SPMD differs from SIMD by that each vector component may be at different points of execution. Another distinction is that most SIMD vector units are often 8 units wide (for floats, 32 for bytes), while the GPU's cores are typically 32 units wide. SPMD is similar to how programs running at different threads can be at different points of execution, and thus the vector units of a GPU are executing *threads*. These threads are divided into divisions of size 32 called *warps*, that are executed in lockstep (as a SIMD processor). This

is further complicated by the fact that each thread can run multiple operations concurrently. Figure 1.4 shows the abstraction of the GPU architecture used in this thesis, and shows an approximation of the bandwidth available between different communication channels.

Multiple warps can be running concurrently on a single SM, which are run in groups, and usually viewed as groups of threads. These groups of threads are called *blocks*, or *thread groups* and each block must be executed on the same SM, and multiple blocks can be executed on the same SM concurrently. Each SM has some *shared memory* which threads within blocks can use to communicate with each other, or use as temporary storage. This memory has about an order of magnitude more bandwidth than the already high bandwidth of the GPU memory. In early GPUs, this memory had as high throughput as arithmetic operations, and was thus as fast as accesses to registers [NVIDIA, 2013]. To distinguish the different memories, the normal memory of the GPU is called *global memory*. There are also a couple of other memories and caches, but those are not relevant to this thesis.

A *kernel* is a program that is run on the GPU on all threads. It has two implicit parameters (apart from the ordinary, explicit function parameters) that tell the program which thread is running and which block is running, called `threadIdx.x` (sometimes simply `tid`) and `blockIdx.x` (sometimes simply `bid`) respectively. There are more implicit parameters, such as the size of a block and the number of blocks run, called the *blocksize* and the *gridsize* respectively, but they are not used by any programs in this thesis.

All threads in a warp do not have to run all instructions, but they have to wait instead. In Figure 1.5 there is a conditional which makes half the threads perform `f()` and the other half `g()`. The runtime could be expected to be $\max(T_f, T_g)$, but this is not the case. All threads in a warp must execute the same instruction, and those that do not execute the same thing as the other threads must stall. The runtime is thus closer to $T_f + T_g$. This phenomenon is called *diverging threads*, or *divergence*. If threads need to communicate with each other, a synchronization point can be inserted which all threads will halt at until all threads have reached that point in their execution. Synchronization points cannot be put within conditionals.

```
if(threadIdx.x % 1 == 0) {
  f();    //half of the threads in a warp are waiting, the other is executing
} else {
  g();    //the reverse situation is occurring
}
```

Figure 1.5: Divergence issues causes a slowdown here.

The main different kinds of GPUs are discrete GPUs, integrated GPUs, and *APUs* (accelerated processing unit). For this thesis, integrated GPUs will be considered equivalent to APUs, which both share the property that they use the same memory as the CPU, while discrete GPUs use their own memory. When running programs on GPUs (be it GPGPU programs or graphics shaders) the CPU must first compile such a program for the specific GPU and then send the program to the GPU. If using a discrete GPU, the CPU also has to send the data that should be operated on to the GPU's memory through a PCI-express bus. This can be a very big bottleneck in many cases, since the bandwidth between the CPU and the GPU is 2.5 times lower than the memory bandwidth of the CPU (20 GB/s [Intel, 2013] vs 8 GB/s [PCI-SIG, 2010]). This bus is two and a half orders of magnitudes slower than the GPU's memory bus. For APUs this problem is nonexistent since no large amount of data has to be transferred over a slower bus.

GPUs are programmed by writing a program (a kernel) that each thread should execute, and then compiling the kernel. To run the kernel, many pieces of data must be transferred to the GPU, such as: the compiled kernel, the data the program is operating on, the parameters of the kernel (including the implicit parameters, such as the blocksize, the number of blocks, and the size of the shared memory the

kernel needs). After this is done the kernel can be run. This whole process is executed by the CPU (or *host* of the GPU) and is called *host code*.

Currently, there are two main frameworks for writing programs for GPUs: *OpenCL* (Open Computing Language) and *CUDA* (Compute Unified Device Architecture). CUDA is a proprietary framework developed by NVIDIA and only available on NVIDIA hardware. OpenCL is a framework created by Apple, but now owned and developed by the Khronos group. It is available on a wide range of hardware platforms, including NVIDIA's GPUs. These frameworks each come with a host library working on the CPU side and a device language working on the GPU, and are described by NVIDIA [2013] and Munshi et al. [2009]. CUDA has been in development for longer, which may be the reason for its more widespread use and more complete platform. For most purposes, these platforms are almost equivalent, and many projects provide backends for both of these frameworks; most relevant is that Obsidian (see Section 1.3.1) has a working CUDA backend and an OpenCL backend is being developed.

1.1.3 Embedded domain specific languages

Embedded Domain Specific Languages (EDSL, will also be referred to as DSL) is a technique frequently used in functional programming. The technique is composed of two steps: first, write a program in a made-up language that solves the problem easily, second, write an interpreter for that language. If solving whole classes of similar problems, the interpreter can be re-used if the language has been carefully designed. The EDSL can either be made up of functions that each have the desired effect, a DSL with shallow embedding, or it can be made up of a data structure, a DSL with a deep embedding. The former has the advantage of being simpler and smaller. The latter has the advantage of enabling multiple interpreters to run the program.

A small, simplified example of a deep EDSL, is the expression data type (which happens to be the expression type used in Obsidian, see Section 1.3.1) shown in Figure 1.6. With this type it is possible to model all possible expressions using the provided operators. For example `2*3+4` becomes `(BinOp Add (BinOp Mul (Literal 2) (Literal 3)) (Literal 4)` using Haskell's built in operator precedence and the fact that `Exp` is an instance of the `Num` class. The `showExp` and `calcExp` functions in Figure 1.6 illustrate the advantage of a deep DSL. There are multiple things one might do with an expression, such as showing it, and calculating the result of the expression. Another thing that can be done to expressions is generating the equivalent C code that will calculate the expression, and this is essentially how Obsidian works.

```
data Exp a where
  Literal :: a -> Exp a

  BinOp   :: Op ((a,b) -> c)
          -> Exp a -> Exp b -> Exp c

data Op a where
  Add :: Num (Exp a) => Op ((a,a) -> a)
  Sub :: Num (Exp a) => Op ((a,a) -> a)
  Mul :: Num (Exp a) => Op ((a,a) -> a)

showExp :: Exp a -> String
calcExp :: Exp a -> a
```

Figure 1.6: An example of a deep EDSL. A simplification of the expression type used in Obsidian

1.2 Objective

As described in previous sections, GPU programming is advantageous for high performance applications, and it is non-trivial to write programs for GPUs that are faster than serial implementations. The task of this thesis is to make implementing GPU kernels easier, by providing tools to evaluate and optimize the performance of kernels.

1.2.1 Scope / limitations

Since the general problem of program optimization is a very large and hard problem (in fact some subproblems of it are NP-complete), this thesis will focus on programs that are often implemented on GPUs, while trying to maintain generality. This section will outline which assumptions we make and in which way the problem is restricted. GPUs are designed for data parallelism (described in Section 1.3) and thus this thesis will only consider that kind of parallelism. Furthermore, the project will work with a number of assumptions:

- Since compute bound kernels are less of a problem than memory bound kernels (see Section 1.1.1), this thesis will focus on the latter and assume that the kernels we are working with are not compute bound and will not become compute bound. That is, all computational operations have only negligible effect on performance.
- Memory optimizations and computational optimizations are assumed to be orthogonal problems; the performance of one does not affect the performance of the other. Furthermore, computation optimizations are assumed to be adequately performed by the compiler compiling the generated code.
- Shared memory is necessary for constructing efficient kernels and should be used as much as possible (for example Larsen [2011] caches global memory in shared memory and achieves a speedup of 8) . The usage should not be limited to a cache of any of the slower memories. Furthermore, registers (variables) are assumed to be even faster than shared memory.

1.3 Related work

Vector machines are a typical example of a flat parallelism execution model (also called *data parallelism* when each node executes the same program, see Figure 1.7 for an example). The operations of the program can do just one thing to multiple data in parallel. These subtasks cannot spawn other tasks, and in the case of vector machines each task must be a primitive operation. NESL [Blelloch, 1995] introduced the concept of nested parallelism in which these restrictions are lifted (see Figure 1.9 for an example). In nested parallelism each subtask can do subtasks in parallel. Thus we can construct arbitrary task trees with nested parallelism. In NESL these nested parallelism programs are turned into flat parallelism programs using a flattening transformation [Blelloch & Sabot, 1990]. When doing this transformation on programs they become easier to parallelize, but at the cost of requiring more memory transfers, which often is the biggest bottleneck of programs. One reason for this is that the flattening transformation destroys some information about locality.

It is important to distinguish regular nested parallelism and irregular nested parallelism (see Figures 1.8 and 1.9 respectively for examples of these concepts). With regular nested parallelism, all subtasks on

the same level must spawn the same number and the same subtasks. Irregular nested parallelism does not have this restriction. The problem this thesis addresses is different from the approach of Blelloch [1995] because we are not targeting vector machines that can only execute flat parallel programs, but we are focusing on transforming regular nested parallel programs into programs that run well on two level nested parallel machines, such as GPUs. Our approach can be extended to any number of nested parallelism levels. Other types of parallelism that may be relevant are using vectorization (which is done in this thesis too), using multiple GPUs per machine, and using multiple machines. I will refer to this general architecture as *multiply nested*.

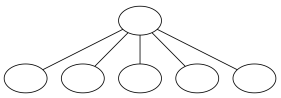


Figure 1.7: Flat parallelism

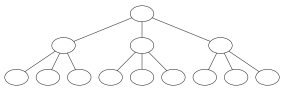


Figure 1.8: Nested regular parallelism

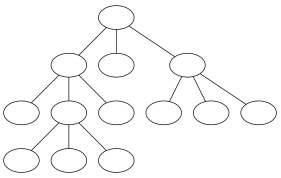


Figure 1.9: Nested irregular parallelism

An example of a DSL for flat parallelism is Accelerate [Chakravarty et al., 2011]. Accelerate is a DSL operating on n-dimensional matrices, vectors, and scalars. It has libraries for all the usual algorithms of linear algebra. Parallelism is achieved by using some parallel primitives, such as `map`, `fold`, `scan`, and `zipWith`. This is the approach of many other frameworks as well, most notably Nikola [Mainland & Morrisett, 2010] and PyCUDA [Klöckner et al., 2009].

Accelerate has multiple backends: OpenCL, CUDA, and Cilk. The GPU backends work by running a kernel for each primitive operation. This is not optimal since running multiple kernels has some performance overhead. A new approach presented by [McDonell et al., 2013] is to divide operations into two classes of operations: consumers and producers, of which producer kernels can merge with other producer kernels and consumer kernels coming later in the operations DAG. This process is not that dissimilar to `Pull` and `Push` arrays described in the next section. A limitation of Accelerate's approach is that it does not use shared memory effectively, which may be attributed to the fact that it only expresses flat parallel programs. There have been approaches to using shared memory more effectively in combination with approaches similar to Accelerate's by letting threads that access the same data do so by first loading the data into shared memory and then accessing the data from there [Larsen, 2011]. This alleviates some of the problems with approaches similar to Accelerate, but it still can only deal with flat parallelism. [Baskaran et al., 2008] goes even further and provides even more optimizations, at the cost of compiler complexity.

For GPU programming there is little previous work in the area of estimating the runtime of a kernel from the kernel code [Hong & Kim, 2009]. Hong & Kim [2009] showed a technique for estimating the runtime of a kernel, achieving an estimate of the measured runtime with average geometric mean error of 5.4%. The technique described in Section 2.1 could be extended with this approach.

1.3.1 Obsidian

This thesis builds on previous work on Obsidian [Claessen et al., 2008]. There have been many different versions of Obsidian, but this thesis will use the version described in this thesis, Svensson [2013], and Claessen et al. [2012]. Obsidian is an EDSL for GPU programming. The deep embedding of Obsidian is shown in Figure 1.10. `Program t a` forms a monad over `a`. `t` is a parameter specifying the level the code operates on in the GPU hierarchy shown in Figure 1.11.

```
data Program t a where
  Identifier :: Program t Identifier

  Assign :: Scalar a
            => Name
            -> [Exp Word32]
            -> (Exp a)
            -> Program Thread ()

  Cond :: Exp Bool
          -> Program Thread ()
          -> Program Thread ()

  SeqFor :: EWord32 -> (EWord32 -> Program t ())
            -> Program t ()

  ForAll :: EWord32
            -> (EWord32 -> Program t ())
            -> Program (Step t) ()

  Allocate :: Name -> Word32 -> Type -> Program t ()
  Declare  :: Name -> Type -> Program t ()
  Sync     :: Program Block ()

  Return :: a -> Program t a
  Bind   :: Program t a -> (a -> Program t b) -> Program t b
```

Figure 1.10: The deep embedding of Obsidian kernels

```
data Step a -- A step in the hierarchy
data Zero

type Thread = Zero
type Block  = Step Thread
type Grid   = Step Block
```

Figure 1.11: The hierarchy of programs in Obsidians

The programmer should almost never have to use any of the primitives in Figure 1.10, but instead interacts through a couple of shallowly embedded libraries of which the biggest part is the libraries relating to Arrays. Obsidian has two kinds of arrays: Pull arrays and Push arrays shown in Figure 1.12. `s` is a size parameter and can be `Word32` or `Exp Word32` (see Section 1.1.3 for an introduction to `Exp`), the former being a constant and the latter a variable. `p` is the type of the level in the hierarchy. A Pull array

describes how to retrieve data from an array given an index. A Push array instead describes how, given a writing function taking an element and an index, an array can be written. Converting between these two representations is done with the functions `force` and `push` shown in Figure 1.12. Other important functions in the array library are shown in Figure 1.15.

```
type EWord32 = Exp Word32
data Pull s a = Pull s (EWord32 -> a)
data Push p s a = Push s ((a -> EWord32 -> Program Thread ()) -> Program p ())

force :: Push p s a -> Program Block (Pull a)
push :: Pull s e -> Push p s e
```

Figure 1.12: The Pull and Push array data types with conversion functions

A very simple example of an Obsidian program is SAXPY shown in Figure 1.13, which calculates `y[i]+a*x[i]` for every `i`. The programs parameters are two Pull arrays and it returns a Push array. It works by first zipping the two arrays with `zipp`, then splitting them into chunks of 256 elements each (becoming the blocksize). Each of these chunks is then mapped over with the SAXPY program turning them into Push arrays with `push`. Finally `pConcat` takes an array of Push arrays and turns them into a big Push array that can then generate the CUDA code shown in Figure 1.14.

```
saxpy :: (Num a, ASize l)
      => a -> Pull l a -> Pull l a -> Push l a
saxpy a x y = pConcat
            $ fmap (push . fmap (\(x,y) -> y+a*x))
            $ splitUp 256
            $ zipp (x,y)
```

Figure 1.13: Obsidian program of SAXPY.

```
__global__ void kernel(int* input2,int* input1,int input0,int* output0){
    output0[(threadIdx.x+(blockIdx.x*256))] =
      (input2[((blockIdx.x*256)+threadIdx.x)]
      +(input0*input1[((blockIdx.x*256)+threadIdx.x)]));
}
```

Figure 1.14: Generated CUDA code of saxpy

```
instance Functor (Pull w) where
  fmap f (Pull n ixf) = Pull n (f . ixf)

instance Functor (Push w) where
  fmap f (Push n p) = Push s $ \wf -> p (\e ix -> wf (f e) ix)

pConcat :: ASize l => Pull l (SPush a) -> Push l a
pConcat arr =
  Push (len arr * fromIntegral rn) $ \wf ->
    forAll (sizeConv $ len arr) $ \bix ->
      let (Push _ p) = arr ! bix
      in p (\a i -> wf a (i+bix*sizeConv rn))
  where
    rn = len $ arr ! 0

splitUp :: (ASize l, ASize l2, Num l)
        => l -> Pull l2 a -> Pull l2 (Pull l a)
splitUp n (Pull m ixf) = Pull (m `div` fromIntegral n) $
                           \i -> Pull n $ \j -> ixf (i * (sizeConv n) + j)

coalesce :: (ASize l, Num l)
        => l -> Pull l a -> Pull l (Pull l a)
coalesce n arr =
  Pull s $ \i ->
    Pull n $ \j -> arr ! (i + (sizeConv s) * j)
  where s = (len arr) `div` n

unzipp :: ASize l =>  Pull l (a,b) -> (Pull l a, Pull l b)
unzipp arr = (Pull (len arr) (\ix -> fst (arr ! ix))
             ,Pull (len arr) (\ix -> snd (arr ! ix)))

zipp :: ASize l => (Pull l a, Pull l b) -> Pull l (a, b)
zipp (arr1,arr2) = Pull (minE (len arr1) (len arr2))
                      $ \ix -> (arr1 ! ix, arr2 ! ix)


halve' arr = (Pull n2     (\ix -> arr ! ix)
             ,Pull (n-n2) (\ix -> arr ! (ix+fromIntegral n2)))
  where n = len arr
        n2 = n `div` 2

evenOdds :: ASize l => Pull l a -> (Pull l a, Pull l a)
evenOdds arr = (Pull (n-n2) (\ix -> arr ! (2*ix)) ,
                Pull n2     (\ix -> arr ! (2*ix + 1)))
  where n  = len arr
        n2 = div n 2
```

Figure 1.15: Relevant parts of the array library for Obsidian

1.3.2 Obsidian Reverse Example

A very simple kernel is the kernel that reverses an array. Reversing a Pull array is a simple operation that is included in the array library and shown in Figure 1.16.

```
reverse :: ASize l => Pull l a -> Pull l a
reverse arr = Pull n (\ix -> arr ! ((sizeConv m) - ix))
  where m = n-1
        n = len arr
```

Figure 1.16: Reverse Pull array

If we now want to construct a kernel from this we need to turn the pull array into a Push array as shown with the `reverseL` function in Figure 1.17. Now we have a program that operates on blocks, and to turn

it into something runnable it needs to be run on each such block with `reverses`. This gives the output shown in Figure 1.18. The program we have written reverses every block of 1024 of its input. This shows all the generated code for the program, which can get quite big sometimes, and for the sake of brevity, most of the output will be omitted and only the important parts of the code will be shown.

```
reverseL :: SPull a -> BProgram (SPush Block a)
reverseL = liftM push . return . reverse

reverses :: DPull a -> DPush a
reverses = pConcatMapJoin reverseL . splitUp 1024
```

Figure 1.17: Reverse kernel

```
/* number of threads needed 1024*/
__global__ void kernel(int* input0,int* output0){

    output0[(threadIdx.x+(blockIdx.x*1024))] =
      input0[((blockIdx.x*1024)+(0x3ff-threadIdx.x))];

}
```

Figure 1.18: reverseL kernel's output

If we change `reverseL` to use `force` as shown in Figure 1.19, we instead get the code shown in Figure 1.20, which saves the intermediate result to shared memory. The analysis described in Section 2.1 shows us that this introduces an unnecessary sync between instructions 0 and 2. An instruction is in this thesis what normally turns into a statement in the resulting C code, such as synchronizations and assignments (`Sync` and `Assign` respectively in the Obsidian DSL).

```
reverseL' :: MemoryOps a => SPull a -> BProgram (SPush Block a)
reverseL' = liftM push . force . reverse
```

Figure 1.19: Revised reverse kernel

```
// Instruction: 0
((int*)sbase)[threadIdx.x] =
    input0[((blockIdx.x*1024)+(0x3ff-threadIdx.x))];
// Instruction: 1
// Unnecessary sync: 2 depends on 0 within same thread
__syncthreads();
// Instruction: 2
output0[((blockIdx.x*1024)+threadIdx.x)] =
    ((int*)sbase)[threadIdx.x];
```

Figure 1.20: reverseL' kernel's output

To write a program that reverses a whole array we instead need the program shown in Figure 1.21. This reverses the blocks, and then reverses each of those blocks in one operation as shown by the output in Figure 1.22. Note the `input0n` parameter, that specifies the size of the `input0` array.

```
largeReverse :: DPull a -> DPush Grid a
largeReverse = pConcat . reverse . pMap reverseL . splitUp 1024
```

Figure 1.21: Reverse kernel for large arrays

```
__global__ void kernel(int* input0,uint32_t input0n,int* output0){

  output0[((blockIdx.x*1024)+threadIdx.x)] =
    input0[(((((input0n/1024)-1)-blockIdx.x)*1024)
          +(0x3ff-threadIdx.x))];

}
```

Figure 1.22: largeReverse kernel's output

If the transformations in Section 2.2 are used the problem becomes much simpler since we let the compiler restructure the program. The needed code is shown in Figure 1.23 and the result is shown in Figure 1.24. In this case the array has size 16384 = 0x3fff + 1, and the strategy chooses a blocksize of 512.

```
smartReverse :: ASize s => Pull s e -> Push s e
smartReverse = push . reverse
```

Figure 1.23: A simpler reverse kernel utilizing automatic code transformations

```
uint32_t i0 = ((blockIdx.x*512)+threadIdx.x);
output0[i0] = input0[(0x3fff-i0)];
```

Figure 1.24: Output of the smartReverse kernel

2

Implementation

2.1 Static analysis

When optimizing code, the optimizer (either a programmer or a compiler) should start by optimizing the biggest bottleneck. When the optimizer then performs a code transformation to optimize the code, it will have to make sure the resulting code is not slower than the previous code. This can be done heuristically using a cost model. To find this bottleneck, some analysis must be performed. This section will describe the development of such an analysis and the integration of it into a cost model.

The rest of this thesis will introduce many processes (analyses and code transformations) which can have quite complex relationships. These relationships, including the transformations described in Section 2.2, are illustrated in Figure 2.1.

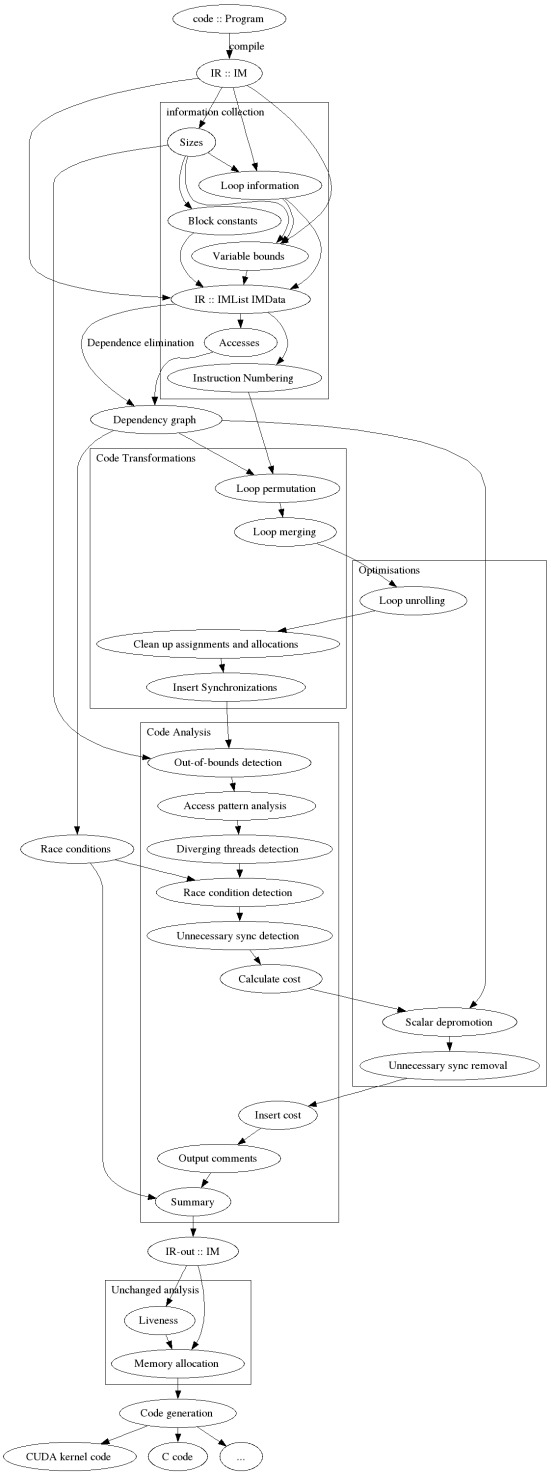


Figure 2.1: The relationships of the functions described in this thesis

2.1.1 IR analysis

When compiling Obsidian kernels, the program is first transformed into an *intermediate representation* (IR, called `IM` in the code) that is easier to work with. The two major components of this transformation are putting names on all temporary variables and removing the monadic structure in favor of a list of statements as represented by `IM`. `IM` is a specialization of `IMList t` which has a `t` for each `Statement`. At the start `t` will be empty (be the unit type `()`), but it will later be augmented with an `IMData` structure.

```
type IMList t = [(Statement t,t)]
type IM = IMList ()

data Statement t
  = SFor LoopType LoopLocation Name EWord32 (IMList t)
  | SCond (Exp Bool) (IMList t)
  | SPar [(IMList t)]
  | forall a. (Show a, Scalar a) => SAssign Name [Exp Word32] (Exp a)
  | forall a. (Show a, Scalar a) => SAtomicOp Name Name (Exp Word32) (Atomic a)
  | SOutput   Name EWord32 Type
  | SAllocate Name Word32 Type
  | SDeclare  Name Type
  | SComment String
  | SSynchronize
```

```
data IMData = IMData
  { getComments :: [String]
  , getUpperMap :: M.Map EWord32 Integer
  , getLowerMap :: M.Map EWord32 Integer
  , getBlockConstantSet :: S.Set EWord32
  , getCost :: Cost
  , getLoops :: [(EWord32, Bool)]
  , getInstruction :: Int
  }
```

The rest of this section will detail how all the necessary information in `IMData` is derived. `IMData`, along with some other data structures, is necessary to produce the analysis described in the following sections. First, we need some way to traverse an `IM`. The following functions are helpful primitives from which we can derive any traversal we could want:

```
traverseIMaccDown :: (b -> (Statement a,a) -> [((Statement a,c),b)])
                  -> b -> IMList a -> IMList c
traverseIMaccUp :: ([b] -> (Statement c, a) -> ((Statement c, c), b))
                -> IMList a -> (IMList c, [b])
traverseIMaccPrePost :: (((Statement a,a),b) -> ((Statement a,c),b))
                     -> (((Statement d,c),b) -> ((Statement d,d),b))
                     -> b -> IMList a -> (IMList d,b)
collectExp    :: (forall e. Scalar e => Exp e -> [b]) -> Exp a -> [b]
traverseExp   :: (forall e. Scalar e => Exp e -> Exp e) -> Exp a -> Exp a
collectExpIM  :: (forall e. Scalar e => Exp e -> [b]) -> IMList a -> [b]
traverseExpIM :: (forall e. Scalar e => Exp e -> Exp e) -> IMList a -> IMList a
```

From these functions, simpler traversals can be derived. The very first traversal that is done is determining the size of all internal arrays. This data is then combined with all the input and output arrays to get the sizes of all arrays. This data is for example used to determine if variables are out of bounds, as explained in Section 2.1.4.

Next, `IMData` is populated using `traverseIMaccDown`, pushing information down to the leaves. The types of information that are pushed down are:

1. Conditions from if statements
2. Ranges from for statements
3. Loop induction variables from for statements
4. If a variable is constant for all threads in a thread-block

This information is then used to construct an `IMData` for each node. The conditions are analyzed to find ranges by looking for inequalities that are turned into inequations which we attempt to solve. Loop induction variables also provide ranges for the induction variables. From this information we get the `getUpperMap` and `getLowerMap` part of `IMData` representing the ranges of variables. This data is used to calculate ranges of expressions (used, among other cases, for determining if variables are out of bounds in Section 2.1.4) by recursing on the structure of the expression and employing some algebraic identities. For example, given the range $x \in [0,2]$, we can conclude that $2 * x + 1 \in [1,5]$.

From the last point above we get `getBlockConstantSet`, telling us which variables are constant over all threads in a block (used for determining memory access patterns in Section 2.1.6). The loop induction variables give us the `getLoops` list of induction variables and whether each induction variable is a parallel loop or a sequential loop (used for calculating the work and depth cost in Section 2.1.5). `getInstruction` is initialized by traversing using `traverseIMaccDataPrePost` such that each instruction has a unique and sequential identifier. `getCost` is initialized to zero and is used as a convenience for later cost analysis. The last part of `IMData` is `getComments` which is used to output data to the user. The analyses described in the rest of this section append information to `getComments` and add cost data to `getCost`. After all the analysis has run the information is then output to the user with an `SComment` inserted before each statement, and the `IMData` is removed before the IR is transferred to the next stage.

2.1.2 Dependence analysis

A well established field in compiler technology is dependence analysis. Kennedy & Allen [2001] has a comprehensive summary of this field. This section also contains an introductory short summary of the field. The basic problem of dependence analysis is to find dependencies of statements in an imperative program. This information is useful in many types of analysis. Its main use in this thesis is in finding race conditions. If there is a dependency between two statements, then those statements should not be reordered, since that would change the behavior of the program. When executing parallel programs, the ordering can sometimes be arbitrary, and that then becomes a race condition if there is a dependence. The result of the program is arbitrary depending on the scheduler; this may sometimes be what you want, but in general, programs are supposed to be deterministic.

```
b = a; //s1
c = b; //s2
```

Figure 2.2: There exists a dependence between s1 and s2

```
for(i = 0; i < 10; ++i) {
  a[i] = b;     //s1
  c += a[i+d]; //s2
}
```

Figure 2.3: Dependencies in loops

An example is shown in Figure 2.2, where `s2` depends on `s1`, since `s1` must be executed before `s2` to yield a correct result. A more interesting example is shown in Figure 2.3. This program contains up to nine dependencies. Computing which iterations depend on each other is done by solving the equation $f(x) = g(y) \Rightarrow i_1 = i_2 + d$, where f and g are the index functions, i_1 and i_2 are the iterations that have a dependence between them. If there are multiply nested loops, the same problem is solved for each loop induction variable. We seldom want to enumerate all dependencies, since it could be costlier than the computation we want to perform. One useful thing that is often possible to compute is the distance vector calculated with: $f(x) - g(y) = d \Rightarrow i_1 - i_2 = d$. The distance vector is calculated for each loop induction variable and kept in an ordered pair. Often, the exact value of d can be hard to calculate, and the exact value is not always necessary. Instead, we can calculate the direction vector:

Direction vector	**Condition**
(<)	0 < d
(=)	0 = d
(>)	0 > d
(*)	d exists

An even simpler question to ask is whether a dependence exists or not. That is the same question as asking if there exists a solution to the equation $f(i_1) - g(i_2) = 0$ (there exists a solution to this, they are dependent, if and only if d has a value). This problem is equivalent to integer linear programming, an NP-complete problem. There are, however, many techniques for proving independence for many simpler cases. In Sections 2.1.2.1, 2.1.2.2 and 2.1.3 the techniques used in this thesis are described.

If every memory access in every instruction is enumerated, each access will, together with the instruction number, get a unique id. We can now construct a graph with accesses as nodes and the edges between them signify a dependence between the accesses. We call this a *dependence graph*. This graph is constructed by connecting every access to every other access accessing the same array, and then eliminating edges with dependence testing. If the two accesses are reads, the dependence is also eliminated, since reads can be reordered.

2.1.2.1 GCD-test

To determine if two accesses are independent, they are first written in an affine form as: $f(x) = a_0 + a_1 * x_1 + \cdots + a_n * x_n$ and $f(y) = b_0 + b_1 * y_1 + \cdots + b_n * y_n$. The GCD-test [Muchnick, 1997] is then simply done by determining if $gcd(a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_n)$ divides $b_0 - a_0$. If it does not divide, we have proved an independence.

Example To prove an independence in the following code it is necessary to determine whether the write and the read depend on each other.

```
for(i = 0; i < 10; ++i) {
  a[2*i] = 0;
  b = a[2*i+1];
}
```

$$\begin{aligned} f(i) &= 2i \\ g(i) &= 2i+1 \\ gcd(2,2) &= 2 \\ 1-0 \bmod 2 &= 1 \neq 0 \end{aligned}$$

2.1.2.2 Banerjee-test

The Banerjee test [Banerjee, 1976] involves checking ranges of expressions using the domains of variables. That is, we try to calculate the domain (the range of values the function can take) of $f(x)-g(y)$ and try to determine if the domain contains zero. If it does not, then we have proved an independence.

Example In the following code we must determine if $f(i)=i$ and $g(i)=i+10$ can intersect.

```
for(i = 0; i < 10; ++i) {
  a[i] = 0;
  b = a[i+10];
}
```

The GCD-test does not tell us anything since $gcd(1,1)=1$ divides everything. The Banerjee-test works it out as follows:

$$\begin{aligned} [0,9] &\in f(i) \\ [10,19] &\in g(i) \\ 0-19 &\not> 0 \Rightarrow \text{no information} \\ 9-10 &< 0 \Rightarrow \text{independence} \end{aligned}$$

2.1.3 Bit test

Since GPUs generally work with multiples of the warp size, accesses will often involve powers of two, and testing for independence in individual bits can often be profitable. Furthermore, many indices are expressed using bitwise operations. The bit test works by testing each bit for independence, that is, if any bit values can be proved to be distinct. The following identities are used to calculate the bit values of the two indices, where X,Y are treated as bit vectors with the indexing type `Word32`, and x,y,X_i are

individual bits:

$$
\begin{aligned}
x \wedge y &= y \wedge x \\
1 \wedge x &= x \\
0 \wedge x &= 0 \\
x \wedge x &= x \\
x \wedge \neg x &= 0
\end{aligned}
$$

$$
\begin{aligned}
x \vee y &= y \vee x \\
0 \vee x &= x \\
1 \vee x &= 1 \\
x \vee x &= x \\
x \vee \neg x &= 1
\end{aligned}
$$

$$
\begin{aligned}
x \oplus y &= y \oplus x \\
0 \oplus x &= x \\
1 \oplus x &= \neg x \\
x \oplus x &= 0 \\
x \oplus \neg x &= 1
\end{aligned}
$$

$$
\begin{aligned}
X_i &= \begin{cases} X\text{'s } i\text{th bit} & 0 \leq i < 32 \\ 0 & \text{otherwise} \end{cases} \\
(X << n)_i &= X_{i-n} \\
(X >> n)_i &= X_{i+n} \\
(X2^n)_i &= (X << n)_i \\
(X/2^n)_i &= (X >> n)_i \\
X - Y &= X + (\sim Y + 1) \\
(X + Y)_i &= X_i \oplus Y_i \oplus \text{carry}_{i-1}(X,Y)
\end{aligned}
$$

$$
\text{carry}_i(X,X) = \begin{cases} 1 & X_i \wedge Y_i \\ 0 & \neg(X_i \oplus Y_i) \\ \text{carry}_{i-1}(X,X) & X_i \oplus Y_i \\ 0 & \neg\text{carry}_{i-1}(X,Y) \wedge (X_i \oplus Y_i \text{ is a variable}) \\ \text{unknown} & \text{otherwise} \end{cases}
$$

There are four possible bit types: True, False, Unknown, and Variable. Variable also contains information about which bit is used and if the variable is negated. If the expression contains a variable, that variables bit value is propagated, using the identities above. If an operation involves multiple variables, the bit is set as unknown. If the result of an operation depends on an unknown value, the result is unknown. To test if two indices are independent (that is, always non-equal), all respective bits of the indices are compared similar to exclusively-oring them, but the bits which use the same bit of the same variable are collected into S. If any bit can be determined to be different, an independence is proven. If S contains all bits of all induction variables, an $(=)$ dependence is proved.

Example The access pattern in the following code is quite common.

```
for(i = 0; i < 4; ++i) {
  a[i<<1] = 0;
  a[i<<1|1] = 1;
}
```

To prove an independence, the indexing expressions must first be listed:

$$f(i)_j = (i << 1)_j = i_{j-1}$$
$$g(i)_j = ((i << 1) \vee 1)_j = i_{j-1} \vee 1_j$$

The next step is determining the value of each bit of the result of f and g:

$$bits(f(i)) = [i_1, i_0, 0]$$
$$bits(g(i)) = [i_1, i_0, 1]$$

Next we test each corresponding bit, trying to determine if any bits are distinct and thus prove an independence:

$$i_2 = i_2 \Rightarrow \text{No information}$$
$$i_1 = i_1 \Rightarrow \text{No information}$$
$$0 \neq 1 \Rightarrow \text{Independence!}$$

2.1.4 Out of bounds checking

Now we get to our first analysis that can output useful information to the user. Using the data acquired above, out of bounds checking is now straightforward: for each index, if possible, calculate the range of the indexing expression, and test if it is inside the size of the indexed array. To simplify the process, the expression is first linearized. This also makes it possible to identify which variables' ranges are unknown. If the index is possibly out-of-bounds or if the range of any variable is not known a helpful error message is produced.

Example Out of bounds checking catches many common off by one errors. A trivial example is the following:

```
for(i = 0; i<=128; ++i) {
  input0[i] = 0;
}
```

The output generated by the analysis is then:

```
// Out-of-bounds: definitely out of range: (0,128) does not fit in input0 with size 128
```

2.1.5 Cost

A cost model determines, given a program, a variable that is supposed to correlate with run-time or some other resource use. When executing parallel programs, a cost model using the concepts of depth and work is a good choice since we can get an estimate of the running time using p processors using Brent's theorem. Whenever a cost is added to the cost model for an instruction, we multiply the cost by the number of times it is executed sequentially to get the depth, and then multiply the work by the number of times it is executed in parallel to get the work. To make analysis more useful each cost is classified.

A very simple cost to calculate is the number of integer and floating-point operations in an instruction (although not very accurately, since compiler optimizations attempts to reduce this number after the code is generated). This number is calculated by traversing each expression in the instruction and calculating the number of operations of each type. This number is then inserted into the cost model and the work and depth of the operation are calculated by multiplying the cost with the ranges of the induction variables of all surrounding sequential loops to get the work, and then the same for parallel loops to get the depth (with special care taken to the `threadIdx` induction variable to compensate for stalling threads with divergence issues).

Every synchronization instruction is also inserted into the cost model in its own class. This cost correlates to the depth of the implemented algorithms, since the depth of an algorithm is related to how many communication stages the algorithm performs, which is exactly what a synchronization instruction enables. And since this cost correlates to the actual number of communication stages (which may be more or fewer than the depth of the algorithm since there may be extra synchronization necessary, and intra-warp communication does not require synchronization), it is a better indicator of performance than depth.

After all cost models have been calculated by the analysis in this and the next sections, the total cost is summed up and a summary is output at the end.

Example The output from the naive SAXPY implementation shown in Figure 1.13 gives us the following output (with cost analysis of the following section also shown):

```
__global__ void kernel(int* input2,int* input1,int input0,int* output0){

    // Cost: Depth:     Work: Type:
    // Cost: 8        / 2048  Operations
    // Cost: 1        / 256   Writes GlobalMem Coalesced
    // Cost: 2        / 512   Reads GlobalMem Coalesced
    output0[(threadIdx.x+(blockIdx.x*256))] =
      (input2[((blockIdx.x*256)+threadIdx.x)]
      +(input0*input1[((blockIdx.x*256)+threadIdx.x)]));

    // Cost: 1        / 256   Syncs
    __syncthreads();

    // Depth:   Work: Type:   Total cost:
    // 8      / 2048  Operations
    // 1      / 256   Syncs
    // 1      / 256   Writes GlobalMem Coalesced
    // 2      / 512   Reads GlobalMem Coalesced

}
```

2.1.6 Memory access patterns

There are many ways memory can be accessed in current GPUs:

1. A **coalesced access** occurs when accessing global memory and the index expression having the form $a + s * tid$, where a is constant for all threads in the warp, the stride of the accesses s is 1 and tid is the thread number.

2. A **broadcasted access** can occur when accessing both global and shared memory if the index expression contains only warp constant variables, that is, if all threads in a block access the same location, i.e. $s = 0$.

3. A **banked access** can only occur when accessing shared memory when all threads access different banks, e.g. $s = 1$, and a modulo the warp size is thread constant.

4. A **bank conflict access** occurs otherwise for shared memory, and results in a serialized access to each conflicting bank. The slowdown depends on how many accesses are serialized, from two times to the warp size times smaller. If using very few threads, each bank may still be accessed only once, but our analysis will not consider this special case.

5. A **serialized access** occurs otherwise for global memory and can in theory be the warp size times slower than the other accesses. In practice the difference is about 8 times slower.

To put the index expression in the $a + s * tid$ form, the expression is linearized and the components are checked to be warp-constant by data from `IMData`. Linearization is the process of turning any expression into affine form. An expression is in affine form if it can be expressed using a sum $a_1x_1 + ... + a_nx_n$ where a_n are constants and x_n are variables or more complex expressions. All expressions are already in this form (since it is possible to simply take it as $1x$) and thus the function is total. The linearization algorithm tries is to separate as many variables as possible. This enables us to easily determine if two expressions are equal with enough accuracy for our purposes. It also enables us to easily put the expression in $a + s * tid$ form by determining the coefficient of tid, and letting all other factors become a. If the expression is too complex for the linearization algorithm we can make conservative statements about the expression.

If an access is a bank conflict or a serialized access, then the access could be slower than it needs to be, and should be improved. A helpful error message is output to help the user locate these places where improvements can be made.

Cost information about accesses is also updated with the information obtained in this pass. Each access is classified according to the types listed above, whether accessing shared or global memory, and whether doing a read or a write. Examples of this is shown in Sections 2.1.5 and 2.1.10 to 2.1.12.

2.1.7 Divergent branch detection

Divergence issues, if some threads are stalled or idle when others are executing a branch, as explained in Section 1.1.2, can also be detected and notified to the user. This is done by comparing all branches containing conditions about the thread id, and then checking whether this condition affects the thread range (taking warp sizes into account). Determining if the thread range is affected is done by comparing the thread range outside and inside of the condition. If the range is not affected then the condition must have another effect, which is assumed to be divergent. An example of a program with divergence issues that is successfully detected by the technique described in this section is found in Figure 1.5.

2.1.8 Race condition detection

A race condition occurs when multiple threads try to access the same memory element. Checking for this is done by first constructing a dependence graph of all possible dependences between accesses. This graph is then simplified by using dependence testing as described in Sections 2.1.2 and 2.1.3, to prove that the accesses are independent.

Even though there may be many dependences, that is not a sufficient condition for a race condition to occur, some dependencies are natural for the program. Therefore the dependences that work on local memory and have a synchronization instruction, or are accessed by the same thread (it is a (=) dependence) or warp, are not counted as a race condition. The remaining dependencies are notified to the user.

To determine if two accesses are from the same warp, a test is needed. A simple test is to determine if there is only a single warp executing, and it is the same warp executing at both accesses, in which case it is proved that the accesses are from the same warp.

2.1.9 Unnecessary synchronization detection

Using the previously described dependence graph, it is possible to determine if a synchronization is necessary. This is done by checking if any dependence exists from after the synchronization to before the synchronization. Furthermore, all dependencies that are accessed by the same thread or warp are not counted either, and if another sync has occurred between reading and writing, this sync may not be necessary either.

2.1.10 Reduction example

Reduction (also called a *fold*) is the process of reducing an array to a single element using a binary operation. A reduction using plus is the familiar *sum* function. This section will outline the process of

writing a correct and fast reduction kernel from a novice's viewpoint. The fastest way of performing reduction on GPUs is to divide the problem into subproblems that each block can handle, and then reduce the results from those blocks. Synchronization between blocks cannot be done within a kernel (at least not portably), and thus we need to run another kernel that reduces the result.

If we start with `red1` in Figure 2.4, our analysis will give us the information visible in Figure 2.5, which shows us a kernel reducing sub-problems of size 64 for brevity, and from here on we will use a size of 1024 instead. The many coalescing issues and bank conflicts tells us we have a problem with the memory access pattern. We also see that we perform an unnecessary synchronization. Looking at the cost, we can tell that the kernel is processing 64 elements, since it is doing 64 global reads (which are not coalesced), and writing a single element. The number of operations performed is estimated to 29, which is small in comparison to the number of memory accesses. For the number of operations to matter, there needs to be about an order of magnitude more operations than there are memory accesses. Synchronization points are also costly, our program uses 7.

```
red1 :: MemoryOps a
    => (a -> a -> a)
    -> SPull a
    -> BProgram (SPush Block a)
red1 f arr
    | len arr == 1 = return $ push arr
    | otherwise    = do
        let (a1,a2) = evenOdds arr
        arr' <- force $ zipWith f a1 a2
        red1 f arr'
```

Figure 2.4: Reduction kernel in Obsidian

```
extern "C" __global__ void kernel(int *input0,int *output0);
/* number of threads needed 32*/
__global__ void kernel(int* input0,int* output0){
    extern __shared__ __attribute__ ((aligned(16))) uint8_t sbase[];
    // Coalesce: Sequential read with stride: 2
    // Coalesce: Sequential read with stride: 2
    // Instruction: 0
    ((int*)sbase)[threadIdx.x] =
      (input0[((blockIdx.x*64)+(2*threadIdx.x))]+input0[((blockIdx.x*64)+((2*threadIdx.x)+1))]);
    // Instruction: 1
    __syncthreads();
    if ((threadIdx.x<16)){
        // Coalesce: Bank conflicts with a factor of: 2
        // Coalesce: Bank conflicts with a factor of: 2
        // Instruction: 2
        ((int*)(sbase+256))[threadIdx.x] =
          (((int*)sbase)[(2*threadIdx.x)]+((int*)sbase)[((2*threadIdx.x)+1)]);
    }
    // Instruction: 3
    __syncthreads();
    if ((threadIdx.x<8)){
        // Coalesce: Bank conflicts with a factor of: 2
        // Coalesce: Bank conflicts with a factor of: 2
        // Instruction: 4
        ((int*)sbase)[threadIdx.x] =
          (((int*)(sbase+256))[(2*threadIdx.x)]+((int*)(sbase+256))[((2*threadIdx.x)+1)]);
    }
    // Instruction: 5
    __syncthreads();
    if ((threadIdx.x<4)){
        // Coalesce: Bank conflicts with a factor of: 2
        // Coalesce: Bank conflicts with a factor of: 2
        // Instruction: 6
        ((int*)(sbase+64))[threadIdx.x] =
          (((int*)sbase)[(2*threadIdx.x)]+((int*)sbase)[((2*threadIdx.x)+1)]);
    }
    // Instruction: 7
    __syncthreads();
    if ((threadIdx.x<2)){
        // Coalesce: Bank conflicts with a factor of: 2
        // Coalesce: Bank conflicts with a factor of: 2
        // Instruction: 8
        ((int*)sbase)[threadIdx.x] =
          (((int*)(sbase+64))[(2*threadIdx.x)]+((int*)(sbase+64))[((2*threadIdx.x)+1)]);
    }
    // Instruction: 9
    __syncthreads();
    if ((threadIdx.x<1)){
        // Coalesce: Bank conflicts with a factor of: 2
        // Coalesce: Bank conflicts with a factor of: 2
        // Instruction: 10
        ((int*)(sbase+16))[threadIdx.x] =
          (((int*)sbase)[(2*threadIdx.x)]+((int*)sbase)[((2*threadIdx.x)+1)]);
    }
    // Instruction: 11
    // Unnecessary sync: 12 depends on 10 within same thread
    __syncthreads();
    if ((threadIdx.x<1)){
        // Instruction: 12
        output0[(threadIdx.x+blockIdx.x)] = ((int*)(sbase+16))[threadIdx.x];
    }
    // Depth:   Work: Type:   Total cost:
    // 29     / 928   Operations
    // 7      / 224   Syncs
    // 1      / 32    Writes GlobalMem Coalesced
    // 6      / 192   Writes SharedMem Parallel
    // 1      / 32    Reads SharedMem Parallel
    // 10     / 320   Reads SharedMem BankConflict
    // 2      / 64    Reads GlobalMem Sequential
}
```

Figure 2.5: Complete output from analysis of red1 reducing sub-problems of size 64

If we change the pattern by using `halve` instead of `evenOdds` (see Figure 1.15 for definitions) as shown in Figure 2.6 we will get the output shown Figure 2.7.

```
red2 :: MemoryOps a
    => (a -> a -> a)
    -> SPull a
    -> BProgram (SPush Block a)
red2 f arr
    | len arr == 1 = return $ push arr
    | otherwise    = do
        let (a1,a2) = halve arr
        arr' <- force $ zipWith f a1 a2
        red2 f arr'
```

Figure 2.6: Reduction kernel in Obsidian

```
((int*)sbase)[threadIdx.x] =
     (input0[((blockIdx.x*1024)+threadIdx.x)]
     +input0[((blockIdx.x*1024)+(threadIdx.x+512))]);

...

if ((threadIdx.x<1)){
    ((int*)(sbase+16))[threadIdx.x] =
        (((int*)sbase)[threadIdx.x]
        +((int*)sbase)[(threadIdx.x+1)]);
}
if ((threadIdx.x<1)){
    output0[(blockIdx.x+threadIdx.x)] =
        ((int*)(sbase+16))[threadIdx.x];
}
```

Figure 2.7: Part of output from analysis of red2

The problem is now that we are using one unnecessary write to shared memory in the last iteration. This can be fixed by using a different base case as shown in Figure 2.8. This gives us the new output in Figure 2.9. Notice the cost analysis. One global write is done per 1024 global reads, and 4 synchronization instructions are needed.

```
red3 :: MemoryOps a
    => (a -> a -> a)
    -> SPull a
    -> BProgram (SPush Block a)
red3 f arr
    | len arr == 2 = return $ push $ singleton $ f (arr!0) (arr!1)
    | otherwise    = do
        let (a1,a2) = halve arr
        arr' <- force $ zipWith f a1 a2
        red3 f arr'
```

Figure 2.8: Reduction kernel in Obsidian

```
...

if ((threadIdx.x<1)){
    output0[(blockIdx.x+threadIdx.x)] =
        (((int*)sbase)[0]+((int*)sbase)[1]);
}

// Depth:     Work: Type:   Total cost:
// 1        / 32    global writes
// 9        / 1120  local writes
// 2        / 1024  global reads
// 18       / 1280  local reads
// 24       / 4352  Ops
// 4        / 2048  Syncs
```

Figure 2.9: Part of output from analysis of red3

*red*3 does 27 local memory accesses and 3 global memory accesses, which means the program may be constrained by local memory (local memory has about an order of magnitude more bandwidth), which is not optimal. The bottleneck of the program should be global memory reads, since every element must be read once, and thus it is a necessary bottleneck. The problem is that only 1024 elements are processed per 512 threads. If we make every thread do more work, we end up with the code shown in Figure 2.10 and the output in Figure 2.11. The cost analysis shows that only two synchronizations are necessary, and the number of local reads and writes has gone down significantly, but the number of global reads has increased to 8 and coalescing issues have been introduced.

```
red4 :: MemoryOps a
    => (a -> a -> a)
    -> SPull a
    -> BProgram (SPush Block a)
red4 f arr = do
    arr' <- force $ pConcatMap (return . seqReduce f)
                               (splitUpS 8 arr)
    red3 f arr'
```

Figure 2.10: Reduction kernel in Obsidian

```
int arr8;
// Coalesce: Bank conflicts with a factor of: 8
arr8 = input0[((blockIdx.x*1024)+(threadIdx.x*8))];
for (int i7 = 0;i7 < 7;i7++){
    // Coalesce: Bank conflicts with a factor of: 8
    arr8 = (arr8+input0[((blockIdx.x*1024)
                        +((threadIdx.x*8)+(i7+1)))]);
}

...

// Depth:     Work: Type:   Total cost:
// 1        / 32    global writes
// 7        / 352   local writes
// 14       / 512   local reads
// 8        / 1024  global sequential reads
// 59       / 6272  Ops
// 2        / 256   Syncs
```

Figure 2.11: Part of output from analysis of red4

Removing the coalescing issues with the help of `coalesce` gives us `red5` shown in Figure 2.12, which

gives us the output shown in Figure 2.13.

```
red5 :: MemoryOps a
    => (a -> a -> a)
    -> SPull a
    -> BProgram (SPush Block a)
red5 f arr = do
    arr' <- force $ pConcatMap (return . seqReduce f)
                               (coalesce 8 arr)
    red3 f arr'
```

Figure 2.12: Reduction kernel in Obsidian

```
int arr8;
arr8 = input0[((blockIdx.x*1024)+threadIdx.x)];
for (int i7 = 0;i7 < 7;i7++){
    arr8 = (arr8+input0[((blockIdx.x*1024)
                        +(threadIdx.x+(128*(i7+1))))]);
}
// Depth:     Work: Type:   Total cost:
// 1        / 32    global writes
// 7        / 352   local writes
// 8        / 1024  global reads
// 14       / 512   local reads
// 58       / 6144  Ops
// 2        / 256   Syncs
```

Figure 2.13: Part of output from analysis of red5

2.1.11 Scan

A scan is the operation of performing an operation on each element of an array, keeping an accumulator along the way. The +-scan program takes a list and returns, for each element the sum that element and of all previous elements. Using the `binSplit` function found in the array library it is quite easy to build a parallel scan based on Sklansky [1960], as shown in Figure 2.14.

```
sklansky :: (Choice a, MemoryOps a)
         => Int -> (a -> a -> a) -> SPull a -> BProgram (SPush Block a)
sklansky 0 op arr = return $ push arr
sklansky n op arr = do
  let arr1 = binSplit (n-1) (fan op) arr
  arr2 <- force arr1
  sklansky (n-1) op arr2
  where
    fan :: Choice a
        => (a -> a -> a) -> SPull a -> SPull a
    fan op arr = a1 `conc` fmap (op c) a2
      where (a1,a2) = halve arr
            c = a1 ! fromIntegral (len a1-1)
```

Figure 2.14: Version one of a Sklansky based scan

If not using the `binSplit` function, this kernel can be rewritten as shown in Figure 2.15. This performs all of its memory accesses in a very slow manner as shown by the cost analysis shown in Figure 2.16.

```
phase :: Choice a
      => Int -> (a -> a -> a) -> SPull a -> SPush Block a
phase i f arr =
  Push l $ \wf -> ForAll s12 $ \tid -> do
    let ix1 = tid .&. (bit i -1)
          .|. ((tid .&. (complement $ bit i - 1)) <<* 1)
        ix2 = complementBit ix1 i
        ix3 = ix2 .&. (complement $ bit i - 1) -- - 1
        i0 = (tid `mod` (bit i))
        i3 = (tid - i0)
        i1 = i0 + i3
        i2 = complementBit i1 i
    wf (arr ! ix1) ix1
    wf (f (arr ! ix3) (arr ! ix2)) ix2
  where
    l = len arr
    l2 = l `div` 2
    s12 = sizeConv l2

compose :: MemoryOps a
        => [SPull a -> SPush Block a]
        -> SPull a -> BProgram (SPush Block a)
compose [f] arr    = return $ f arr
compose (f:fs) arr = do
  arr2 <- force $ f arr
  compose fs arr2

sklansky2 :: (Choice a, MemoryOps a)
          => Int -> (a -> a -> a)
          -> SPull a -> BProgram (SPush Block a)
sklansky2 n op = compose [phase i op | i <- [0..(n-1)]]
```

Figure 2.15: Version two of a Sklansky based scan

```
// Depth:     Work: Type:   Total cost:
// 256      / 131072        Operations
// 11       / 5632  Syncs
// 2        / 1024  Reads GlobalMem Coalesced
// 2        / 1024  Writes SharedMem Parallel
// 2        / 1024  Writes GlobalMem Sequential
// 18       / 9216  Writes SharedMem Sequential
// 30       / 15360 Reads SharedMem Sequential
```

Figure 2.16: Cost analysis of sklansky2

If the elements are loaded into shared memory before performing the scan as shown in Figure 2.17 this problem can be alleviated. The analysis now shows us in Figure 2.18 that the global reads are coalesced.

```
load :: ASize l => Word32 -> Pull l a -> Push Block l a
load n arr =
  Push m $ \wf ->
    forAll (sizeConv n') $ \tid ->
      seqFor (fromIntegral n) $ \ix ->
        wf (arr ! (tid + (ix*n'))) (tid + (ix*n'))
  where
    m = len arr
    n' = sizeConv m `div` fromIntegral n

sklansky3 :: (Choice a, MemoryOps a)
          => Int -> (a -> a -> a) -> SPull a
          -> BProgram (SPush Block a)
sklansky3 n op arr = do
  im <- force $ load 2 arr
  compose [phase i op | i <- [0..(n-1)]] im
```

Figure 2.17: Version three of a Sklansky based scan

```
// Depth:     Work: Type:   Total cost:
// 256      / 131072        Operations
// 10       / 5120  Syncs
// 2        / 1024  Reads GlobalMem Coalesced
// 2        / 1024  Writes SharedMem Parallel
// 2        / 1024  Writes GlobalMem Sequential
// 18       / 9216  Writes SharedMem Sequential
// 30       / 15360 Reads SharedMem Sequential
```

Figure 2.18: Cost data for sklansky3

Using the `inplace` library, the scan described by Brent & Kung [1982] can be efficiently implemented, as shown in Figure 2.19. This scan has the advantage of having $O(n)$ work, instead of $O(n \log n)$ work which the approach based on Sklansky [1960] has. Figures 2.18 and 2.20 compares cost of the *sklansky*3 kernel and the `inplace` kernel. The depths of the two versions are similar, (although `scan1` is using twice the number of synchronizations), but the work has dropped considerably by not performing the copying necessary for an algorithm copying arrays back and forth.

```
scan1 :: (MemoryOps a) => (a -> a -> a)
      -> Pull Word32 a
      -> Program Block (Pull Word32 a)
scan1 f a = do a' <- forceInplace (push a)
               scan1' f 1 a'
               return $ pullInplace a'

scan1' :: (MemoryOps a)
       => (a -> a -> a) -> Word32
       -> Inplace Word32 a -> Program Block ()
scan1' f s' a = do
  let s = sizeConv s'
      al :: Word32
      al = fromIntegral $ len a
  when (s' < len a) $ do
    inplaceForce a $ pusha (len a) (al`div`(s'*2)) $
      \wf i -> do
        let j = 2*s*i+2*s-1
        wf j $ (a!j) `f` (a!(j-s))
  when (s'*2 < len a) $ do
    scan1' f (s'*2) a
    inplaceForce a $ pusha (len a) (al`div`(s'*2)-1) $
      \wf i -> do
        let j = 2*s*i+2*s-1
        wf j $ (a!(j+s)) `f` (a!j)
```

Figure 2.19: A scan implemented using the inplace library

```
// Depth:     Work: Type:   Total cost:
// 1        / 1024  global writes
// 1        / 1024  local writes
// 1        / 1024  global reads
// 1        / 1024  local reads
// 19       / 2272  local sequential writes
// 38       / 4544  local sequential reads
// 211      / 28064 Ops
// 20       / 20480 Syncs
```

Figure 2.20: Cost analysis for scan1

2.1.12 SAXPY

The SAXPY operation is performing the element-wise operation `y[i] += a*x[i]`. A very simple SAXPY program is `saxpy0` shown in Figure 2.21, which give use the predictable output shown in Figure 2.22.

```
saxpy0 :: (Num a, ASize l)
    => a -> Pull l a -> Pull l a -> Push Grid l a
saxpy0 a x y = pConcatMap
    (return
    .push
    .fmap (\(x,y) -> y+a*x))
    (splitUp 256 $ zipp (x,y))
```

Figure 2.21: The saxpy0 kernel

```
output0[((blockIdx.x*256)+threadIdx.x)] =
    (input2[((blockIdx.x*256)+threadIdx.x)]
    +(input0*input1[((blockIdx.x*256)+threadIdx.x)]));
// Depth:     Work: Type:   Total cost:
// 1        / 256   global writes
// 2        / 512   global reads
// 8        / 2048  Ops
```

Figure 2.22: Analysis output from the saxpy0 kernel

Increasing the work done per thread by performing eight SAXPY operations per thread as shown in Figure 2.23 should make our code faster. The output shown in Figure 2.24 does however show that this introduces coalescing conflicts.

```
saxpy1 :: (Num a, ASize l)
    => a -> Pull l a -> Pull l a -> Push Grid l a
saxpy1 a x y = pConcatMap
    (return
    .pConcatMap (return . seqMap (\(x,y) -> y+a*x))
    .splitUp 8)
    (splitUp (8*256) $ zipp (x,y))
```

Figure 2.23: The saxpy1 kernel

```
for (int i0 = 0;i0 < 8;i0++){
    // Coalesce: Sequential read with stride: 8
    // Coalesce: Sequential read with stride: 8
    // Coalesce: Sequential read with stride: 8
    output0[((blockIdx.x*0x800)+((threadIdx.x*8)+i0))] =
        (input2[((blockIdx.x*0x800)+((threadIdx.x*8)+i0))]
        +(input0*input1[((blockIdx.x*0x800)+((threadIdx.x*8)+i0))]));
}
// Depth:     Work: Type:   Total cost:
// 8        / 2048  global sequential writes
// 16       / 4096  global sequential reads
// 112      / 28672 Ops
```

Figure 2.24: Analysis output from the saxpy1 kernel

This causes non-coalesced memory accesses. These coalescing issues can be fixed by using the `coalesce` function, as shown in Figure 2.25. The output of this shown in Figure 2.26 reveals that all coalescing issues were not fixed, and an error has been introduced into the program since the indices written to and read from are different. The error made is that while `pConcat` and `splitUp` are each other's inverses, `coalesce` 's inverse is not `pConcat`.

```
saxpy2 :: (Num a, ASize l, MemoryOps a)
    => a -> Pull l a -> Pull l a -> Push Grid l a
saxpy2 a x y = pConcatMap
    (return
    .pConcatMap (return . seqMap (\(x,y) -> y+a*x))
    .coalesce 8)
    (splitUp (8*256) $ zipp (x,y))
```

Figure 2.25: The saxpy2 kernel

```
for (int i0 = 0;i0 < 8;i0++){

    // Coalesce: Sequential read with stride: 8
    output0[((blockIdx.x*0x800)+((threadIdx.x*8)+i0))] =
        (input2[((blockIdx.x*0x800)+(threadIdx.x+(256*i0)))]
        +(input0*input1[((blockIdx.x*0x800)+(threadIdx.x+(256*i0)))]));
}
// Depth:     Work: Type:   Total cost:
// 16       / 4096  global reads
// 8        / 2048  global sequential writes
// 112      / 28672 Ops
```

Figure 2.26: Analysis output from the saxpy2 kernel

To fix the issues with `saxpy2` the inverse of `coalesce`, called `pUnCoalesce` shown in Figure 2.27, must be introduced. Using `pUnCoalesce` the code becomes `saxpy3` shown in Figure 2.28 which output is shown in Figure 2.29.

```
pUnCoalesceMap f = pUnCoalesce . pMap f
pUnCoalesce :: ASize l => Pull l (SPush t a) -> Push (Step t) l a
pUnCoalesce arr =
  Push (n * fromIntegral rn) $ \wf ->
  do
    forAll (sizeConv n) $ \bix ->
      let (Push _ p) = arr ! bix
      in p (g wf)
  where
    n  = len arr
    rn = len $ arr ! 0
    s  = sizeConv rn
    g wf a i = wf a (i `div` s + (i`mod`s)*(sizeConv n))
```

Figure 2.27: Showing the implementation of pUnCoalesce

```
saxpy3 :: (Num a, ASize l, MemoryOps a)
    => a -> Pull l a -> Pull l a -> Push Grid l a
saxpy3 a x y = pConcatMap
    (return
    .pUnCoalesceMap (return . seqMap (\(x,y) -> y+a*x))
    .coalesce 8)
    (splitUp (8*256) $ zipp (x,y))
```

Figure 2.28: The saxpy3 kernel

```
for (int i0 = 0;i0 < 8;i0++){

    output0[((blockIdx.x*0x800)+(threadIdx.x+(i0*256)))] =
        (input2[((blockIdx.x*0x800)+(threadIdx.x+(256*i0)))]
        +(input0*input1[((blockIdx.x*0x800)+(threadIdx.x+(256*i0)))]));

}
// Depth:     Work: Type:   Total cost:
// 8        / 2048  global writes
// 16       / 4096  global reads
// 160      / 40960 Ops
```

Figure 2.29: Analysis output from the saxpy3 kernel

The above error that was made in `saxpy2`, shown in Figure 2.25, suggests that another function pattern should be used. The functions `pSplitMap` and `pCoalesceMap`, shown in Figure 2.30, can be introduced, and should remove the problem. These functions do not decouple the two different ways of splitting up and concatenating arrays and does not enable the error made with `saxpy2`. Using these functions we get `saxpy4` shown in Figure 2.31 which gives the same output as `saxpy3`.

```
pSplitMap n f = pConcat . pMap f . splitUp n
pCoalesceMap n f = pUnCoalesce . pMap f . coalesce n
```

Figure 2.30: The implementation of pSplitMap and pCoalesceMap

```
saxpy4 :: (Num a, ASize l, MemoryOps a)
    => a -> Pull l a -> Pull l a -> Push Grid l a
saxpy4 a x y = pSplitMap (8*256)
                 (return . pCoalesceMap 8
                   (return . seqMap (\(x,y) -> y+a*x)))
            $ zipp (x,y)
```

Figure 2.31: The saxpy4 kernel

2.1.13 Bitonic sort

The code shown in Figure 2.32 produces a bitonic sort network. The whole output is almost 700 lines long since it is unrolled. The last iteration of the output is shown in Figure 2.33. It shows that the implementation has many coalescing and divergence problems.

```
bitonicMerge1 :: (MemoryOps a, OrdE a)
              => Word32 -> Word32
              -> Pull Word32 a
              -> BProgram (Push Block Word32 a)
bitonicMerge1 s m a = do
  let s' = fromIntegral s
      m' = fromIntegral m
      b = pusha (len a) (len a) $
        \wf i -> ifThenElse ((i .&. s' ==* 0) /=* (i .&. m' ==* 0))
          (wf i (minE (a!i) (a!(i `xor` s'))))
          (wf i (maxE (a!i) (a!(i `xor` s'))))
  if s <= 1
    then return b
    else do b' <- force b
      bitonicMerge1 (s`div`2) m b'

bitonicSort1 :: (MemoryOps a, OrdE a)
             => Pull Word32 a
             -> BProgram (Pull Word32 a)
bitonicSort1 a = f 2 a
  where
    f m a | m >= len a = return a
    f m a = do b <- bitonicMerge1 m (2*m) a
               b' <- force b
               f (m*2) b'
```

Figure 2.32: Bitonic sorter

```
// Unnecessary sync:
__syncthreads();
// Diverging: This condition causes divergence issues
if ((((threadIdx.x&1)==0)/=((threadIdx.x&256)==0))){
    // Coalesce: The following variables are not warp constant: [( ^  threadIdx.x 1 )]
    ((int*)sbase)[threadIdx.x] =
        min(((int*)(sbase+0x800))[threadIdx.x]
        ,((int*)(sbase+0x800))[(threadIdx.x^1)]);

}
// Diverging: This condition causes divergence issues
if ((!(((threadIdx.x&1)==0)/=((threadIdx.x&256)==0)))){
    // Coalesce: The following variables are not warp constant: [( ^  threadIdx.x 1 )]
    ((int*)sbase)[threadIdx.x] =
        max(((int*)(sbase+0x800))[threadIdx.x]
        ,((int*)(sbase+0x800))[(threadIdx.x^1)]);
}
```

Figure 2.33: Last iteration of the last merge phase of the bitonic sorter

2.2 Code transformation

The previous section was concerned with helping the programmer making decisions about code. This section focuses on making some of these decisions automatically or with less input from the programmer. This should have the result of making it simpler to write faster programs. The Obsidian DSL (both the `Program` data type and the array library) has an inherent structure, which forces the programmer to structure their code. In this section a new DSL is introduced that makes it possible to write code without structure. Care must be taken to not decrease the expressiveness of the DSL when changing it.

Most of this work is done by using code transformations that use some of the analysis described in the previous section. The relationships between all of these processes is quite complex, and is illustrated in Figure 2.1.

2.2.1 DSL deep embedding

```
data LoopLocation = Unknown | Kernel | Grid | Block | Thread | Vector
data LoopType = Seq | Par
type Volumes = [(LoopType,LoopLocation,Word32)]

data Program a where
  --Combiners
  For :: LoopType
      -> LoopLocation
      -> EWord32
      -> (EWord32 -> Program ())
      -> Program ()

  Cond     :: Exp Bool -> Program () -> Program ()
  SeqWhile :: Exp Bool -> Program () -> Program ()
  ParBind  :: Program a -> Program b -> Program (a,b)
  Return   :: a -> Program a
  Bind     :: Program a -> (a -> Program b) -> Program b

  --Statements
  Assign :: Scalar a
         => Name
         -> [Exp Word32]
         -> (Exp a)
         -> Program ()

  AtomicOp :: Scalar a
           => Name
           -> Exp Word32
           -> Atomic a
           -> Program (Exp a)

  Identifier :: Program Identifier
  Break      :: Program ()
  Allocate   :: Name -> Word32 -> Type -> Program ()
  Declare    :: Name -> Type -> Program ()
  Output     :: Type -> EWord32 -> Program Name
  Comment    :: String -> Program ()
```

Figure 2.34: The new DSL for Obsidian

The new `Program` data type is shown in Figure 2.34. The biggest change made is that the type does not contain a level parameter. This simplifies a lot of the code, merging all kinds of for loops into one constructor named `For`. The `For` constructor has been enhanced with information about where the loop should be executed.

The `Volumes` type is designed to be an efficient way of representing a strategy of structuring code for multiply nested architectures. It does this by trying to distribute the iteration volume of the program to the iteration volume of the architecture, which I will refer to as the former *using up* the latter. What the data structure specifies is in which order the iteration spaces should be used. There are two places that this information is used and it works similarly for both places. The first is in `preferredFor` described below in this section, the second is as a parameter to the code generation in which it is used as a way of compiling programs which have no structure specified. These two methods complement each other in two use-cases of these code transformations. When the programmer knows exactly which structure the program should have he can use `preferredFor` or directly specify a strategy for the whole program. When the programmer does not know which structure the code should have, a default structure is generated using the default strategy.

Figure 2.35 shows the default strategy. The way it works is by telling the code transformation to first use one parallel block computation (running zero blocks results in the empty program), and then use at least

one warp. If more volume is needed, it will use more blocks so that at least each SM has something to run (this should vary by processor, but 32 is good maximum). Next use more threads, up to 32 warps yielding 1024 threads, which is the maximum on most GPUs. Next use up to four vector operations per thread. After that use up to the maximum 65536 ($= 2048 \times 32$) number of blocks. If even more volume is needed, then sequential loops at thread level should be used.

```
defaultStrategy =
  [(Par,Block,1)
  ,(Par,Thread,32)
  ,(Par,Block,32)
  ,(Par,Thread,32)
  ,(Par,Vector,4)
  ,(Par,Block,2048)
  ,(Seq,Thread,0)
  ]
```

Figure 2.35: The default strategy used to distribute iteration volume

To improve this default strategy we do an intermediate step to it. Before the program is compiled into IR, the maximal volume of the program is estimated and that volume then tries to use up the default strategy. What has been used up is then sorted by level and combined into a new strategy which will then becomes the default strategy.

```
preferredFor :: Volumes -> EWord32 -> (EWord32 -> Program ()) -> Program ()
```

Figure 2.36: The type of the preferredFor function

The `preferredFor` function shown in Figure 2.36 is equivalent to `For` semantically, but it tries to split the loop into multiple loops in the way specified by the `Volumes` parameter. Furthermore, it does this splitting by placing loops operating at the Thread level innermost. If the original algorithm only does accesses in a linear fashion (for example if its accesses look like `a[x+b]`, where `x` is the induction variable, and `b` is (block) constant), then the generated code's accesses will always be coalesced using this scheme (since `a[threadIdx.x+n*blockIdx.x+b]` is coalesced).

2.2.2 Intermediate representation

To make it easier to perform some analyses, the `Program` data type is compiled into an IR using the `compile` function. This IR is similar to the one described in Section 2.1.1. The biggest change is in procedure is that the `Volumes` parameter of `For` must be converted into one or more `SFor` s. The goal of this is to have a structure with nested for loops that may not necessarily have the correct order. In the next section loop transformations are described that fix this and which try to make the order correct.

This process works by compiling `Program` into an IR as previously described, but when a `For` is encountered it is processed as following: If it has a specified strategy, let that strategy use up the default strategy (removing volumes of the same level and type from the default strategy). Otherwise it tries to split the loop into multiple loops using `preferredFor` which uses up the default strategy (using the default strategy to create an iteration volume, while removing volume from it).

2.2.3 Loop transformations

One important problem that must be solved with the new structure is converting the volumes of iteration space of the program into the volumes of iteration space as specified by the strategy. This is done by dividing the volume into parts, then reordering these volumes by type and finally volumes of the same type.

The `preferredFor` function performs the first task by doing a transformation called loop tiling, loop blocking if done in two dimensions, or strip-mining (also called loop-sectioning) if done for vectorization purposes. This splits the loop into components, but does not do the necessary reordering and merging. For that we need two other transformations: Loop interchange (also called loop permutation) and loop merging.

Loop interchange tries to reorder the loops into the order specified by the `LoopLocation` parameter to the `SFor` constructor. Loop interchange is only possible when it is safe to interchange the loops. The safety condition is that all dependences in the loop should have their direction preserved through the transformation. With parallel loops this is always the case, and moving a parallel loop into another loop is also always possible. The last cases are when the outermost loop is a sequential loop. In that case we have to fall back on dependence analysis to determine interchange order.

Loop merging is the simple process of taking loops that should be executed in the same way and merging them into one loop. An example to illustrate this transformation is shown in Figure 2.37.

```
for(i = 0..2)
  for(j = 0..256)
    f(i,j);

for(t = 0..512)
  i=t/256
  j=t%256
  f(i,j);
```

Figure 2.37: Showing code of before and after the loop merging transformation is done

```
//original code
for(x = 0..512) -- (Block,16,Par)
  for(y = 0..32)
    f(x,y);

//after loop tiling
for(x_block = 0..16)
for(x_thread = 0..32)
  for(y_block = 0..4)
  for(y_thread = 0..8)
    f(x_thread+32*x_block,y_thread+8*y_block);

//after loop interchange
for(x_block = 0..16)
for(y_block = 0..4)
  for(x_thread = 0..32)
  for(y_thread = 0..8)
    f(x_thread+32*x_block,y_thread+8*y_block);

//after loop merging
for(block = 0..64)
  x_block = block%16;
  y_block = block/16;
  for(thread = 0..256)
    x_thread = thread%32;
    y_thread = thread/32;
    f(x_thread+32*x_block,y_thread+8*y_block);
```

Figure 2.38: Showing the code after each loop transformation

An example of how the code looks after all these transformations is shown in Figure 2.38. The last form is exactly the structure needed to generate GPU code. Notice that `x_thread` and `y_thread` are placed innermost. If `f` only performs accesses in a linear fashion, then those accesses will be coalesced.

Another loop transformation is loop unrolling that simply duplicates the body of a loop the number of times the loop should be executed. This should only be done when the body is simple and the number of loop iterations is small, to prohibit code growth. The criterion used here is to only unroll loops smaller than a specified parameter (with four as the default). This loop unrolling enables otherwise not possible optimizations, described in the next sections.

Loop unrolling is also the way vectorization of loops and latency hiding is handled. If the compiler of the generated code is able to reorder a list of instructions (that is, the output of an unrolled loop) into an optimal vector form, then this will be sufficient for vectorization. If the specific GPU does not support vectors, this process will make it easier for the compiler to schedule instructions for hiding the latency of operations and thus increased performance.

2.2.4 Other transformations

Some cleanup transformations are also performed, such as replacing variables that have simple values with their respective value and removing declaration of variables and arrays that are never used. Figure 2.39 shows the stages of this transformation.

```
//after loop merging
for(block = 0..64)
  int x_block;
  x_block = block%16;
  int y_block;
  y_block = block/16;
  for(thread = 0..256)
    int x_thread;
    x_thread = thread%32;
    int y_thread;
    y_thread = thread/32;
    f(x_thread+32*x_block,y_thread+8*y_block);

//after assignment replacement
for(block = 0..64)
  int x_block;
  int y_block;
  for(thread = 0..256)
    int x_thread;
    int y_thread;
    f((thread%32)+32*(block%16),(thread/32)+8*(block/16));

//after declaration removal
for(block = 0..64)
  for(thread = 0..256)
    f((thread%32)+32*(block%16),(thread/32)+8*(block/16));
```

Figure 2.39: Illustrating the cleanup transformations done

Synchronizations are inserted between every block-level sequential statement and loop that can introduce a dependence (i.e. excluding declarations). Using the analysis described previously to determine if a synchronization is unnecessary is very beneficial at this step. Figure 2.40 shows an example of this transformation working.

```
//before inserting synchronizations
for(block = 0..64)
  for(thread = 0..256)
    f((thread%32)+32*(block%16),(thread/32)+8*(block/16));
  for(thread = 0..256)
    g((thread%32)+32*(block%16),(thread/32)+8*(block/16));

//after synchronization insertion
for(block = 0..64)
  for(thread = 0..256)
    f((thread%32)+32*(block%16),(thread/32)+8*(block/16));
  synchronize()
  for(thread = 0..256)
    g((thread%32)+32*(block%16),(thread/32)+8*(block/16));
  //no synchronization needed here
```

Figure 2.40: Illustrating the result of the synchronization insertion transformation

2.2.5 Scalar depromotion

This section describes an optimization that removes many extraneous memory accesses through *Scalar depromotion*. *Scalar promotion* is the process of transforming a loop local variable into an array outside of the loop to enhance dependence analysis. But since memory accesses are slower than register accesses, the reverse transformation is advantageous. Figure 2.41 shows typical code for the base case of a recursively defined function. The inefficiency of an extra memory access is often solved by using a more

complex base case, but with scalar depromotion the code is automatically transformed into the desired result.

```
//before scalar depromotion
for(block = 0..64)
  for(thread = 0..256)
    shared[thread] = f(thread);
  for(thread = 0..256)
    output[block*256+thread] = shared[thread];

//after scalar depromotion
for(block = 0..64)
  int ts;
  for(thread = 0..256)
    ts = f(thread);
  for(thread = 0..256)
    output[block*256+thread] = ts;
```

Figure 2.41: Showing the result of the Scalar depromotion transformation

This transformation is only possible when a thread reads from the same location as the same thread wrote to. The write to the memory location can be removed (as in the example in Figure 2.41) if all reads are done from the same thread. Furthermore, since only a single variable is inserted (unless loop unrolling is used to insert more variables), the transformation will not work if more than one value is needed.

2.2.6 Reduction example

Compare `red13` in Figure 2.42 with `red5` in Figure 2.12. `red13` is similar to the rather naive version of reduction `red2`, but a strategy is specified.

```
red13 :: MemoryOps a
    => (a -> a -> a)
    -> SPull a
    -> Program (SPush a)
red13 f arr
    | len arr == 1 = return $ push arr
    | otherwise    = do
        let (a1,a2) = halve arr
        arr' <- force $ zipWith f a1 a2
        red13 f arr'
```

Figure 2.42: A simpler implementation of reduction

The difference between `red5` and `red2` is that `red5` has the following optimizations:

- A better base case at two elements that does not force an array with a single variable.
- The data is split up into pieces and sequential reduction is performed on those pieces.

When generating the code for `red13` with all the transformations described above performed, we get a program that looks more similar to `red5`. Scalar depromotion makes the optimization of the base case unnecessary. The code generated for `red13` that is relevant for the second optimization is shown in Figure 2.43. To get higher performance, `red5` uses sequential reduction, an algorithm that is not the same algorithm as parallel reduction. But if both algorithms are unrolled, the resulting codes are very similar.

The performance should be equivalent. Thus the generated code from `red13` is almost equivalent to an unrolled sequential reduction, but without the need to specify that algorithm.

```
ts0 = (input0[((blockIdx.x*1024)+threadIdx.x)]+input0[((blockIdx.x*1024)+(threadIdx.x+512))]);
ts1 = (input0[((blockIdx.x*1024)+(128+threadIdx.x))]+input0[((blockIdx.x*1024)+((128+threadIdx.x)+512))]);
ts2 = (input0[((blockIdx.x*1024)+(256+threadIdx.x))]+input0[((blockIdx.x*1024)+((256+threadIdx.x)+512))]);
ts3 = (input0[((blockIdx.x*1024)+(384+threadIdx.x))]+input0[((blockIdx.x*1024)+((384+threadIdx.x)+512))]);
ts5 = (ts0+ts2);
ts6 = (ts1+ts3);
ts8 = (ts5+ts6);
((int*)(sbase+6144))[threadIdx.x] = ts8;
__syncthreads();
if ((threadIdx.x<64)){
    ts10 = (ts8+((int*)(sbase+6144))[(threadIdx.x+64)]);
    ((int*)(sbase+7168))[threadIdx.x] = ts10;

}
```

Figure 2.43: Output of start of red13

2.2.7 Matrix multiplication

To illustrate that Obsidian works for algorithms operating on something other than lists this section demonstrates an example program operating on matrices. A basic operation for linear algebra is matrix multiplication. Matrix multiplication can be defined as, for each element, take the dot product of its row in one matrix with its column in another matrix. This translates quite naturally into the code shown in Figure 2.44. The dot product is calculated sequentially which simplifies the code considerably.

```
matMul a b = pConcatMap (matMulRow (transpose b)) a
  where matMulRow mat row = return $ pConcatMap (dotP row) mat
        dotP a b = return
                 $ seqReduce (+)
                 $ zipWith (*) a b
```

Figure 2.44: Matrix multiplication kernel

3

Evaluation

3.1 Benchmarks

The benchmarks in this section were all run using the CUDA interface for Haskell (specifically the `cuda` package version 0.5.0.3). Similarly, the n-body simulation uses the OpenGL and GLUT interface (packages `opengl` version 2.8.0.0 and `GLUT` version 2.4.0.0). This means that the kernels were run with the Haskell runtime system which uses garbage collection for resources, which may have slowed some benchmarks down. The kernels were compiled with `nvcc` NVIDIA's CUDA compiler, release 5.0 and run on a GeForce GTX 480. Each kernel was run 1000 times and the average was extracted from this.

3.1.1 Reduction

First, we must determine what we should measure. The standard technique of running a reduction kernel on parallel processors is to divide the problem into subproblems that can be reduced fast, and then reduce the result of those. This last reduction of the results often takes negligible time and will not be considered here, which means we can focus on making a single kernel very fast. Since the problem is divided into subproblems, the size of the subproblems must be known, and this becomes a tuning parameter. Thus we should measure the performance for different values for this tuning parameter to find the fastest version. Now, we obviously want to measure the running time of the program, but that measurement will only tell us how fast that particular problem is. A more useful metric is time taken per element which, if the program scales well, will be a consistent measurement for the program. This is actually the inverse of the bandwidth (elements processed per second) of the program. Well, does our program scale? As can be seen in Figure 3.1 the program scales with a coefficient below one (sub-linear scaling, which can be attributed to a non-proportional initialization time the program must do). A conclusion that can be made from Figure 3.1 is that a blocksize of 128 is best.

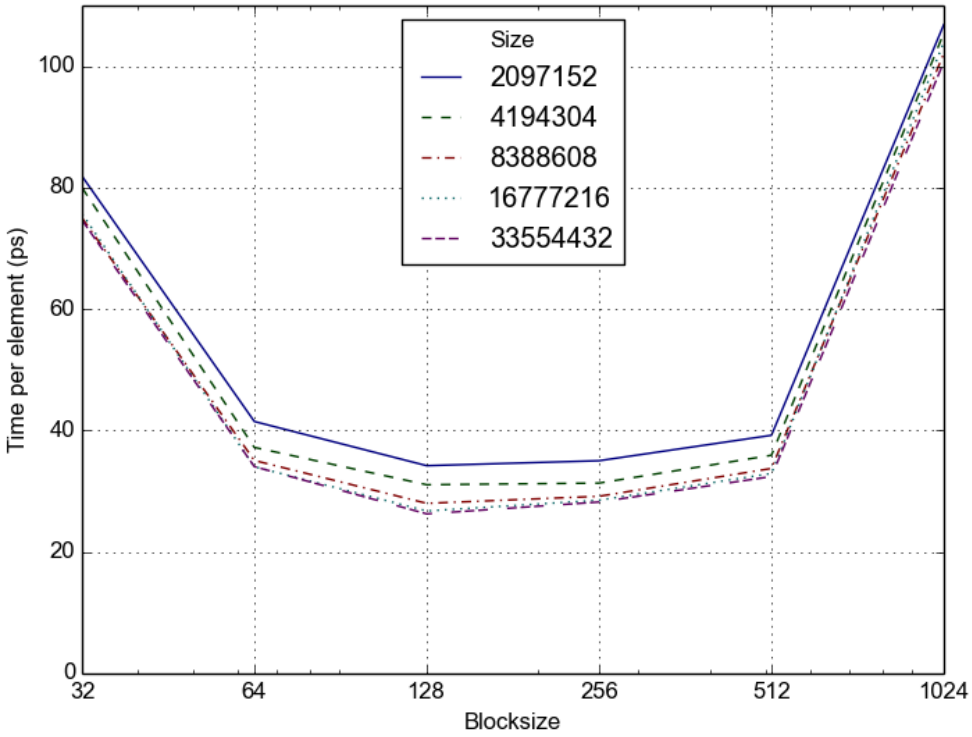


Figure 3.1: Reduction using red3 with different problem sizes

In Section 2.1.10 the optimization of a reduction kernel was explored using programs `red1` through `red5`, and in Section 2.1.10 the compiler did those optimizations for us in `red13`. The result is shown in Figure 3.2. `red5-opt` and `red13-opt` are their respective programs run with all smaller optimization turned on. There are clearly three different classes of programs. The first contains the naive `red1`, the coalesced `red2`, and `red3` which has a better base case. All of these process only two elements per thread. The next and faster class contains `red4` which processes eight elements per thread, but reads them in an uncoalesced order. `red5` fixes this problem and defines the last class. `red13`, which is a naive implementation that is using the code transformations is also in this fastest class. The optimization transformations does very little for performance, but `red13-opt` is slightly faster for the smaller subproblems.

This demonstrates the usefulness of this thesis. The analyses performed help the user to optimize `red1` to `red5` achieving the fastest program class. The transformations makes these optimizations automatic and gives you `red13`, also achieving the fastest program class.

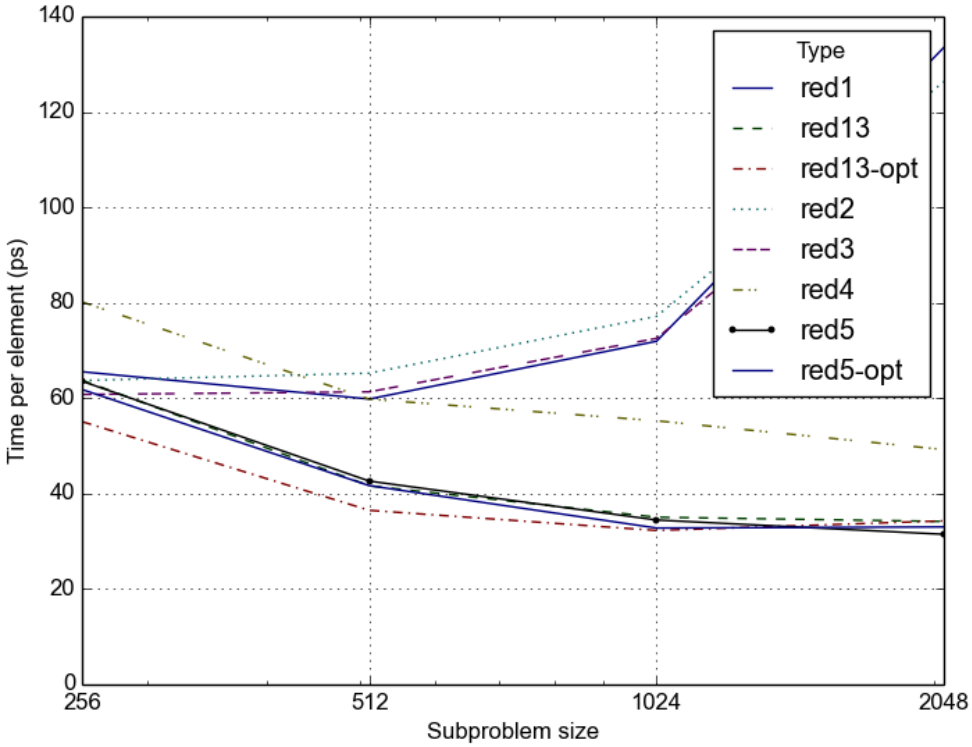


Figure 3.2: Different reduction programs

Figure 3.2 showed that the optimizations did little to make a faster program. Which transformations are affecting the runtime the most? Figure 3.3 shows just this. As can be seen, most optimizations have almost no effect on the performance. The Scalar depromotion optimization does improve performance for kernels with a blocksize larger than 256, and makes the kernel with blocksize 64 slower.

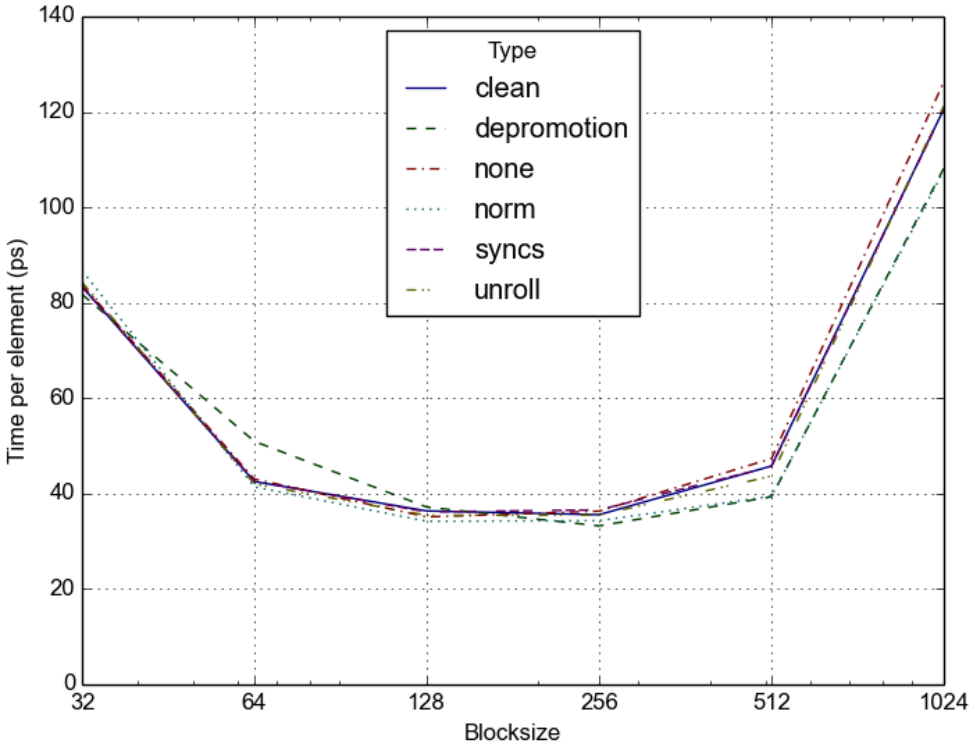


Figure 3.3: Reduction with different optimizations

In Figure 3.1 the performance of the kernel translates to a bandwidth of 128.7 GB/s. Vucud & Harris [2012] achieves a speed of 80.4 GB/s for the same problem size and using manual optimizations on an NVIDIA m2090, which has similar memory bandwidth as the GPU used for testing in this thesis. These manual optimizations are equivalent to the optimizations performed in *red*1 to *red*5.

3.1.2 Scan

In Section 2.1.11 two solutions to the scan problem where developed. `sklansky1` through `sklansky3` shown in Figure 2.17 tries to do a scan based on Sklansky [1960]. `scan1` shown in Figure 2.19 on the other hand uses the inplace library, which enables efficient implementation of the scan algorithm described by Brent & Kung [1982], to perform a lot less work ($O(n \log n)$ vs $O(n)$), at the cost of twice the depth (both have $O(\log n)$ depth). Figure 3.4 shows that this optimization actually causes a very big performance loss. The `sklansky` kernels scan two elements per thread, while `scan1` only scans one per thread. If `scan1` is enhanced by doing twice the amount of work per thread (using a better strategy) it becomes `scan1-2` in Figure 3.4. This makes the performance gap smaller, but it does not eliminate it. It can be concluded from this that, in this case, it is better to minimize the depth of a kernel than to minimize the work performed. This result can be generalized to programs with regular work load are often better for GPUs than programs with irregular workload. This is also why bitonic sort (see Section 2.1.13) is an often used sorting algorithm for GPUs.

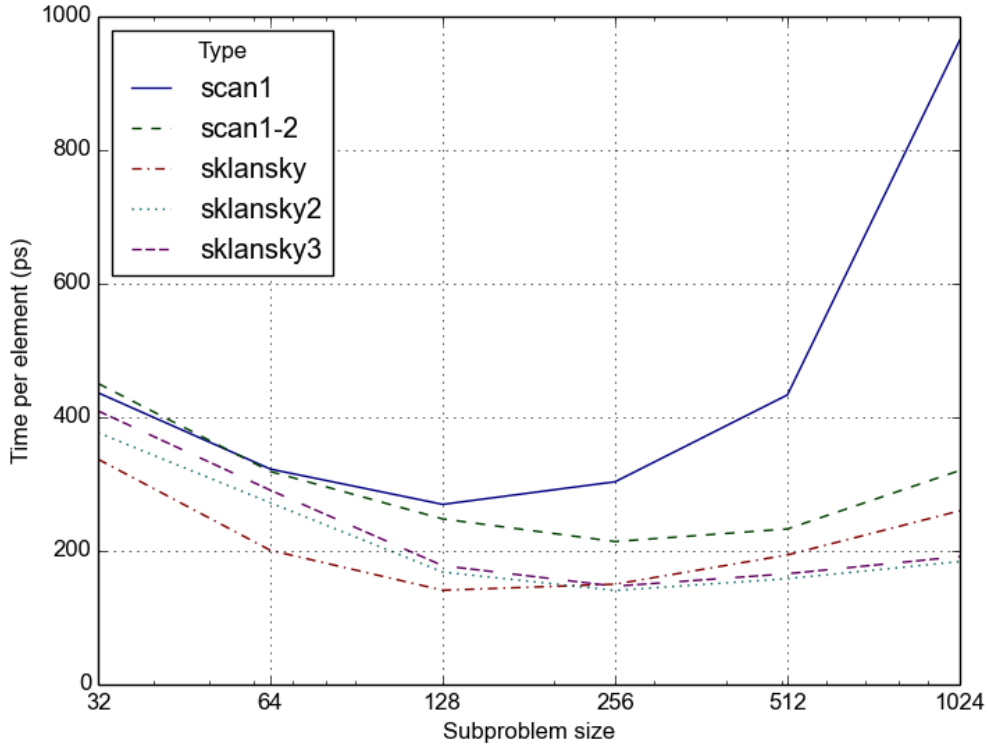


Figure 3.4: Different scan programs

3.1.3 SAXPY

Section 2.1.12 shows the implementation of SAXPY kernels performing the element-wise operation `y[i] += a*x[i]`. Figure 3.6 shows the runtime of those kernels. Section 2.1.12 describes the process of implementing a hand-optimized kernel that does this operation. `saxpy0` is the naive kernel, which is improperly improved upon in `saxpy1` and `saxpy2` by doing eight elements per thread, but does so in an uncoalesced manner. `saxpy3` and `saxpy4` improve upon this by making the accesses coalesced.

By using the code transformations introduced in Section 2.2, two new kernels can be created, as shown in Figure 3.5. In `saxpy0` through `saxpy4` the problem is manually split into subproblems, which `saxpy5` does not do. It is expressed as a simple map over the elements, but has a specified manual strategy that performs loop tiling that does the same thing as `saxpy4`. `saxpy6` instead uses the default strategy, which does similar transforms.

```
saxpy5 :: (Num a, ASize l)
       => Word32 -> a -> Pull l a -> Pull l a -> Push l a
saxpy5 bs a x y = pushA [(Par,Thread,bs),(Seq,Thread,8),(Par,Block,0)]
                $ fmap (\(x,y) -> y+a*x) $ zipp (x,y)

saxpy6 :: (Num a, ASize l)
       => Word32 -> a -> Pull l a -> Pull l a -> Push l a
saxpy6 bs a x y = push $ fmap (\(x,y) -> y+a*x) $ zipp (x,y)
```

Figure 3.5: saxpy5 and saxpy6 kernels

Figure 3.6 shows that all the kernels are fast, except the improperly optimized `saxpy1` and `saxpy2`.

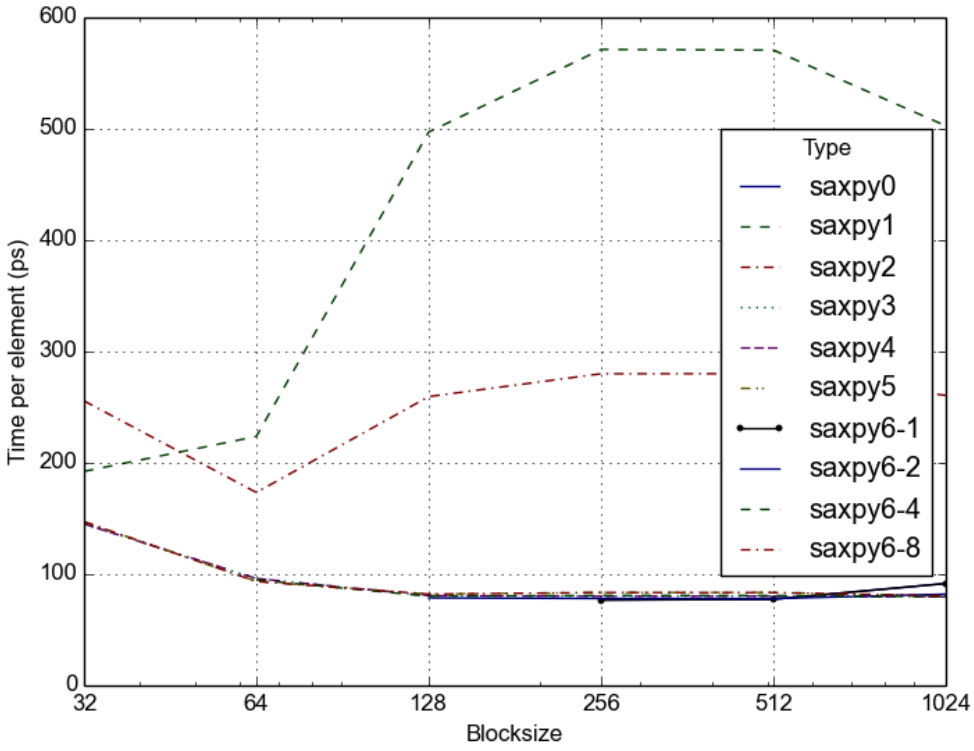


Figure 3.6: Different SAXPY programs

3.1.4 Bitonic sort

Bitonic sort is a non-linear algorithm: it has work $O(n \log^2 n)$ and depth $O(\log^2 n)$, and thus time per element cannot be compared. Instead, we compare total running time when sorting 1M elements divided into chunks of size 512. Figure 3.7 shows the results and compares the result of the kernel described in Section 2.1.13 (Figure 2.32) to the handwritten kernel shown in Figure 3.8 described by Claessen et al. [2012]. Even though the kernel described in Section 2.1.13 is not optimized in any way, it produces a faster kernel than the handwritten one, which may not either have been optimized.

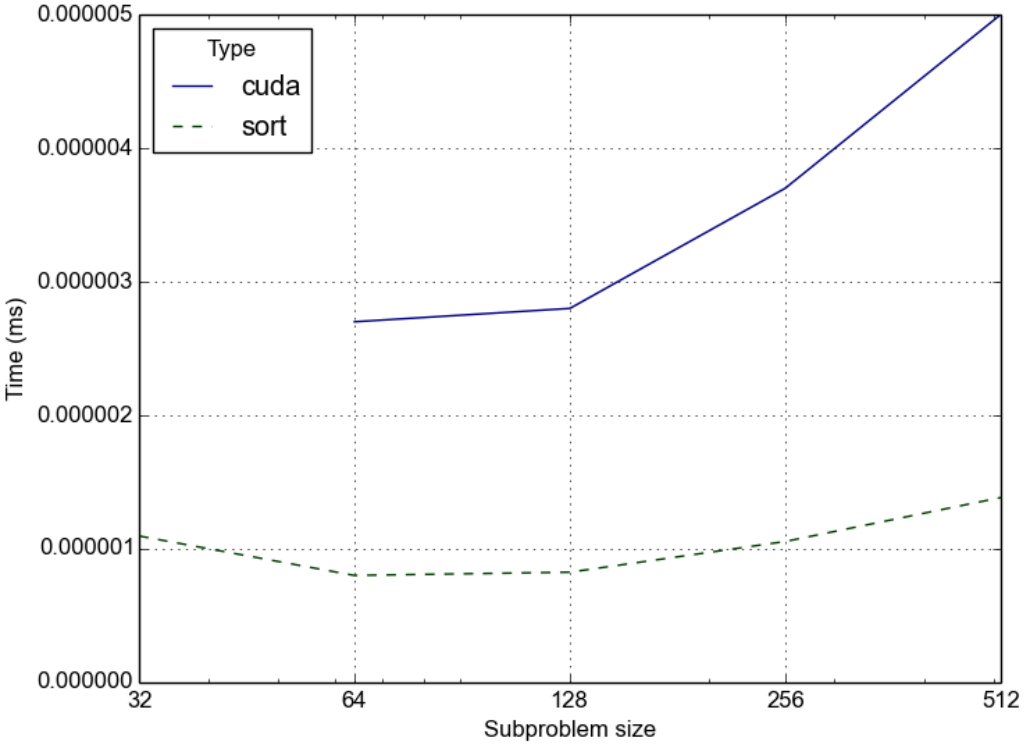


Figure 3.7: Different bitonic sort programs. cuda is the handwritten kernel presented below and sort is the kernel described in Section 2.1.15.

```
__device__ inline void swap(int & a, int & b)
{
  int tmp = a;
  a = b;
  b = tmp;
}

__global__ static void bitonicSort(int * values, int *results)
{
  extern __shared__ int shared[];

  const unsigned int tid = threadIdx.x;
  const unsigned int bid = blockIdx.x;

  // Copy input to shared mem.
  shared[tid] = values[(bid*NUM) + tid];

  __syncthreads();

  // Parallel bitonic sort.
  for (unsigned int k = 2; k <= NUM; k *= 2)
  {
    // bitonic merge
    for (unsigned int j = k / 2; j>0; j /= 2)
    {
      unsigned int ixj = tid ^ j;
      if (ixj > tid)
      {
        if ((tid & k) == 0)
        {
          if (shared[tid] > shared[ixj])
          {
            swap(shared[tid], shared[ixj]);
          }
        }
        else
        {
          if (shared[tid] < shared[ixj])
          {
            swap(shared[tid], shared[ixj]);
          }
        }
      }
    __syncthreads();
    }
  }

  // Write result.
  results[(bid*NUM) + tid] = shared[tid];
}
```

Figure 3.8: Handwritten bitonic sort kernel

3.1.5 n-body

The n-body problem is the problem of simulating n bodies that all interact with each other through a field, such as the gravitation field or the electric field. This has work $O(n^2)$ since each element must calculate the interaction with every other element. The minimal depth is $O(\log n)$ (summing n elements), but the implementation shown in Figure 3.9 has depth $O(n)$. This enables each thread to process a single element, and since the work is quadratic, creating n threads, enough parallelism will be created anyway. `calcA` calculates the interaction force between two elements, and `updatePos` takes the sum of all interactions and calculates the updated position. Thus the job of `nbody1` is to read in all elements, and calculate the sum of the interaction force between them, feeding that to `updatePos`. A smart way of doing this is letting the threads cooperate by having each thread in a block load an element and then

share it with the other threads. `bs` (the blocksize) times fewer elements per thread are thus loaded from the slower global memory. The interaction of these elements is then calculated and then the next batch of elements is processed by the block. This pattern requires the `seqFoldA` function which performs a sequential reduction, using an array as an accumulator (the ordinary `seqFold` can only use a scalar as an accumulator).

```
type Pos = (EFloat,EFloat,EFloat,EFloat)
type Pos3 = (EFloat,EFloat,EFloat)

calcA :: Pos -> Pos -> Pos3 -> Program Pos3
calcA pi pj ai = do
  let bi = getPos3 pi
      bj = getPos3 pj
  r <- scalarForce $ op (-) bj bi
  dist <- scalarForce $ eps + abs r
  let invDistCube = 1 / (sqrt $ dist * dist * dist)
  s <- scalarForce $ invDistCube * getW pj
  return $ op (+) ai $ sop (*s) r

updatePos :: EFloat -> Pos3 -> (EFloat,Pos3,Pos3) -> Program (Pos3,Pos3)
updatePos t a (w,p,v) = do
  tw <- scalarForce $ t*w
  v' <- scalarForce $ op (+) (sop (tw*) a) v
  p' <- scalarForce $ op (+) p v'
  return (p',v')

nbody1 :: EFloat -> Word32 -> Word32 -> SPull (EFloat,Pos3,Pos3) -> SPush (Pos3,Pos3)
nbody1 t bs ur all =  pSplitMap bs f all
  where
    n=sizeConv $ len all
    f :: SPull (EFloat,Pos3,Pos3) -> SPush (Pos3,Pos3)
    f ball = pJoin $ do
      let barr :: SPull Pos
          barr = fmap (\(w,(x,y,z),_) -> (x,y,z,w)) ball
          arr :: SPull Pos
          arr = fmap (\(w,(x,y,z),_) -> (x,y,z,w)) all
      ais <- seqFoldA (push $ replicate bs zeroPos3)
                      (splitUp bs arr)
                      (g barr)
      return $ pConcat $ fmap (singletonP . (uncurry $ updatePos t)) $ zip ais ball
    g :: SPull Pos -> SPull Pos3 -> SPull Pos -> SPush Pos3
    g barr ai carr = pJoin $ do
      sh <- force carr
      let sh' = splitUp ur sh
      return $ pConcatMap (singletonP . h sh') $ zip ai barr
    h :: SPull (SPull Pos) -> (Pos3,Pos) -> Program Pos3
    h sh (ai,b) = do
      seqFoldP ai sh $ \ai' cs -> do
        foldM (\ai'' i -> calcA b (cs ! fromIntegral i) ai'') ai' [0..len cs-1]

seqFoldA :: (ASize l, MemoryOps a)
        => SPush a
        -> Pull l b
        -> (SPull a -> b -> SPush a)
        -> Program (SPull a)
```

Figure 3.9: nbody kernel

Using this kernel there are now multiple ways the kernel can be run. Figure 3.10 shows the result of different ways of running the kernel. `normal` is the naive way of running the kernel, using an array for each type of data read and written to the kernel, and then reading the position data back to the CPU using OpenGL to draw the points. This round trip to the CPU is very slow, but this could be removed by using CUDA-OpenGL interoperability functions (which however does not currently exist in the current CUDA library). This interoperability works by reserving some GPU memory that the CUDA runtime can pin

and use, then unpin and let OpenGL use. To have OpenGL use this data to draw with requires it to have the correct format. The required format is to have one array with the coordinates interleaved, i.e. having an array of vectors of the coordinates. These arrays of vectors can be used for all data, having one vector with the coordinates and the weight of the particle, and having another for the velocity vector (`Pos` and `Pos3` respectively). This kernel is called `vector`.

The memory access pattern created when using arrays of vectors is not that good for GPGPU kernels since the kernels will not be coalesced. So if we go back to using one array for each type of data, and then run another kernel that reorganizes the data, we may achieve higher performance. This is done with `interop` and gives a small increase in performance. Turning on all optimizations gives us `interop-opt`, which is as fast. If we forgo displaying the data at all as in `fast`, the performance increases slightly.

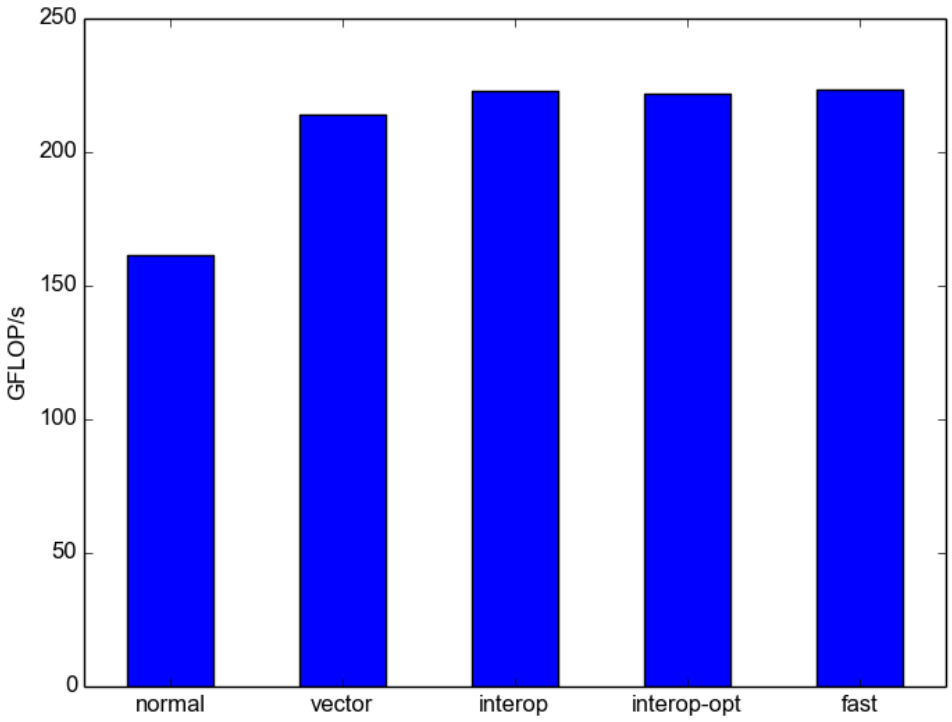


Figure 3.10: Different nbody programs

4

Discussion

When comparing kernels the gold standard is run time measurements. This will only give the information of how fast a kernel is, and not which parts can be improved. Big bottlenecks are the obvious thing to improve, and while the method of finding bottlenecks described in Section 1.1.1 is the best way to determine if something is a bottleneck, finding what may be a bottleneck in the first place is another story. The cost analysis described in Sections 2.1.5 and 2.1.6 should be a good way to finding possible bottlenecks. The memory access pattern analysis and divergence issues detection are also good for finding parts of the code that are slower than they need to be. Changing the memory access pattern and which threads are executing can be accomplished by changing which threads are doing which task. This transformation can actually be done automatically if an access is written as $f(tid)$, where tid is the thread number, and f^{-1} can be found. Then use $tid' = f^{-1}(tid)$ as the new thread number, which makes the access linear because $f(tid') = f(f^{-1}(tid)) = tid$ (assuming f is the only access pattern). For example, if $f(tid) = 32(tid \mod 32) + tid/32$ (imagine transposing square matrices of size 32), then setting $tid' = f^{-1}(tid) = 32(tid \mod 32) + tid/32$ (transpose being its own inverse), will yield $f(tid') = tid$.

NVIDIA's visual profiler can be seen as a complement to the static analysis performed. It runs the program to determine how long time it took, what took time and what issues it found with the kernel such as uncoalesced global memory accesses and divergent branch detection. These last issues can also be determined with the static analysis described in Sections 2.1.6 and 2.1.7. When testing some of the issues of the reduction kernels, NVIDIA's visual profiler could not find any of the issues that the static analysis of this thesis found.

The methods described in this thesis (mainly the structural code transformation, the scalar depromotion, and the static analysis feedback), are not specific to functional programming since the new `Program` data type is imperative. This should make it possible to use these techniques in other compilers, such as the CUDA compiler. Since the dependence analysis and transformations are standard concepts in compilers, they may already be present in the CUDA compiler, which should make implementing the methods described in this thesis simpler.

An interesting question is why `saxpy0` in Section 2.1.12 is as fast as the optimized versions, when it should be slower. The answer may be that the CUDA compiler performs the same optimization by itself. There are compilers for GPUs capable of doing vectorization (loop tiling) [Rotem, 2011] automatically. This optimization does usually work only for simple kernels (as `saxpy0` is). The CUDA compiler is

not able to vectorize more complicated kernels, as demonstrated with `scan1` in Section 2.1.11. Another, simpler explanation is that the kernel is memory bound and the overhead of starting many kernels is negligible.

Sections 2.2.3 and 2.2.4 describe how synchronizations are inserted into the kernels: first synchronization points are inserted everywhere and then the unnecessary ones are removed. This is a completely automatic process and it is a conservative and safe operation, that removes all race conditions within thread groups, which are the cause of many hard-to-find bugs in many applications, not only GPU programming. Race conditions in general (missing synchronizations cause race conditions) are a problem for many programs, which the race condition analysis helps the user with. One problem with this method is that performance may be diminished if the analysis is imprecise.

The n-body example described in Section 3.1.5 shows us that it is possible to write complicated kernels using Obsidian; the rest of this thesis has focused on simple kernels. The example also illustrates the advantage of Obsidian over pure CUDA (or functional programming approaches over imperative approaches), because it enables changing the memory layout (which is very important for performance) without any changes to the core kernel function, and thus promoting composability. Since it is easier to write kernels it is also easier to run multiple kernels, which can often be beneficial for performance.

Another important benefit of a strong type system is that some guarantees of running the kernels can be made (not shown in this thesis). For example, OpenGL-CUDA interop requires a lock on the memory so that OpenGL and CUDA will not change the same memory. This lock can be quite easily realized with `withMappedResources` shown in Figure 4.1. This function is the only way to create a `CUDAVector` from an OpenGL resource, and if no OpenGL functions can be run in the `CUDA` monad then it is impossible to violate this lock by virtue of the type system.

```
withMappedResources :: [CUGL.Resource] -> ([CUDAVector a] -> CUDA b) -> CUDA b
useGLVectors :: Storable a
             => [Vector a]
             -> ([(CUGL.Resource, GL.BufferObject)] -> CUDA b)
             -> CUDA b
```

Figure 4.1: The useGLVectors function. Vector is a host side array. CUDAVector is a CUDA array. CUGL.Resource is represents an OpenGL BufferObject. CUDA is the monad in which CUDA programs are run.

4.1 Future work

The dependency tests described in Sections 2.1.2 and 2.1.3 are conservative and safe, but not exact. Since the problem is NP-complete, exactness may not be desirable since the analysis would take too long. The tests performed in this thesis are the easiest and fastest (with the exception of the introduced bit-test). There are other tests that are only a little slower but more exact such as the Omega test [Pugh, 1991]. Using some of these tests for increased precision could be an interesting extension to the work presented in this thesis.

There are other cases where better analysis could increase the precision of the analysis. One is divergent branch detection described in Section 2.1.7. This could be quite naively enhanced by simply enumerating all threads per group (which are at most 1024) and determine if only some threads per warp are executing.

Larsen [2011] describes a technique for caching global memory reads in shared memory automatically. This optimization could be implemented in the framework described in this thesis to increase performance

and should be quite easy to do. Similarly, Hong & Kim [2009] describe a technique for estimating runtime of kernels accurately, which could extend the analysis performed in this thesis and would complement the information provided.

The transformations described in Sections 2.2.3 and 2.2.4 does not always do what the programmer expects. If specifying an impossible strategy, (for example requiring the program to change an access ordering), the program will sometimes fail to compile, which is better than it compiling an incorrect program, but not as good as what happens the other times, which is using some heuristics that uses the strategy differently to construct a correct program. The error message generated in this case is often not related to transformation, but instead related to incorrectly structured programs. One further problem that this creates is that it is impossible to specify a default strategy that will work for all programs, which then requires a manual strategy to be specified to make the program correct. With better heuristics this problem may be alleviated. Another approach may be to have a check for incorrect program-strategy combinations somewhere in the compilation phase.

The premise of this thesis is the use-case of a user requiring a faster program than those provided by approaches similar to Accelerate's, and being not very knowledgeable of GPGPU programming tools (referred to as a novice in this thesis), such as CUDA. The author of this thesis cannot be considered to be fulfilling the second criterion (by necessity since the work in this thesis requires it), and as such, the embedded language may not be as easy to understand as a novice would want. This could be improved by having novice users use the language and give feedback on it. Another use-case considers a user that wants to be more productive than CUDA allows, which this thesis does enhance, provided none of the problems described are encountered.

Another use-case this thesis enables is using another, higher level language as a frontend for Obsidian, and since the DSL is general and imperative, any frontend outputting such code could potentially be used. Feldspar [Axelsson et al., 2010] is such a frontend that ordinarily generates C code. Feldspar is a DSL designed for parallel computing, mostly with DSP, which are similar to GPUs. Feldspar could benefit from a GPU backend. One of the IRs used by Feldspar is imperative and converting it to the Obsidian IR described in this thesis is possible. A proof of concept version of this has been produced which could be expanded upon to make Obsidian a backend of Feldspar.

5

Conclusion

This thesis has shown extensions to Obsidian that can successfully aid in writing efficient and correct GPU kernels. It is done by providing information through different kinds of analyses, such as out of bounds checking, memory access patterns analysis for determining coalescing and bank conflict issues, divergent branch detection for issues with divergence, race condition detection, and unnecessary synchronization detection.

This thesis has also shown how many of the decisions about optimizing kernels can be made automatically through different code transformations, such as loop tiling, loop interchange, loop merging, loop unrolling, scalar depromotion, and unnecessary synchronization removal. The resulting DSL requires less knowledge of GPU programming, making it easier to write correct and fast programs, while still being able to generate kernels that are as expressive and as fast.

6

Bibliography

Akenine-Moller, T., Moller, T., & Haines, E. (2002). *Real-time rendering*. AK Peters, Ltd.

Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., & Vajdax, A. (2010). Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on* (pp. 169–178).: IEEE.

Banerjee, U. (1976). *Data dependence in ordinary programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., & Sadayappan, P. (2008). Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (pp. 1–10).: ACM.

Blelloch, G. E. (1995). *NESL: A Nested Data-Parallel Language*. Technical Report CMU-CS-95-170, Pittsburgh, PA, USA.

Blelloch, G. E. & Sabot, G. W. (1990). Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2), 119–134.

Brent, R. P. (1974). The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2), 201–206.

Brent, R. P. & Kung, H.-T. (1982). A regular layout for parallel adders. *Computers, IEEE Transactions on*, 100(3), 260–264.

Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., & Grover, V. (2011). Accelerating haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming* (pp. 3–14).: ACM.

Claessen, K., Sheeran, M., & Svensson, B. J. (2012). Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming* (pp. 21–30).: ACM.

Claessen, K., Sheeran, M., & Svensson, J. (2008). Obsidian: GPU programming in Haskell. In *Proc. of 20th International Symposium on the Implementation and Application of Functional Languages (IFL'08)*.

Demmel, J. (2011). Applications of parallel computers. `http://www.cs.berkeley.edu/~demmel/cs267_Spr11/`.

Hong, S. & Kim, H. (2009). An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37 (pp. 152–163).: ACM.

Intel (2013). Desktop 4th generation Intel© Core™ processor family and desktop Intel© Pentium© processor family. `http://ark.intel.com/products/52213`.

Kennedy, K. & Allen, J. R. (2001). *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2009). PyCUDA: GPU run-time code generation for high-performance computing. *Parallel Computing, Elsevier, CoRR, abs/0911.3456*.

Larsen, B. (2011). Simple optimizations for an applicative array language for graphics processors. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming* (pp. 25–34).: ACM.

Mainland, G. & Morrisett, G. (2010). Nikola: embedding compiled GPU functions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, volume 45 (pp. 67–78).: ACM.

McDonell, T. L., Chakravarty, M. M., Keller, G., & Lippmeier, B. (2013). Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13 (pp. 49–60). New York, NY, USA: ACM.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Munshi, A. et al. (2009). The OpenCL specification. *Khronos OpenCL Working Group*, 1, 11–15.

NVIDIA (2010). Nvidia GeForce GTX 580 GPU datasheet. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications`.

NVIDIA (2012). Tesla K20X GPU accelerator board specification. `http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf`.

NVIDIA (2013). CUDA C programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

PCI-SIG (2010). PCI Express© base specification. `http://www.pcisig.com/specifications/pciexpress/specifications`.

Pugh, W. (1991). The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (pp. 4–13).: ACM.

Rotem, N. (2011). Intel opencl sdk vectorizer. In *LLVM Developer Conf. Presentation*.

Sklansky, J. (1960). Conditional-sum addition logic. *Electronic Computers, IRE Transactions on*, (2), 226–231.

Svensson, B. J. (2013). *Embedded Languages for Data-Parallel Programming*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology.

Vucud, R. & Harris, M. (2012). Optimizing parallel reduction in cuda. `http://vuduc.org/teaching/cse6230-hpcta-fa12/slides/cse6230-fa12--05b-reduction-notes.pdf`.

Williams, R. (2010). Intel's Core i7-980X extreme edition. `http://techgage.com/article/intels_core_i7-980x_extreme_edition_-_ready_for_sick_scores/8/`.