Meta-programming and Auto-tuning in the Search for High Performance GPU Code

Michael Vollmer Bo Joel Svensson Eric Holk Ryan R. Newton
Indiana University, USA
{vollmerm,joelsven,eholk,rrnewton}@indiana.edu

Abstract

Writing high performance GPGPU code is often difficult and time-consuming, potentially requiring laborious manual tuning of low-level details. Despite these challenges, the cost in ignoring GPUs in high performance computing is increasingly large.

Auto-tuning is a potential solution to the problem of tedious manual tuning. We present a framework for auto-tuning GPU kernels which are expressed in an embedded DSL, and which expose compile-time parameters for tuning. Our framework allows for kernels to be polymorphic over what search strategy will tune them, and allows search strategies to be implemented in the same meta-language as the kernel-generation code (Haskell). Further, we show how to use functional programming abstractions to enforce regular (hyper-rectangular) search spaces.

We also evaluate several common search strategies on a variety of kernels, and demonstrate that the framework can tune both EDSL and ordinary CUDA code.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; I.2.8 [*Artificial Intelligence*]: Problem Solving, Control Methods, and Search—Heuristic methods

Keywords auto-tuning, meta-programming, parallelism, GPUs

1. Introduction

Modern GPUs offer significant performance potential, but this potential has proven to be difficult to realize in practice. Developing high-performance GPGPU programs is a tedious and time-consuming process, often involving manual tuning and non-portable, architecture-dependent optimization. Programmers are responsible for managing details of memory access, thread divergence, and synchronization points, all while fighting with programmers manuals that are vague about the actual workings of their device.

Still, GPUs offer enormous benefit in performance on some problem domains compared to CPUs, and GPUs are becoming more common on consumer electronics devices like laptops and smart phones. As GPUs become commonplace, the cost of ignoring the potential performance of GPUs will continue to increase.

FHPC'15, September 3, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3807-3/15/09...$15.00
http://dx.doi.org/10.1145/2808091.2808092

1.1 Tuning

When implementing an algorithm for a GPU using CUDA, the programmer is forced to make choices about how to decompose the workload over—in NVIDIA CUDA terminology—threads, warps, blocks, and the grid. The number of threads and blocks, the size of the grid, is called the *launch configuration* and is specified upon launching a workload onto the GPU. NVIDIA provides an Excel spreadsheet called the *Occupancy Calculator* (see reference [20] for information) that helps the programmer choose an appropriate launch configuration[1]. The programmer provides parameters for register and shared memory usage of the kernel, compute capability and shared memory configuration of the target GPU and can read out a launch configuration that leads to high GPU occupancy. There are two caveats: high occupancy does not always imply high performance, and high-performance CUDA code will often be specialized to a *specific* decomposition of work over threads warps, blocks, memory access patterns and array chunk sizes. Thus the tool may recommend a launch configuration that is not compatible with your code. In this case the programmer is forced to rewrite her code in order try out a different launch configuration.

One solution to this problem is to write "meta-programs", or programs that *generate* specialized CUDA kernels. Embedded, domain-specific languages for building these sorts of programs have been developed for languages like C++, Haskell, Python and many more [6, 7, 15, 17]. Such meta-programs may expose parameters to the meta-language, where they can be programmatically tuned.

Obsidian [24] is an embedded DSL in Haskell for GPU programming that provides a high-level functional interface for CUDA programming but still exposes the ability to make low-level decisions about the GPU kernel. In previous work on Obsidian, we have taken advantage of the ability to programmatically tune kernels by exhaustively searching over parameter spaces exposed by CUDA (number of blocks and threads per block) while generating size-specialized CUDA code. This approach can be successful, but the size of the search space combined with long CUDA compilation time make exhaustive search impractical in general. Additionally, a programmer may wish to expose more parameters than the block/thread parameters explored previously with Obsidian.

1.2 Auto-tuning

Auto-tuning is the technique of automatically controlling parameters to improve program performance. Borrowing from artificial in-

[1] CUDA 6.5 and up provide functionality similar to the Occupancy Calculator programmatically to the developer.

telligence research, auto-tuning systems often make use of heuristics when searching, avoiding expensive exhaustive searches [25].

High-level DSLs, like Obsidian, are well suited to take advantage of auto-tuning, because of their ability to expose compile-time decisions as ordinary parameters in their meta-language. This property also opens up an interesting possibility: the language for expressing kernels and the language for expressing auto-tuning can both be embedded side-by-side in the same meta-language. Using only in-language features, rather than (for example) using text substitution or pre-processing of source code, means that one is free to arbitrarily nest or compose auto-tuning sessions, or compute the configuration for an auto-tuning search at runtime, without much difficulty.

1.3 Contributions

The contributions of this paper are:

- We present a general framework for auto-tuning search in Haskell, including implementations of many common search strategies, like hill climbing, simulated annealing, and a genetic algorithm.
- We evaluate the auto-tuning search strategies by tuning GPGPU kernels, both generated by a functional GPU language (Obsidian) and written in an imperative language (CUDA), and present results showing that such kernels can be effectively tuned.
- We show how auto-tuning with regular search spaces is well captured as an applicative functor, or by treating each parameter request as a separate effect in a monad for extensible effects (Section 7).

2. GPU Programming

In this paper we focus on auto-tuning of GPU kernels, specifically, NVIDIA CUDA GPU kernels as generated by Obsidian. For more information about programming GPUs using CUDA see [19].

A GPU supporting CUDA is capable of running thousands of threads simultaneously and thrive on highly regular workloads. A CUDA program is expressed in SPMT form, Single Program Multiple Threads, which means that a single program control flow is executed over a (large) number of threads. The thread's identity is available in the program and often used to make choices or for indexing into data. The threads are mapped onto a hierarchy of threads, warps, blocks and finally the grid. This program abstraction hierarchy in turn maps onto the hardware hierarchy where different capabilities are present at different levels. For example, only threads that are part of the same block can exchange data via local shared memory. From the bottom up: each thread belongs to a warp, each warp to a block and all of the blocks make the grid.

Obsidian is an embedded language for programming GPU kernels. Specifically Obsidian generates kernels specialized for a a chosen problem size (array size). Generating kernels for fixed sizes reduces the amount of dynamic control flow that the GPU needs to execute at runtime.

The goal of Obsidian is to provide a tool for design space exploration when implementing GPU kernels [24]. To meet this end Obsidian exposes enough low-level details to enable tuning of GPU code for performance. Amongst these details are usage of shared memory, memory access patterns and distribution of work over threads, warps and blocks.

There are two different delayed array representations in Obsidian, *push* and *pull* arrays, which are also found in the embedded language Feldspar [5]. The two delayed array types correspond to different methods of computing an array; in both cases, a `compute` function computes the array elements and makes them manifest in memory. For more details about Obsidian programming see [9].

Obsidian has benefits over CUDA in that *hierarchy-level polymorphic* code can be expressed. Code can be written once and then reused either as a sequential computation in a thread, a parallel implicitly synchronized warp program or an explicitly synchronized block program. The example below implements a hierarchy-level-polymorphic reduction kernel:

```
reduceKernel :: (Compute t, Data a)
                => (a -> a -> a)
                -> Pull Word32 a
                -> Program t (Push t Word32 a)
reduceKernel op arr
  | len arr == 1 = return $ push arr
  | otherwise =
    do let (a1,a2) = halve arr
       arr' <- compute $ zipWith op a1 a2
       reduceKernel op arr'
```

The code above implements reduction of an array of length 2^n by recursively splitting the input array in half and combining halves, using `zipWith op`.

The type of `reduceKernel` says that its result is a `Program t`. That is a program that is to be executed at level `t` in the GPU hierarchy. The t parameter can be any of:

- `Thread` : for sequential execution on a thread.
- `Warp` : for parallel execution over the threads in a warp.
- `Block` : for parallel execution over the threads in a block.
- `Grid` : for execution over a collection of blocks.

In this case the `t` parameter is restricted by the constraint `Compute t`. this restriction means that the code can only be instantiated as a `Thread`, `warp` or `Block` program; these are the levels of the hierarchy that allow use of GPU shared memory. The function `compute`, stores an array in shared memory.

As an example, the programmer can instantiate the same reduction code at warp or block level, just by specializing the type:

```
blockReduce :: Data a
            => (a -> a -> a)
            -> Pull Word32 a
            -> Program Block (Push Block Word32 a)
blockReduce = reduceKernel

warpReduce :: Data a
           => (a -> a -> a)
           -> Pull Word32 a
           -> Program Warp (Push Warp Word32 a)
warpReduce = reduceKernel
```

The generated CUDA code for the warp version will not contain explicit synchronizations, while the block version will. This enables the programmer to easily implement a hybrid warp/block reduction strategy where the array that each block is processing is split up into chunks reduced in parallel by the different warps of the block. This is then followed by a reduction of the results produced by each warp.

```
hybridReduce :: Data a
             => Word32
             -> (a -> a -> a)
             -> Pull Word32 a
             -> Push Block Word32 a
hybridReduce warp_th f arr = exec $ b_body arr
  where
    b_body arr = do
      arr' <- compute
              $ asBlockMap (exec . warpReduce f)
              $ splitUp warp_th arr
      reduceKernel f arr'
```

The `hybridReduce` program makes use of the same reduction code twice: On line 10, `warpReduce` is mapped over sub-arrays and composed into a block computation with `asBlockMap` and on line 12 where partial results computed by each warp is reduced.

The `hybridReduce` program exposes a `warp_th` parameter that decides how large sub-arrays each warp should reduce and can be varied as long as it evenly divides the input array.

When generating CUDA code using Obsidian, knobs for varying the number of threads and blocks are exposed by default. In the above case, yet another knob for tuning is made available via the `warp_th` parameter.

3. A Simple Auto-tuning Framework

We developed a simple framework for tuning of Obsidian kernels over a fixed number of parameters. We based our framework on the following criteria:

1. It should easily interface with Obsidian.
2. Kernel code and scoring function should be reusable between search strategies.

Obsidian makes use of the NVIDIA NVCC compiler to compile GPU kernels. This process is monadic since it involves IO. Thus the scoring function needs to be able to perform IO actions. The IO effect needs to be composed with the tuning effect, for which we introduce a class of monads called `TuneM`:

```
class TuneM m where
  -- | Get parameter by index.
  getParam :: ParamIdx -> m Int

type ParamIdx = Int
```

This class exposes a single `getParam` function that takes an index representing a parameter and returns a parameter value. Each different search strategy provides an instance of `TuneM` along with a `runSearch` function. When running a search the user specifies from what ranges the parameters are sampled. These ranges are provided at the point of `runSearch` to (1) achieve a separation of concerns, and (2) to make sure the full dimensions of the search space are known before running the computation. In Section 7 we discuss improvements to this simple interface; for example, what happens if the programmer requests an index, using `getParam` that is out of bounds? In this simple version that is a runtime error. We can, however, catch this error at compile time with a more sophisticated use of the type system, which we demonstrate in Section 7.2.

The code below is an example of what a scoring function for an Obsidian kernel looks like. It is a monadic computation yielding a `Maybe Result`. The example tunes a kernel while varying two parameters `threads` and `blocks` obtained by two calls to `getParam`.

```
scoreIt :: (MonadIO m, TuneM m)
        => m (Maybe Result)
scoreIt = do

  threads <- getParam 0
  blocks  <- getParam 1

  liftIO $ catch (
    do time <- timeIt threads blocks
       return $ Just
              $ Result ([threads,blocks],time)
    )
    (\e -> do putStrLn (show (e :: SomeException))
              return Nothing
    )
```

The `timeIt` function, wrapped in exception handling, represents compiling, executing and timing the kernel. This involves invoking external tool, linking compiled binary and calling a foreign function and it can go wrong. If everything is successful a result containing the parameter settings and score (in this case, the time) is returned, otherwise `Nothing`.

The result type that the search strategies use as a score during tuning is specified by the user. Different search strategies place different constraints upon the result type. However, all instances require an `Ord` instance for results. In the `scoreIt` example above, we could have used a simple result type, `type Result = Double`. But we instead show a simple custom datatype that also keeps the parameters that led to a timing, paired together with the time:

```
data Result = Result ([Int],Double)

instance Ord Result where
  compare (Result (_,d1)) (Result (_,d2))
    = compare d1 d2

instance Show Result where
  show (Result (p,r)) = show p ++ " | " ++ show r
```

For our evaluation in Section 6 we focus on single-objective searches, measuring only overall runtime. However, it would not be difficult to incorporate multiple objectives, such as memory usage or energy consumption, by adding them to the `Result` type and extending the `Ord` instance of `Result` to weigh the different values appropriately.

When tuning an Obsidian kernel the `timeIt` function generates the CUDA code, compiles it using NVCC and then executes and times that kernel.

```
timeIt :: Int -> Int -> IO Double
timeIt threads blocks = do
  -- Generate code using parameters.
  -- Compile generated code.
  -- Run and time executable.
```

As an example, the code below shows what running a search using the exhaustive strategy looks like. Section 4 presents the other search strategies implemented using the framework.

```
import Auto.ExhaustiveSearch as ES

main = do
  resLog <- ES.runSearch
              (ES.Config [ [x*32| x <- [1..32]]
                         , [x*32| x <- [1..32]]])
              scoreIt

  let best = getBestResult resLog

  case best of
    Nothing -> putStrLn "No results"
    Just r  -> putStrLn $ show r
  putStrLn "Done!"
```

The `runSearch` function provided by `ExhaustiveSearch` takes a configuration as parameter that specifies the parameters values to sample exhaustively as well as the monadic action used to score each parameter setting. The result of a search is a log collected during searching. This log includes the best setting found during search.

4. Searching

For the "auto" part of "auto-tuning," we need to *search*. With our framework, a program may expose a fixed number of parameters to be automatically tuned according to a programmer-defined search strategy. All the potential configurations for these parameters form a set referred to as the *search space*.

4.1 Strategies

We implemented the following heuristic search strategies in our framework: hill climbing, simulated annealing, and a genetic algorithm. Each of these search strategies have a notion of *neighborhoods*. In our auto-tuning experiments with these three strategies, we chose a search space of bit strings. Each solution was made of some number of integers, represented in bit strings, and the neighborhood of each solution is all the other bit strings that differ by only one bit. This representation is useful for discrete optimization problems because it provides many neighbors for each state, allowing a search strategy to make both large and small jumps around a search space.

- *Hill climbing*. Hill climbing is an search strategy that starts with a random solution, and seeks to incrementally improve it by changing individual parts of the solution [21]. It tries different permutations of the current solution (neighbors), and if it finds one better than the current solution, it *moves* there, or sets that neighbor as the current best solution. This process is repeated up to some fixed number of iterations, or until no neighbors of the current solution are an improvement. This is a simple and powerful search strategy, but because it stops when there are no neighbors with better evaluations, the search can get stuck in local optima. We refer to the variant of hill climbing we use as "bit climbing," which is hill climbing with a bit string.
- *Simulated annealing*. Simulated annealing is a heuristic-based search strategy that is inspired by the process of annealing in metallurgy [13]. Like hill climbing, the search starts with a random state, and generates a neighborhood. The search also has a *temperature*, which denotes how likely it is to *move* to a neighbor that is worse than its current solution. Over the course of the search, the temperature *cools*, making the search less likely to move to worse solutions, and more likely to always choose new solutions that are as good or better than the current solution.
- *Genetic algorithm*. A genetic algorithm is a heuristic-based search strategy that attempts to mimic the behavior of evolution by natural selection [18]. A population of candidate solutions is first randomly generated, then *evolved* based on some fitness criteria. Each solution is evaluated, which determines its fitness, then solutions are combined based on their fitness. We use tournament selection: some number (greater than two) of solutions are picked from the population to engage in a tournament, where the two with the highest fitness cross over (combine) to produce an offspring—we repeat this process until we have a new generation of solutions. The offspring solutions are also subject to random mutations, which provide some randomness to help prevent the search from getting stuck in local optima

Each search provides a function for launching the auto-tuning search, and may expose any search-specific configuration options.

```
-- | Configuration options for hill climbing.
data Config = Config { numBits   :: Int
                     , numParams :: Int
                     , numIters  :: Int
                     }

-- | Run the hill climbing search.
runSearch :: (Show result, Ord result)
          => Config
          -> BitClimbSearch (Maybe result)
          -> IO (ResultLog result)
```

Searches are launched by calling the provided run function.

```
runSearch (Config bitCount paramCount iterCount)
          (scoreIt :: BitClimbSearch
                      (Maybe Result))
```

If we wanted to convey to the search that we wanted to, for example, exclude part of the search space (if we had domain knowledge telling us this would be beneficial), there are a couple of ways we could do this. We could, for example, extend `Config` to take a new field, and modify `BitClimbSearch` to use it. We could also modify the range of numbers searched, and change the evaluation function to interpret the parameters differently.

While our evaluation only considers problems of a few parameters, these search techniques (and different permutations and improvements on them) have been shown [4] to perform well on high-dimensional problems. Additionally, there is value in applying auto-tuning to programs with a small number of parameters—such as the occupancy calculation mentioned in the introduction—and we believe our results are representative of the auto-tuning process for many GPU kernels.

4.2 Exploration vs. Exploitation

A trade-off evident in these approaches, and in search generally, is the balance between *exploration* and *exploitation*. Hill climbing is 100% exploitation, because it always moves to a better solution in its neighborhood. In the case of a local optima, this means it has nowhere to go, and if there is a better global optima, it has no way to reach it. In contrast, simulated annealing and genetic algorithms involve some amount of randomness, which is the key to exploration. In a search space with many local optima, some exploration is generally needed to avoid getting trapped.

This trade-off between exploration and exploitation is critical to evaluating these search strategies in auto-tuned GPU kernels, and we discuss it in more detail in Section 6.

4.3 Domain Knowledge

These search strategies themselves are tunable. They provide some configurable knobs that may be set up to take advantage of domain knowledge about our kernels. For example, the genetic algorithm exposes knobs for the mutation rate and tournament size, both of which affect the ratio of exploitation/exploration that happens in the search. All the searches also provide general knobs, such as scaling the parameters they tune and excluding ranges of parameter values from being considered.

Our framework would allow nested auto-tuning searches, so it is possible to apply the auto-tuning search to choosing parameters for auto-tuning—a topic for future work.

5. Applications

In Section 6, we evaluate the search framework on a number of applications. The evaluation is based on a number of Obsidian kernels—Fractals, Histogram and Reduction—as well as a breadth-first search algorithm implemented directly in CUDA. This section describes what we tune for each of these applications and what tuning those parameters means.

5.1 Fractal

The Mandelbrot fractal is an example of an embarrasingly parallel application. In this program we compute a 1024×1024 pixel image of the Mandelbrot set. Each pixel is computed completely independently of every other by an iterative process.

Our tuning effort for this program tunes the number of threads and the number of blocks used to compute the image. Varying these numbers changes the number of pixels computed per CUDA thread and block.

5.2 Histogram

Computing an histogram is done by counting the number of occurences of each value in an array. For example the histogram of $\{0,1,1,4,3,3,0\}$ is $\{2,2,0,2,1\}$. One way to compute a histogram in parallel on a GPU is to use the atomic increment instruction available on CUDA GPUs.

The parameters we tune for this program is the number of threads and blocks used. Sliding the values for these parameters changes the amount of sequential computation within a thread or block. It also potentially has effects on contention as different atomic operations performed in sequential code will not contend with each other. The trade-off is sequentiality vs contention.

5.3 Reduction

In Section 2, reduction was used as an example Obsidian program. When auto-tuning this program there are up to three parameters available for tuning. The first is an threshold or chunk size at which to run warps in parallel, each performing a reduction of a part of the input array. The second parameter decides the number of threads to use per block, which changes the number of warps that run in parallel. And finally the number of blocks to launch can also be tuned.

5.4 Breadth-First Search

Graph algorithms are irregular and thus at first glance not a perfect fit for GPUs that thrive on regular, massively parallel problems. Work has been put into developing efficient graph algorithms for GPUs, for example [10, 11]. And with the introduction of dynamic parallelism in CUDA for compute capability 3.5 and above [2] the situation for graph algorithms on GPUs is improving [26].

The breadth-first search (BFS) code behind reference [26] is used as one of the auto-tuning case studies in this paper. This algorithm dynamically switches between several code variants based on the number of outgoing edges on each vertex. The smallest vertices (i.e. those with out degree less than SMALL_VERTEX_TH) are processed sequentially using a simple `for` loop. Medium-sized vertices (those larger than small vertices but with out degree less than KERNEL_TH) are processed using a warp-cooperative algorithm. Finally, the largest vertices are processed by launching a child kernel to process all outgoing edges in parallel. Each successive variant adds overhead and the variant should only be used when there is enough additional parallelism to overcome the overhead.

These two parameters are amenable to tuning by our framework to find the best cutoff points.

Note that the BFS application is implemented in C++ and CUDA directly. Our search framework is flexible enough to handle tuning arbitrary functions: the scoring function for the BFS code performs IO actions that interface with the CUDA compiler and runs the application via a shell command (`createProcess`). This process was more cumbersome than tuning Obsidian code directly, but it demonstrates the flexibility of our approach.

6. Evaluation

In this section we evaluate the search strategies for a selection of GPU applications. Each timing measurement is the result of many repeated runs of the GPU kernel. We conducted our experiments on a system with dual 12 core Intel Xeon CPU E5-2670 v3 running at 2.30GHz and a Tesla K40 GPU. The Tesla K40 GPU has 2880 CUDA cores running at 745.0 MHz.

For some of the applications (where it was feasible), we use a series of exhaustive searches to approximate optimal parameter assignments, and compare our search results to these approximate optimal values. The parameter assignments that yield the best result (lowest time) changes from run to run—many measurements differ by milliseconds, so the data is necessarily noisy. Note that these exhaustive searches are performed at a lower resolution, where the space is uniformly sampled with a fixed stride in each dimension.

We represent search spaces visually as heat maps, with "hotter" (lighter color) sections representing better solutions. In our results, an "iteration" of the search is defined as an invocation of the evaluation function, so (for example) a genetic algorithm with a population of 10 will do 10 iterations per generation. For mandelbrot and histogram searches where parameters were multiples of 32, we ran the search for fewer iterations than the other benchmarks (25 rather than 50 iterations).

6.1 Fractal

Our mandelbrot kernel exposed two parameters: number of threads and number of blocks. The results are shown in Figures 2, 3, and 4.

For this kernel we also used the CUDA Occupancy Calculator to see what guidance it provides for chosing parameters. The caclulator suggest that the number of threads should be one of 128, 256, 512 and 1024 and that at those settings you need 2, 4, 8 or 16 blocks per GPU multiprocessor to have enough warps to completely saturate the GPU. The TESLA K40 GPU has 15 multiprocessor meaning that 30, 60, 120 or 240 blocks in total saturates the device. Figure 1 suggests that using any of these settings blindly is not ideal. The programmer needs to apply some heuristic or experimentation in addition to using the occupancy calculator to find one of the better parameter settings.

We know that, when it comes to the number of threads, it is very likely that the best parameter setting should be a multiple of 32. This is because the warp size on a CUDA GPU is 32 threads. In our evaluation, we considered searches that took advantage of this knowledge, and searches that did not. It is clear that the addition of domain knowledge to this problem greatly improves the results from all of the searches. With this domain knowledge included,

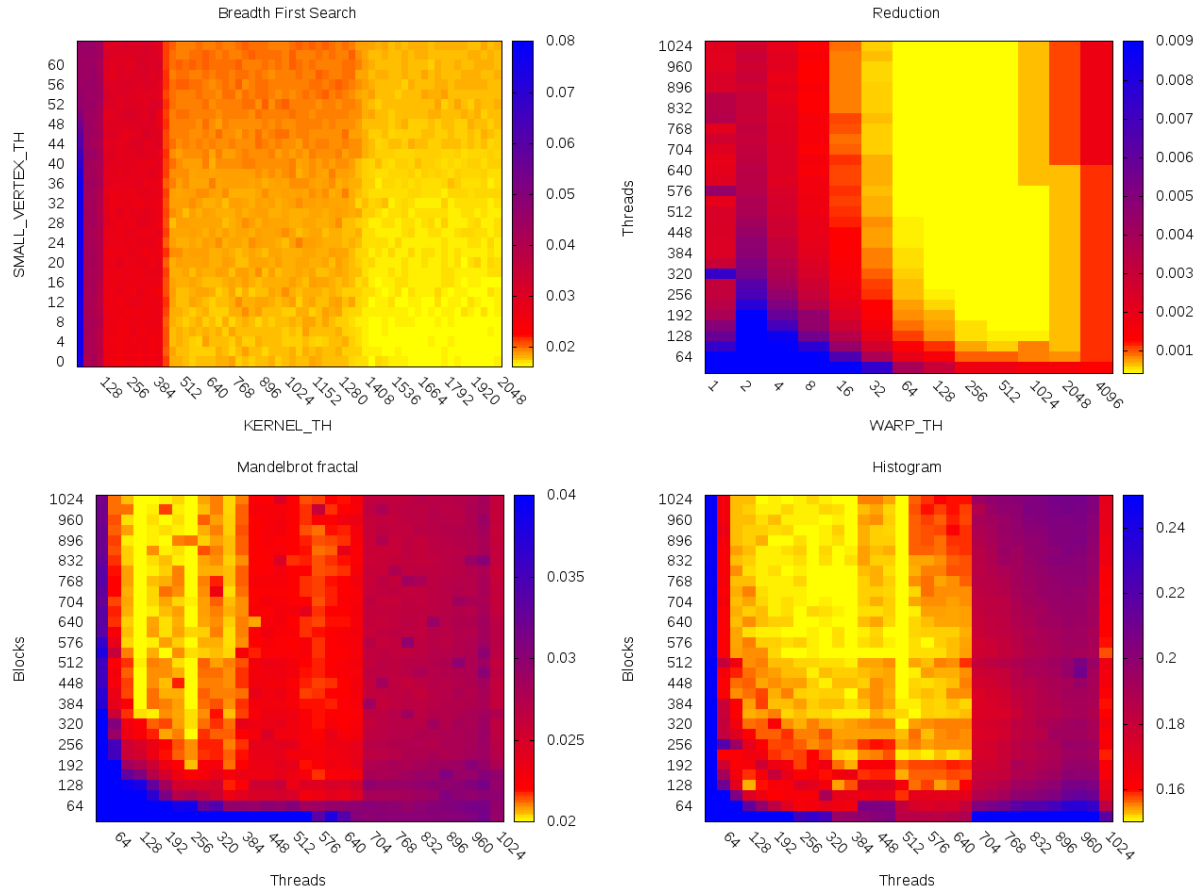

Figure 1. Heat maps showing the fitness landscape for each search space

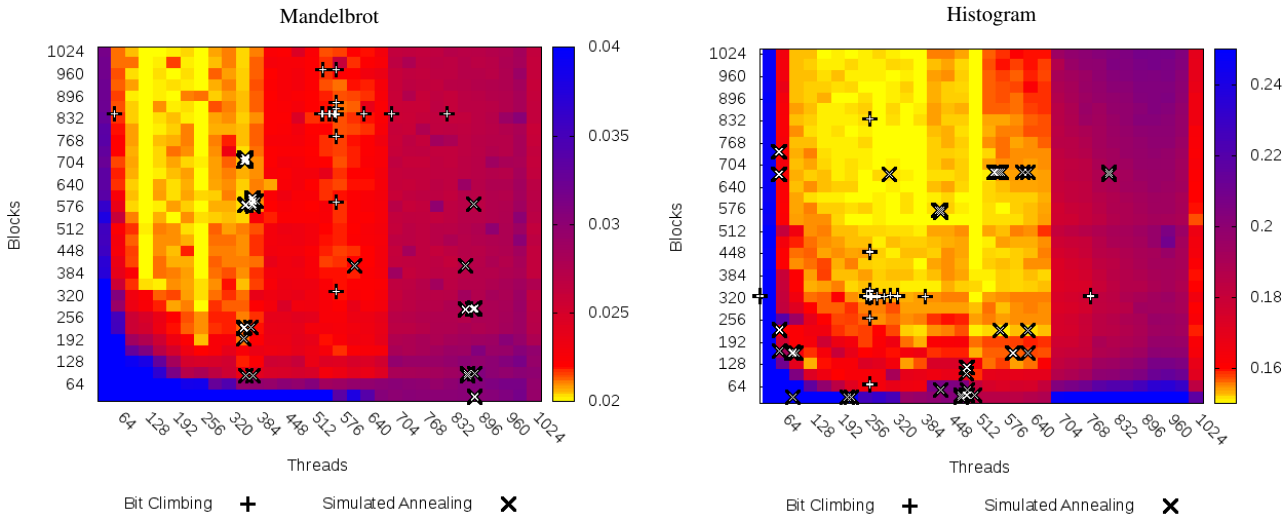

Figure 2. *Left*: Two dimensional heat map of Mandelbrot search space. *Right*: Two dimensional heat map of histogram search space

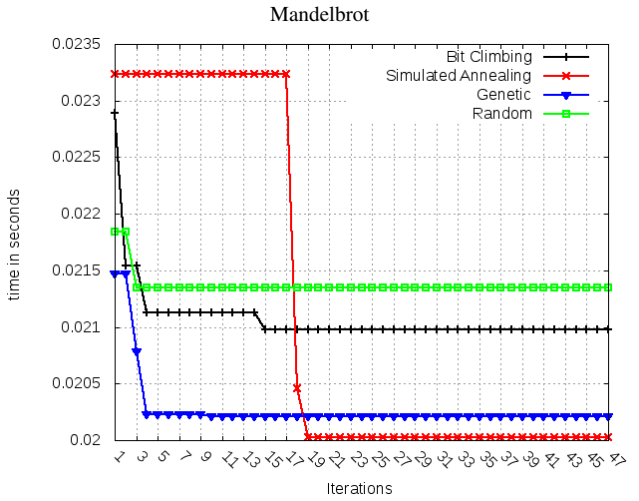

Figure 3. Best solutions over time during search for Mandelbrot

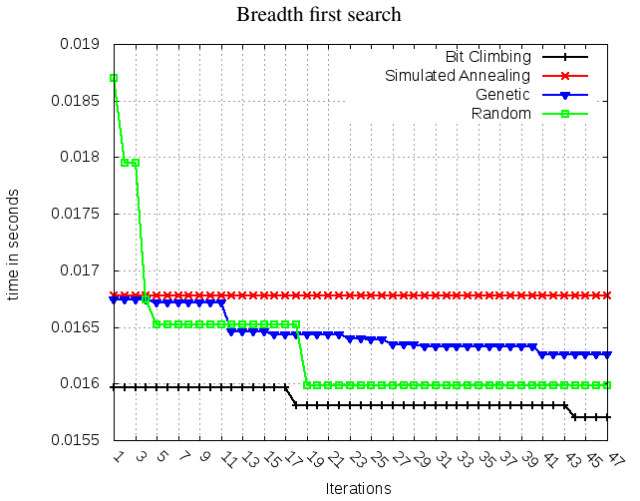

Figure 5. Best solutions over time during search for BFS

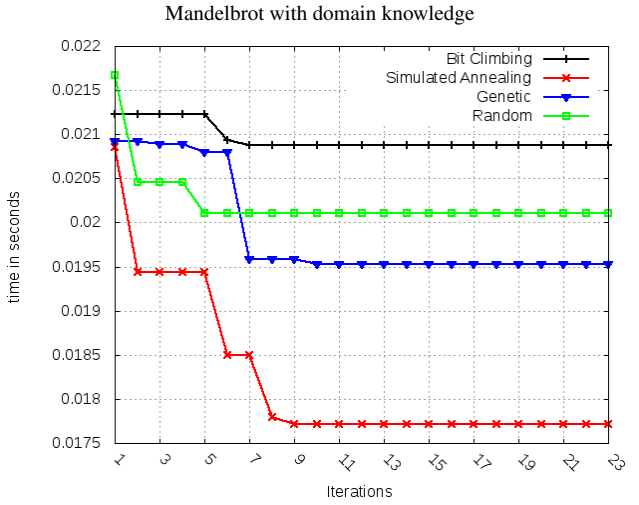

Figure 4. Best solutions over time during search for Mandelbrot, using multiples of 32.

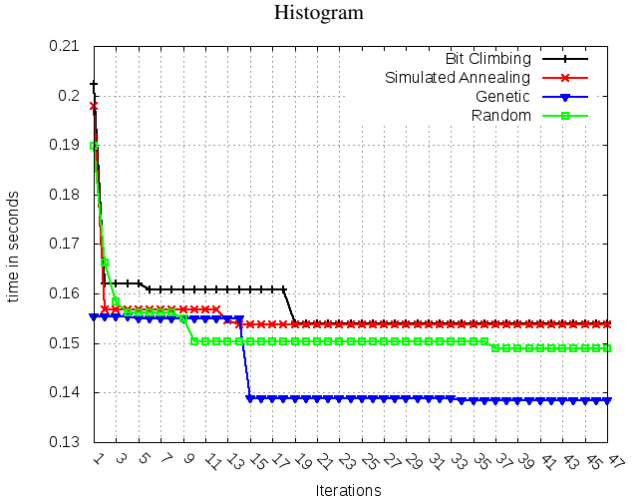

Figure 6. Best solutions over time during search for Histogram

even the random search reached within 5% of the approximate optimal value after 25 iterations.

By looking at the heat map in Figure 2, it is obvious that there are plenty of potential local optima for a search to get stuck in. The data presented in Figure 3 and Figure 4 confirm this intuition: hill climbing performs poorly, getting stuck in a local optima, while the other searches (which involve some amount of exploration) perform better.

In particular, simulated annealing is the big winner. With multiples of 32 it reached a value within 1% of the approximate optimal within 10 iterations, and with the full search it reached a value within 11.3% of the approximate optimal within 20 iterations.

6.2 Breadth-First Search

Our BFS kernel exposes two parameters: `SMALL_VERTEX_TH` and `KERNEL_TH`. The results are shown in Figure 5. Because of the large range of reasonable cutoffs, the search space for BFS is big. The heatmap in Figure 1 shows part of that search space, sampling multiples of 32 in the X axis. Compile time for this application is $\approx$ 11 seconds on the benchmarking platform, which limits our ability to exhaustively search a large space.

Here, in contrast to the previous mandelbrot kernel, hill climbing performs the best overall, and simulated annealing never even manages to improve its initial guess. Interestingly, the random search ends up beating two of the heuristic searches (genetic and simulated annealing) after 19 iterations. The search space is large enough, and there are enough points on it yielding sub 0.02 second times, that we speculate these two searches with randomness become basically interchangeable with purely random selection inside 50 iterations.

6.3 Histogram

Our histogram kernel exposes two parameters: threads and blocks. The results are shown in Figure 6, Figure 7, and Figure 2.

As with mandelbrot, we have some knowledge of good parameter assignments: they are likely multiples of 32. In the search over multiples of 32, the search space is small enough that all the searches converge on more-or-less equivalent results after 16 iterations. By reducing the search space with domain knowledge, the tuning problem has become simple enough that search heuris-

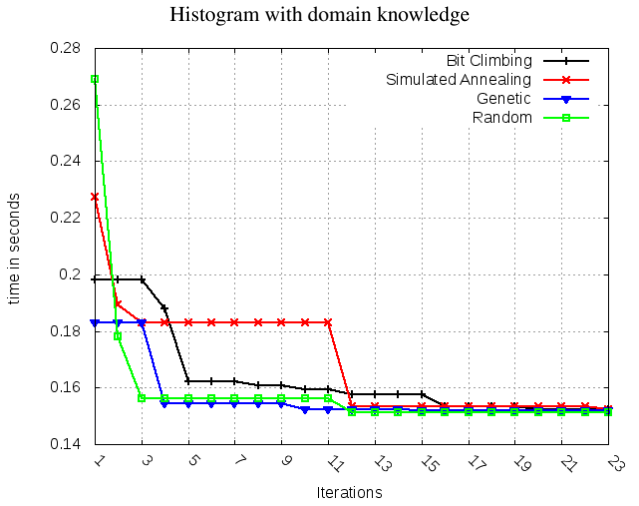

Figure 7. Best solutions over time during search for Histogram using multiples of 32

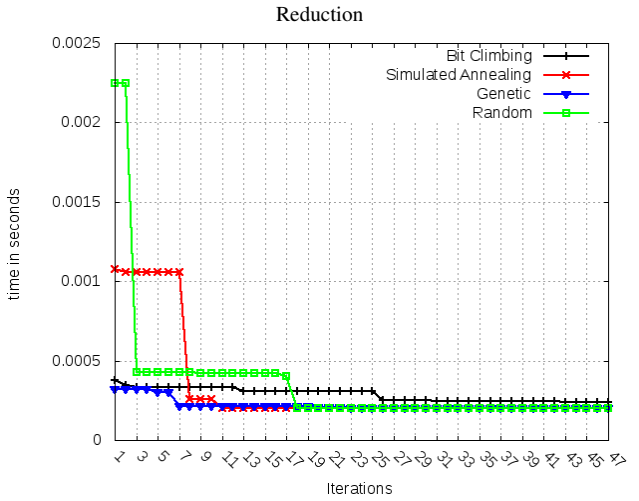

Figure 8. Best solutions over time during search for reduction

tics are no longer necessary. The best result, however, from these searches was still 7% slower than the approximate optimal result.

The full search covers a significantly larger space. Here, the genetic algorithm performs the best after 15 iterations, reaching a result that is equivalent to the approximate optimal result.

6.4 Reduction

Our reduction kernel exposes three paremters: warp threshold, threads, and blocks. The results are shown in Figure 8. The heat map for reduction in Figure 1 is obtained by varying the warp threshold and number of threads while keeping the number of blocks fixed. An exhaustive search over all parameters would have rendered a volume.

All searches, including random search, yield nearly identical results after 25 iterations. Like BFS, the search space is too large to exhaustively search, but from our experimentation we speculate that the results found by the auto-tuning searches are very close to optimal.

This tells us that this problem is not very hard, despite having three parameters (one more than the other applications considered above) and a very large search space. Scaling up the problem size increases the times, but the results follow the same pattern.

While this data does not require much explanation or analysis, it is representative of common GPU kernels that are trivial to tune with simple searching—low hanging fruit for optimization with Obsidian.

6.5 Interpretation

When domain knowledge is incorporated into the auto-tuning searches for fractal and histogram, the auto-tuning searches perform very well. On mandelbrot, for example, searches over multiples of 32 start out with times close to 0.02 seconds, which is within 12% of the approximate optimal value, while some full searches do not surpass 0.02 seconds even after 50 iterations. This is not surprising: having knowledge of the search space that drastically reduces its size is going to make the search much more effective.

Going by the heat maps, it is obvious that there are many spots in the search spaces that are not optimal but are reasonably close. Continuing with the example of mandelbrot, times of around 0.02 seconds are solid yellow on the heat map, and while Figure 4 shows how searches narrow in on sub 0.02 second times, one can imagine situations where being within 12% of the optimal answer is "good enough."

In the case of BFS, because the search space was too expensive to exhaustively search, we cannot make an argument or conclusion about optimality. Still, the auto-tuning searches returned results below 0.017 seconds, which appears to be (at least informally) in the "good enough" range.

7. Discussion: Functional Interfaces for Auto-Tuning

The simple auto-tuning interface we have presented is a lot cleaner than a pile of scripts and C preprocessor hacks. However, it is possible to go further and statically *guarantee* that the application being tuned never requests a parameter which does not exist. In this section we consider two solutions to this problem.

7.1 Applicative Tuning

One solution is to formulate auto-tuning as an *applicative functor* rather than a monad, corresponding to the `Applicative` class in Haskell. Applicatives are often useful when the structure of the computation needs to be observed, such as collecting data fetch requests in the Haxl project [16], or collecting all `getParam` parameter requests here. That is, if we can collect all `ParamID`s used by the computation before running it, we can ensure that we bind all parameters that it will possible reference.

We can create an `Applicative` tuner whose representation is simply a function of tuning parameters together with possible settings for those parameters, as follows:

```
data Tune a = Tune { run :: Params -> a
                   , paramDomains :: [Domain] }
type Params = [Int]
type Domain = (Int,Int)
```

In this case, the parameterized function is pure, `Params -> a`, but in general it's better to allow this computation to itself be monadic, like the IO-based Obsidian computations we've seen in this paper. Thus, we make `Tune` into a kind of transformer, `TuneT`, parameterized over a monad `m`:

```
data TuneT m a =
     TuneT { run :: Params -> m a
           , paramDomains :: [Domain] }
```

```
instance Monad m => Functor (TuneT m) where
  fmap f (TuneT g l) =
    TuneT (\p -> do x <- g p; return (f x)) l

instance Monad m => Applicative (TuneT m) where
  pure x = TuneT (\[] -> return x) []
  TuneT f l1 <*> TuneT g l2 =
    let len1 = length l1
    in TuneT (\ls -> let (x,y) = splitAt len1 ls
                     in do f' <- f x
                           v  <- g y
                           return $ f' v)
             (l1 ++ l2)
```

Figure 9. The full Functor and Applicative instances for TuneT. Here the applicative instance ensures that the subcomputations each get their own share of the parameter settings.

Note that this business of abstracting over an underlying monad, `m`, was *not* necessary with `TuneM` above, because *any* monad could be made an instance of `TuneM`.

In this applicative `TuneT` setting, `getParam` is defined as:

```
getParam :: Monad m => Domain -> TuneT m Int
getParam d = TuneT (\[p] -> return p) [r]
```

Because `getParam` returns a non-monadic value, `do` notation cannot be used, and instead we use applicative combinators. For example, to add the value of two tuning parameters, we would write:

```
(+) <$> getParam (1,10) <*> getParam (0,3)
```

And then to run a tunable computation, a particular search strategy might expose the following run function, which, given a computation that exposes a `Score`, searches for values of `Params` that maximize that score.

```
tune :: Tune IO (a,Score) -> IO (a, Params, Score)
...
type Score = Double    -- Simple scoring...
```

In this case, we assume the underlying monad is `IO`, although it could also accept any `Monad m`, provided there is a function to run that monad inside IO, i.e., `m a -> IO a`. In either case, the `tune` function must be able to *run* the underlying monadic computations, so that it can execute different configurations as it varies `Params`.

Many tunable computations gather runtime timings, which necessitates IO. However, it is also important to consider tuning pure functions and monadic computations with *dischargeable* effects:

```
tunePure :: Monad m => (forall b . m b -> b) ->
            Tune m (a,Score) ->  (a, Params, Score)
```

This might arise, for example, if we are tuning a computation against an abstract, deterministic performance model.

To conclude this section, Applicative offers a semantic fit for auto-tuning with fixed search spaces. It accomplishes the "staging" whereby parameters used are gathered before the tunable computations begin execution, without necessitating the indirection of requesting parameters by index or key. The drawback of the applicative approach—and our reason to consider one more alternative design—is that an applicative that returns a monadic value is awkward to deal with. It is a form of effect composition that works much less smoothly than, e.g., monad transformers, which retain do-notation.

7.2 Monads with Extensible Effects

The final solution we consider here—and which we ultimately recommend—uses a modern mechanism for composable effects in Haskell. In particular, we employ the `extensible-effects` library by Kiselyov et. al.[2]. As described in a 2013 paper [14], this framework replaces monad transformer stacks with a single monad `Eff r a`, where `r` encodes the set of effects, and constraints such as `Member (State Int) r`, capture the fact that `r` contains *at least* a state effect storing an integer state.

For the purpose of auto-tuning, we add an effect `Param s`. This is akin to a *Reader* effect, indicating that the computation reads a parameter identified by `s`. We could use *any type* for `s` but we choose type-level strings, i.e. of kind `Symbol`, which provide us with an unlimited source of unique types without requiring extra `newtype` declarations.

Since GHC 7.8, the `GHC.TypeLits` module provides facilities for dealing with type-level string literals, including synthesizing instances of the `KnownSymbol` class, which we use below.

```
-- The data type for tuning effects:
data Param s a =
  KnownSymbol s => GetParm (Int -> a)
 deriving (Typeable)

-- getParam is polymorphic in *which* param:
getParam :: (Member (Param s) r,
             KnownSymbol s, Typeable s) =>
            Proxy s -> Eff r Int
```

The return value of `getParam` is still a simple `Int`. For its input, here we use the standard approach of passing a `Proxy` datatype—with a phantom type argument—as a means of passing in a type argument to the function. To invoke `getParam` then, the user needs only pass in the type-level name of the parameter:

```
go = do x <- getParam (Proxy::Proxy "a")
        y <- getParam (Proxy::Proxy "b")
        ...
```

Type-level keys By naming parameters at the type level, it is possible to know which parameters a computation uses before running it. A `runSearch` function provide by a search strategy then takes a particular parameter and the desired domain for that parameter. For example:

```
x = runSearch $
    setParam (Proxy::Proxy "a") (0,10) $
    setParam (Proxy::Proxy "b") (10,20) $
    go
```

Here it is possible to write different versions of runSearch for different search strategies: e.g. hill climbing vs. genetic algorithms. Thus it is not necessary to have a type class like TuneM above, because the selection of monad (Eff) does not determine the search strategy. Further, in the above example, any parameter expected by the computation but not provided by a `runParam` would result in a static *type error*.

Thus, the extensible effect approach ensures **safe parameter access**: it is impossible to request a parameter value that has not been provided by the tuning framework. Further, the search domain of all parameters is known *before* execution begins. And it also

[2] `https://hackage.haskell.org/package/extensible-effects`

enables **composability**. It is possible to run a sub-computation, with two parameters, or to compose it with a larger computation and jointly optimize across the larger set of parameters. No code modification in the tunable computations is required to enable this composability.

8. Related Work

OpenTuner [4] is a general framework for auto-tuning fixed parameter search spaces. It provides an interface for building and composing domain-specific search strategies that can be tailored to a particular tuning problem. OpenTuner also introduces the idea of *ensambles*, combining search strategies to cooperatively find optimal sollutions.

PATUS [8] is a auto-tuning and code generating framwork for stencil computations. A user of PATUS describes a stencil computation in a machine independent way, at a high-level, and the system generates architecture specific and auto-tuned implementation for a target GPU or CPU. The tuning process is configurable so that different auto-tuning strategies can be used.

In reference [1] tuning is applied to the problem of compiler optimization pipelines. A compiler is designed with a number of reorderable optimization passes. Different orderings of these optimization passes are evaluated during the tuning process.

PetaBricks[3] is a language and auto-tuning compiler that includes algorithmic choice. For example, sorting algorithms are often implemented using hybrids of several algorithms, parhaps starting out as merge sort but switching to insertion sort for small arrays (reference [22] uses an hybrid algorithm to implement fast sorting on GPUs). The problem with hybrid algorithms is that the cut-off point, where to switch between algorithms, often varies between target systems. PetaBricks tunes these cut-off points as well as selects algorithms to use from a palette of choices.

In reference [23] the authors propose a rewrite system for generating efficient OpenCL kernels from high-level code. The rewrite rules define a search space which can be searched over in the goal to produce efficient low-level GPU code. As an example a map operation over an array can be decomposed in different ways over the hierarchy of the GPU programming model. Tuning in this system means to select rewrite rules to apply.

In reference [12], a framework for implementation of auto-tuned EDSLs in Python is described. The system generates multiple variants of target code from a single high level description. When invoking a computation on a set of data, the runtime system chooses one of the variants and executes it. If all variants have been tried, the fastest variant is choosen. The approach to exploring the search space is exhaustive search. The system does not perform install time tuning, the tuning happens online as the programs are being run.

9. Conclusion

From the data presented above, and from our experience in implementing and tuning these kernels, we can make some general conclusions.

- We confirmed expected results regarding the utility of auto-tuning for setting thresholds and block sizes. As always, incorporating domain knowledge, whenever possible, into an auto-tuning search is beneficial, and adding domain specific knowledge to the search using the same meta-language used to implement the GPU kernel felt natural.
- Wiring up auto-tuning programmatically in Haskell proved to be convenient. We needed to evaluate many different search configurations on the same few GPU kernels, and we were able to use ordinary Haskell code automate much of the work of setting up and evaluating the different searches.

Because we find good results with such modest effort, we believe that average GPU kernels should be auto-tuned by default. With Obsidian's meta-programming approach, no additional effort is required to make tunable designs, even if they explore different uses of the GPU's hierarchy.

Acknowledgments

This work was supported by NSF grant XPS-1337242.

References

[1] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 231–239, New York, NY, USA, 2004. ACM. ISBN 1-58113-806-7. . URL `http://doi.acm.org/10.1145/997163.997196`.

[2] Andrew Adinetz. Adaptive Parallel Computation with CUDA Dynamic Parallelism, May 2014. URL `http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-par%allelism/`.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. . URL `http://doi.acm.org/10.1145/1542476.1542481`.

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. . URL `http://doi.acm.org/10.1145/2628071.2628092`.

[5] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. 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*, pages 169–178. IEEE, 2010.

[6] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. . URL `http://doi.acm.org/10.1145/1941553.1941562`.

[7] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[8] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. .

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

[10] A. Davidson, S. Baxter, M. Garland, and J. Owens. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359, May 2014. .

[11] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In S. Aluru, M. Parashar, R. Badrinath, and V. Prasanna, editors, *High Performance Computing HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-77219-4. .

[12] S. A. Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-specific Embedded Languages*. PhD thesis, Berkeley, CA, USA, 2012. AAI3555748.

[13] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[14] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2383-3. . URL `http://doi.acm.org/10.1145/2503778.2503791`.

[15] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[16] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 325–337, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. . URL `http://doi.acm.org/10.1145/2628136.2628144`.

[17] M. D. McCool and S. D. Toit. *Metaprogramming GPUs with Sh*. A K Peters, 2004. ISBN 978-1-56881-229-8.

[18] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[19] NVIDIA. CUDA C Programming Guide. URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3XP6X8l00`.

[20] NVIDIA Corporation. Cuda C Best Practices Guide, 2015. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`.

[21] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2003. ISBN 0-13-790395-2.

[22] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10): 1381 – 1388, 2008. ISSN 0743-7315. . General-Purpose Processing using Graphics Processing Units.

[23] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating Performance Portable Code using Rewrite Rules: From High-level Functional Expressions to High-Performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, New York, NY, USA, 2015. ACM.

[24] B. J. Svensson, M. Sheeran, and R. R. Newton. Design Exploration Through Code-generating DSLs. *Commun. ACM*, 57(6):56–63, June 2014. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/2605685`.

[25] R. W. Vuduc. Autotuning. In *Encyclopedia of Parallel Computing*, pages 102–105. 2011. . URL `http://dx.doi.org/10.1007/978-0-387-09766-4_68`.

[26] P. Zhang, E. Holk, M. Zalewski, S. Mcmillan, J. Chu, and A. Lumsdaine. Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms, 2015. `http://www.cs.indiana.edu/~eholk/papers/bfsdp.pdf`.