Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

Beauty And The Beast: Exploiting GPUs In Haskell

Alex COLE [a,1], Alistair McEWAN [a] and Geoffrey MAINLAND [b]
[a] *Embedded Systems Lab, University Of Leicester*
[b] *Microsoft Research, Cambridge*

Abstract. In this paper we compare a Haskell system that exploits a GPU back end using Obsidian against a number of other GPU/parallel processing systems. Our examples demonstrate two major results. Firstly they show that the Haskell system allows the applications programmer to exploit GPUs in a manner that eases the development of parallel code by abstracting from the hardware. Secondly we show that the performance results from generating the GPU code from Haskell are acceptably comparable to expert hand written GPU code in most cases; and permit very significant performance benefits over single and multi-threaded implementations whilst maintaining ease of development. Where our results differ from expert hand written GPU (CUDA) code we consider the reasons for this and discuss possible developments that may mitigate these differences. We conclude with a discussion of a domain specific example that benefits directly and significantly from these results.

Keywords. GPU, Haskell, domain specific language.

Introduction

This paper stems from previous work on array computation libraries, extending it to include a Domain Specific Language (DSL) in Haskell with various data-parallel back ends. This DSL is called "PEGGY" and the results presented here compare CUDA code generated by this system to code hand-written in a variety of other languages and for a variety of data-parallel execution systems, while also looking at the development efforts involved in each. Haskell is a high-level functional programming language with a proven track record in embedding DSLs, [1,2,3]. The main target examined here is CUDA, a low-level imperative programming language for developing General Purpose Graphics Processing Unit (GPGPU) software.

Multiple libraries have been written for Haskell to enable the use of GPUs [4,5,6], the aim is to build on top of these libraries and enable the development of code in a language closer to the array computations in question. This was as a proof-of-concept to show a language that uses domain-specific abstractions (Section 1), without exposing the underlying back end libraries to anyone unfamiliar with, say, CUDA. Section 2 looks in more detail at using this library.

Three case studies (Convolution, Electrostatic Charge Map Generation, and UDWT; Section 3) are developed using PEGGY. These sections show code that would be familiar to any Haskell programmer, originally operating on lists and minimally modified in an attempt to reclaim some of the performance losses. Section 4 then looks at how the CUDA code generated by Haskell performs against expert hand-written CUDA code, C code running sequen-

[1]Corresponding Author: *Alex Cole, Department of Engineering, University of Leicester, Leicester, LE1 7RH, England.* Tel.: +44 (0)116 252 2578; E-mail: `ac245@le.ac.uk`.

tially, and code written using the Accelerator library [7] (this is included as it was looked at in previous work in this area [8]).

There are three main points of interest:

1. Is it possible to build a language on top of multiple back ends simultaneously, in a domain specific language?
2. Does this abstracted code produce code a run-time with adequate performance from concise problem descriptions?

This leads to a third question:

3. Can this be done again for other languages?

The contribution of this paper is to show that the answer to all these questions is "yes"—a DSL built on top of existing Haskell libraries can still get performance close to what is achievable in CUDA when compared to other execution systems, while allowing for a more terse representation of algorithms. This is preliminary feasibility work before building a full DSL for Mass Spectrometry analysis (Section 5).

The authors' previous paper on this topic [8] looked at the execution of several exemplar algorithms using a range of parallel programming systems. While that work was interested purely in GPU performance for its own sake, this new work has slightly different requirements stemming from the Mass Spectrometry side of the research.

This use of GPUs for general-purpose programming is known as GPGPU and largely took off with the advent of configurable graphics cards, allowing for great flexibility in the transformations which could be applied to image data. Systems were developed to convert between "regular" data sets and image textures such that these transformations could be applied for data processing, abstracting away from graphics APIs such as DirectX [9]. This is the case with the Accelerator library [7] from Microsoft, which was examined in the previous paper on this topic, and is included here only for comparison.

More recently dedicated GPGPU systems have been developed, bypassing the full image rendering pipeline to provide direct access to the massively parallel configurable cores. CUDA [10] from NVIDIA and OpenCL [11] from the Khronos group are major systems in this field, along with DirectCompute [12] from Microsoft. These utilise dedicated hardware on the GPU and provide C-like languages to enable low-level programming without specific graphics knowledge. Despite this, some target knowledge may still be required—getting the best performance requires writing code to take in to account hardware designs such as memory location, thread loads and device abilities.

1. Abstraction

Haskell is a strongly typed, lazy, functional programming language. Functions in Haskell are similar to mathematical functions in that for any given set of inputs the output is always the same as there is no global state to mutate (they are "pure"), any mutation is explicitly encapsulated and protected. The strong typing means that functions can only take parameters of exactly the right type and no implicit type conversion is done.

Functions are first-class citizens of the language: they can be passed around, composed, or otherwise operated on just like other data. They can also be partially applied, or "curried"—a function with two inputs may be provided with only one to return a new single parameter function instead of the final result. The laziness of the language means that no code is evaluated to a final result until it is required; multiple computations are chained together until required by some observable system—often the IO system, which provides a protected interface to the rest of the world and its state. How does this design allow for higher abstraction?

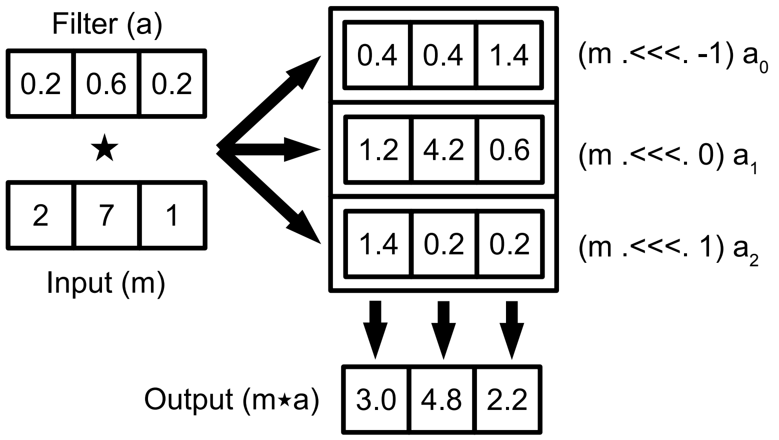

Figure 1. Basic convolution ($\star$) of an array m by filter a.

The three case studies examined share a common pattern—they all take a large data set and combine it with a much smaller data set in some way. For the Electrostatic Charge Map the large data set is a grid of discretised space and the small data set is a set of atoms. For UDWT the large data is a graph to be smoothed and the small data is a set of four related filters. For the convolution, the large input is an image and the small input is a Gaussian blur filter.

The most basic of these is convolution (also an important part of the later UDWT case study). The equation below shows the convolution ($\star$) of a 1D array m with a filter a of length A, for one array index x. Not shown is that $(x - H + i)$ is clamped to the size of m. To demonstrate the development of this code using PEGGY requires first developing the code in Haskell.

$$H = \frac{A-1}{2}$$

$$m_x \star a = \sum_{i=0}^{A-1} m_{(x-H+i)} a_i$$

An array shift operator $.\lll.$ can be written to adjust indices such that:

$$[1,2,3,4,5]\,.\lll.\,1 = [2,3,4,5,5]$$
$$[1,2,3,4,5]\,.\lll.\,(-2) = [1,1,1,2,3]$$

Using this, convolution can be implemented as an array shift followed by a multiplication. Rather than iterating over a data set and applying the filter to each index sequentially, this iterates over the filter sequentially, and applies the current filter value to each datum (optionally in parallel).

```
convolve :: [Float] → [Float] → [Float]
convolve filter input = sum convolvedParts
  where
    oneFilterElement shift mul = (input .⋘. shift) ^* mul
      -- Get the RADIUS of a filter.
    r = length filter ‘div‘ 2
    convolvedParts = zipWith (oneFilterElement input) [−r . .] filter
```

Here the ^∗ operator performs an element-wise multiplication of a list by a single value. This results in a list of part convolutions which need to be summed up as shown in Figure 1. The sum function takes a list of elements for which + is defined and sums them up.

This programming style allows the use of maps and folds (such as sum) over the data to generate the processing code, functions defined in the Haskell *Prelude* [13]. In C++ these patterns are usually expressed using loops:

```
// Map.
for (size_t i = 0; i != len; ++i)
{
    out[i] = in[i] * 7;
}
// Fold (sum).
result = 0;
for (size_t i = 0; i != len; ++i)
{
    result += out[i];
}
```

A C++ function to map over a filter would be possible, but not in the same way as in Haskell (e.g. with partial application). Additionally defining a sum between a C array and a single number is not possible

Converting the Haskell code to PEGGY code to be run on the GPU is now trivial. Only two small changes are required. Firstly the input data must be converted from a Haskell list to a PEGGY abstract array. Secondly, the type signature changes; however, the compiler can often infer the types—they are included here for clarity and may be omissible. This is made possible by the implementation of the relevant operators within PEGGY such as the array shift and multiply operators.

```
convolve :: [Float] → PYExpr Float TargetObsidian → PYExpr Float TargetObsidian
convolve filter oldInput = sum convolvedParts
  where
    oneFilterElement shift mul = (input .⋘. shift) ^∗ mul
    r = length filter ‘div‘ 2
    convolvedParts = zipWith oneFilterElement [−r . . r] filter
      -- Convert the old input to PEGGY GPU data.
    input = toArray oldInput
```

After all this abstraction, the remaining sums, maps and zipWiths relate purely to the algorithm being performed on the data, and not to the data itself. This brings the algorithm of interest to the forefront of the code, instead of the data on which it is being run. For data-parallel programming applications the data is important and vast, but ultimately what is being done with the data is more important. Arguably this is similar to the CUDA model of computation; a kernel function is written to identify and operate on a single output (which may involve multiple inputs), but notably data identification is still there.

As the case studies will show, several basic optimisation techniques in CUDA revolve around complex input data management, an issue now entirely hidden in PEGGY. to the question of performance addressed in this paper is the fact that the current PEGGY implementation does not pre-load and cache data.

2. Our Programming Environment

The "PEGGY"[1] Haskell library is designed as a write-once, run-everywhere system built on top of multiple parallel execution back ends. The Obsidian [5] library for generating CUDA code at run-time is the main back end of interest here, but there are also other back ends for running code sequentially or parallely on a CPU or for using Accelerator.

A basic expression in PEGGY has the type:

```
PYExpr domain target
```

domain is the type of the underlying data such as Float, and target is the back end on which to run the expression, for example TargetObsidian[2]. The example of adding seven to every element in a list is shown again below, this time using PEGGY. The list is converted to a 1D abstract array (toArray) and seven is added, yielding a new expression. The expressions are calculations to be performed in one go when specified. eval runs these on an as-yet undefined back end, and the result is converted back to a Haskell list.

```
input :: [Float]
input = [0 .. 9]
  -- Convert the data.
peggyInput :: PYExpr Float target
peggyInput = toArray input
  -- Create the expression.
peggyExpr :: PYExpr Float target
peggyExpr = peggyInput + 7
  -- Evaluate the expression.
peggyResult :: PYNative Float target
peggyResult = eval peggyExpr
  -- Convert back to a list.
result :: [Float]
result = fromArray peggyResult
```

At this point the type of peggyInput is PYExpr Float target—the domain is known but the execution back end is not, and without this vital information the code above will not run. The simplest way to set this is to explicitly set one of the types anywhere in the code:

```
  -- Convert the data for an explicit execution engine.
peggyInput :: PYExpr Float TargetObsidian
peggyInput = toArray input
```

Alternatively a function other than the generic eval can be used by specialising it to one execution engine, with it returning that back end's native result type:

```
runOnGPU :: PYExpr Float TargetObsidian → PYNative Float TargetObsidian
runOnGPU = eval
```

Using this method, the algorithm can be moved and only written once, being passed to multiple eval functions for different targets. result2 has been converted back from back end

[1] "**P**arallel **E**xecution, **G**enerally on a **G**PU, by **Y**_Less".

[2] In Haskell domain and target are type variables to be replaced with real types, TargetObsidian is a "phantom" type—it has no valid data and is only used to specify target specific data storage and expression formatting via type families.

return data to a Haskell list; result1 has not yet been converted, in this case it is still stored as a C array underneath because that is what GPU operations using CUDA return.

```
  -- This function adds seven to any input on any back end.
  -- "toArray" works on lists, "PYExpr"s and "PYNative"s too.
abstractExpression input = toArray input + 7
  -- Return a native array from Obsidian.
runOnGPU :: PYExpr Float TargetObsidian → PYNative Float TargetObsidian
runOnGPU = eval
  -- Return a Haskell list from sequential execution.
runOnCPU :: PYExpr Float TargetHaskell → [Float]
runOnCPU = fromArray ∘ eval
  -- Use the above generic algorithm.
inputList :: [Float]
inputList = [0 . . 9]
result1 = runOnGPU (abstractExpression inputList)
result2 = runOnCPU (abstractExpression inputList)
```

3. The Case Studies

3.1. Convolution

Section 1 showed the derivation of a PEGGY convolver, along with the equation for a 1D convolution. For a 2D input m and a 2D filter a, the filter surrounds any point of interest and would require A^2 operations. A "separable" convolution is one where separate 1D convolutions in the x then y dimensions of 2D data produces the same result as a 2D convolution. This requires only $2A$ operations total, but two separate convolution steps.

Generating code for a separable convolver merely involves calling the PEGGY convolution function twice, each time specifying shifts in different dimensions.

The CUDA implementation of this code (shown in Appendix **??**[3]) has separate functions for each dimension, each one with data caching for that dimension. Block-local shared memory is faster than global memory so this code pre-loads data for a section of the convolution, coalescing reads for efficient sequential memory access from aligned memory boundaries. Full details on these optimisations are available from NVIDIA [14].

The source lines of code metric, based on data from David A. Wheeler's "SLOCCount" program [15], are shown below.

Table 1. SLOC count for the Haskell and CUDA convolution.

Code	**SLOC Count**
Haskell Total	11
CUDA Total	116

3.2. Electrostatic Charge Map

An Electrostatic Charge Map is a representation of the electrical field surrounding a set of atoms. For a 3-dimensional grid consisting of k *points* and n *atoms* spaced inside it, each

[3]Supplementary materials are available from the WoTUG CPA website (`http://wotug.org/cpa2012/`). Full source code is available from `https://github.com/Y-Less/PEGGY`.

point involves a sum over the charge of every *atom* divided by that atom's distance to the current point. The full output involves $O(nk)$ operations. The calculations for one point are independent from every other point, making this an embarrassingly parallel computation. The effect of each atom on a single point could be done in parallel rather than the effect of one atom on every point. but in practice the number of points far outweighs the number of atoms.

The implemented case study only looks at a single 2D slice of a grid, calculating a full 3D field would be done through repeated applications of 2D slice calculations. This is shown in Haskell below:

```
type Expr = PYExpr Float TargetObsidian
  -- "Expr" type for brevity only.
chargeSpace :: [Atom] → (Expr, Expr, Expr) → Expr
chargeSpace atoms gridPoint = sum (map oneCharge atoms)
  where
    oneCharge a = charge a / distance gridPoint a
      -- Get the charge on a single atom.
    charge (Atom _ _ _ c) = c
      -- Get the distance between an the current atom and grid points.
    distance (gridX, gridY, gridZ) (Atom x y z _) =
      (diff gridX x ∗ diff gridX x) +
      (diff gridY y ∗ diff gridY y) +
      (diff gridZ z ∗ diff gridZ z)
      -- Convert the Atom pos to useful data.
    diff arr pos = arr − pos
```

The atom locations are randomly generated in advance within a 3D grid and passed to every implementation at initialisation, which is prior to code generation in the case of the Haskell code. This results in the atom data being either hard-coded in to the generated program code in the case of the Haskell and Accelerator targets, or stored in fast "constant" memory for the CUDA code. Optimisations in the convolution case-study came from the memory load patterns, with no significant memory reliance optimisations in this case must come from elsewhere.

The CUDA implementation (Appendix **??**), based on that from "Programming Massively Parallel Processors" [16], improves performance by using a single thread to calculate multiple grid points, amortising the y and z parts of the calculation across them all. Tuning resulted in six points being run per thread. Conversely, no optimisations at all were applied to the Haskell generated CUDA code beyond those implemented in Obsidian (which are minimal) and the CUDA compiler. Instead, any performance expected comes from the use of constant atom locations at compile time.

SLOC counts for the Haskell and CUDA Electrostatic Charge Map implementations are shown in Table 2.

Table 2. SLOC count for the Haskell and CUDA Electrostatic Charge Maps.

Code	SLOC Count
Haskell Total	17
CUDA Total	84

3.3. UDWT

UDWT is the Undecimated Discrete Wavelet Transform and is one algorithm used for denoising Mass Spectrometry data [17]. This algorithm builds on the simple convolution example by applying convolutions repeatedly (four times per level), but in a slightly different manner to that seen in the convolution case study. For this version, the convolution input data is wrapped modulo the length of the data instead of being clamped.

This transform applies a high-pass filter using convolution to get the upper half of all frequencies and a low-pass filter to get the lower half for level one. For subsequent levels this process is repeated on the lower half of the frequencies obtained at the previous level. The first level uses standard convolution as previously seen, but the higher levels (for the undecimated version of this transform) use larger filter gaps. A radius three filter in level one may operate on data elements 1, 2, 3; but in level two it will operate on elements 1, 3, and 5; and in level three on 1, 5, and 9.

After the multiple convolutions, the results are appended to give an overview of the full frequency space, and low intensity frequencies are removed through a basic cut-off filter. After this, the frequency-domain representation of the signal is transformed back to the original time-domain through a reversal of the deconstruction procedure using two new filters that act in opposition to the original two.

This algorithm is an important case study as it builds on the earlier convolution case study and more importantly (as mentioned), it is used in Mass Spectrometry data processing. This is a current area of research interest discussed further in the conclusion, bringing this work in line with the wider research context. It is, however, a recent addition to the case studies available, and as such is not as mature as the others—currently all results are for a single level of filtering only (the desired filtering level is an algorithm parameter). The CUDA implementation of the code (Appendix **??**) uses basic global data caching and load coalescing, but has not been tuned.

This case study does highlight several issues with the current system, but the main interest in PEGGY is as a preliminary feasibility study, not a robust development system. In this implementation the filtering threshold is based on the median value from the first level's high-pass output, which requires sorting the data—an operation not currently available in PEGGY. The full Haskell code (Appendix **??**) uses code from the Convolution case study modified to support convolution using inputs with different offsets. This code no longer uses $.\lll.$ because the convolution here wraps around on edge cases, it does not extend the boundaries.

Another problem it raised with the underlying system that does need addressing was the generation of multiple kernels. Each high-pass filter output is a distinct result, calculated independently on a GPU; but the higher the level, the more low-pass filters need to be run prior to calculating the desired high-pass output. This resulted in monolithic generated code due to loop-unrolling and failed to even compile above level two.

Table 3. SLOC count for the Haskell and CUDA UDWT.

Code	**SLOC Count**
Haskell Total	31
CUDA Total	186

4. Performance Analysis

To facilitate running these tests, a generic framework was developed to abstract as much common code as possible. This includes setup and tear-down for multiple hardware targets; algorithm common code; and timing, repeat runs, and verifications for each target. This was run on a Windows 7 machine with an Intel Core 2 Extreme 3.0GHz processor (X9650), 8Gb of RAM, and an NVIDIA GeForce GTX 460 with 2Gb of graphics RAM (Palit Sonic).

The expert CUDA code, and the Accelerator and Haskell codes are all written and compiled in advance, but the CUDA code generated by Haskell is created and compiled at exper-

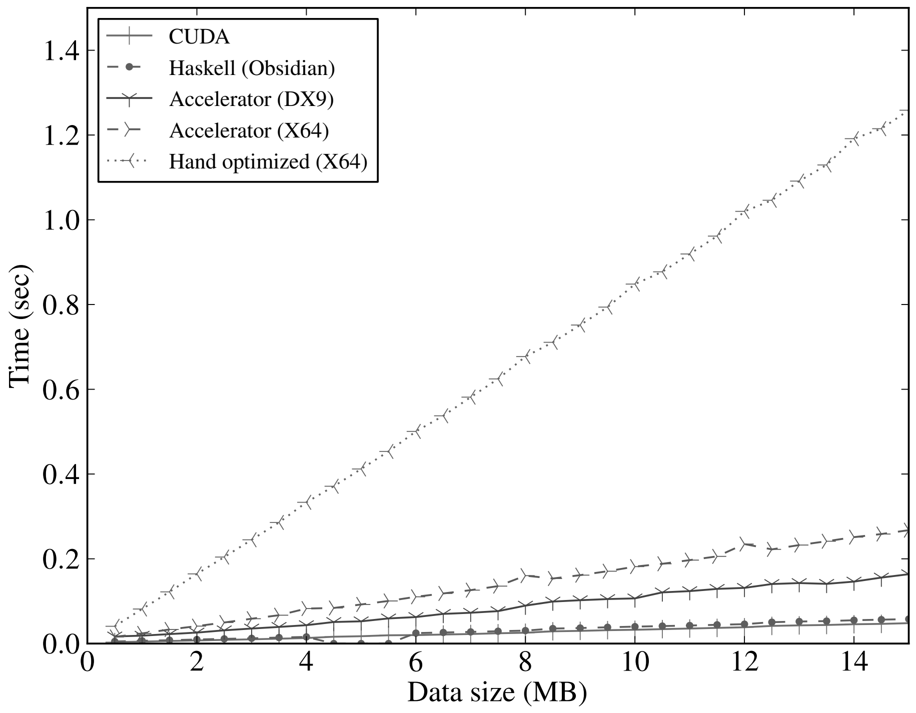

Figure 2. Graph of all targets in the convolution case study.

iment run-time. PEGGY was originally designed for use purely in Haskell on Haskell data, but for the purposes of timing it instead generates and compiles CUDA code, then passes the object file to the test framework for multiple timed runs using the same data as all the other implementations.

Figures 2 and 3 show firstly all the results for all the convolution targets and then zoom in on the bottom of the graph to show just the three GPU targets. "Accelerator (DX9)" targets the GPU through DirectX 9, and "Accelerator (X64)" targets multicore SSE2 instructions. Figure 4 shows the results of the Electrostatic Charge Map on the GPU only. The GPU has been previously shown to significantly outperform the CPU in this experiment and these new results merely confirmed that. Figure 5 shows the results for a level one UDWT filter on all targets.

The first graph (Figure 2) shows how previous conclusions that the DX9 target significantly outperformed sequential code were justified. Drilling down in to the second graph shows just how much faster both the CUDA and Haskell code are than the Accelerator code, and show a gap between those two that was almost indistinguishable in the first. While previous results were questionable given the obvious differences between the CUDA and DX9 targets upon zooming, the new results are much more satisfying. There is a repeatable anomaly around the 5Mb mark, but at this point the cause is unclear. Despite this, all other results are known to give accurate answers when verified against reference implementations.

The UDWT results in figure 5 are the most interesting from a development point of view, and show the results from the parallel part of the most complex algorithm (again part of this code is not done in Haskell, but this is due to the modifications to use the generated code within the test framework). The CUDA version of this algorithm was not as extensively profiled and tweaked as the CUDA versions of Convolution and Electrostatic Charge Map were—some effort was made to coalesce reads and cache data, but edge cases where the loading was wrapped modulo the input length were not optimised. After all this, the CUDA and Haskell results lines are neck and neck—the bottom line on the graph in fact shows both.

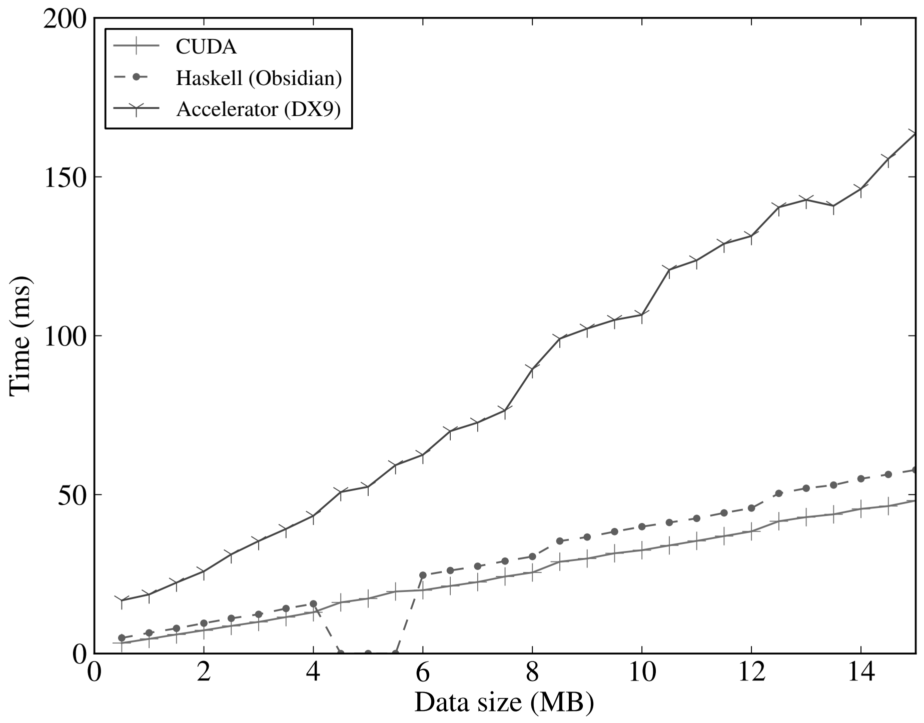

Figure 3. Graph of GPU targets in the convolution case study.

5. Conclusions And Future Work

The introduction posed the following questions:

1. Is it possible to build a language on top of multiple back ends simultaneously, in a domain specific language?

For the domain of array-based computations, a library was developed (Section 2) that could be shown to be translatable from Haskell (Section 1), at least for certain classes of problems mapping to its whole array representation of data. Section 3 continued this demonstration, but also showed that there was still a long way to go with the abstractions—especially for the more complex algorithms (which are of course of much greater interest).

2. Does this abstracted code produce code a run-time with adequate performance from concise problem descriptions?

Each of the case studies presented their Haskell implementations and provided references to their much longer CUDA equivalents (with line counts). The graphs in Section 4 showed how these implementations performed against each other. The Haskell code in all cases was much shorter and written in a target language of interest. While none of the results were faster in Haskell than CUDA, despite additional compile-time knowledge of the input data, one result was the same speed. The other two were fifteen and three hundred per cent slower in Haskell; this is a large number, but the tests also showed just how much faster than the sequential implementations they were.

3. Can this be done again for other languages?

The answer to this is believed to be yes, and will frame all future work. This revolves around the development of a domain specific language (DSL) through which experts in Mass Spectrometry (but not necessarily computer programming) could describe algorithms as they were needed and have them run in a massively parallel manner. The two main targets for

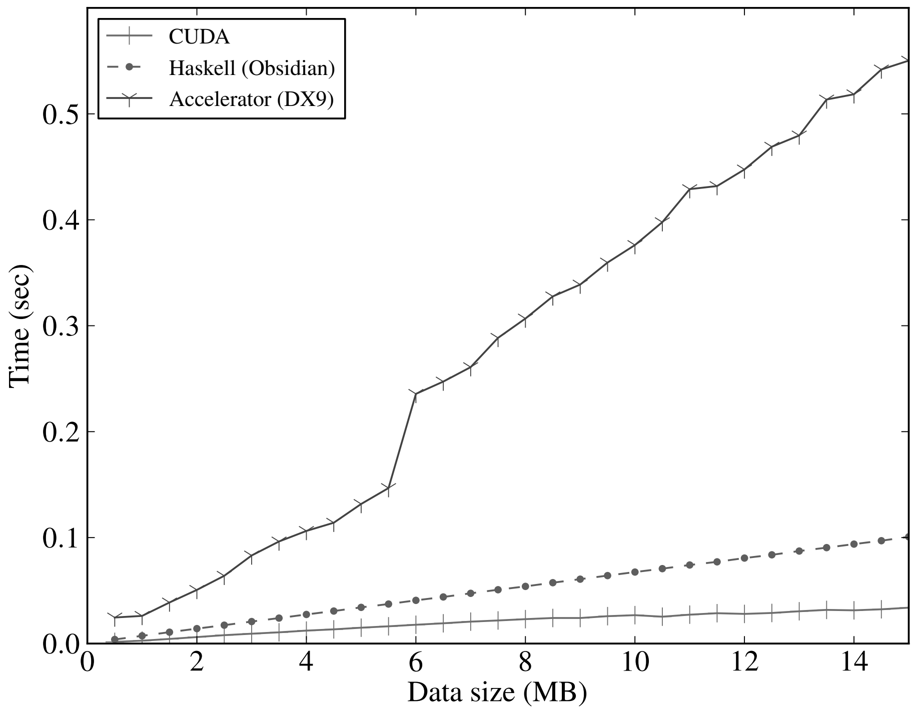

Figure 4. Graph of GPU targets in the Electrostatic Charge Map case study.

this work are the development of the high-level DSL, using idioms familiar to chemists, and the translation of this code down to data-parallel devices such as graphics cards containing graphics processing units (GPUs).

Mass Spectrometry is an analytical process which can determine the makeup of a chemical compound or mixture of compounds. This process involves hardware (a Mass Spectrometer (MS)) designed to take an input of chemicals and produce an electronic representation of the input, and software to derive the original input from this [18]. For example, an ESI-TOF-MS[4] passes a chemical liquid through an electric field to impart a charge on to the constituent molecules, another field is then used to accelerate these charged particles towards a detector. A given mass with an imparted charge of ± 1 (or twice that mass with a charge of ± 2) will take a consistent time to reach the detector, the number of strikes detected after that time determine the intensity of that “mass-to-charge” ratio in the original mixture.

The output from the hardware is a noisy graph of mass-to-charge ratio against intensity (a spectra). The software then takes this raw data and attempts to determine what combination of input materials could have produced the observed output. This is complicated by the fact that input compounds are subject to chemical reactions inside the hardware, resulting in detected chemicals that were not present in the original mixture.

The data from a single MS run can be many gigabytes and the analysis algorithms can take a long time to run, but can scale linearly with the number of available processors [19]. There are many published algorithms to analyse this data, specialised to different workflows or hardware, and with different selectivity and sensitivity properties. However, many of these algorithms are not used—the theory behind them exists, but either there is no software implementation of them at all, or their implementations are not compatible with existing software [20] stacks presently in use.

[4]Electro-Spray-Ionisation Time-Of-Flight Mass Spectrometer

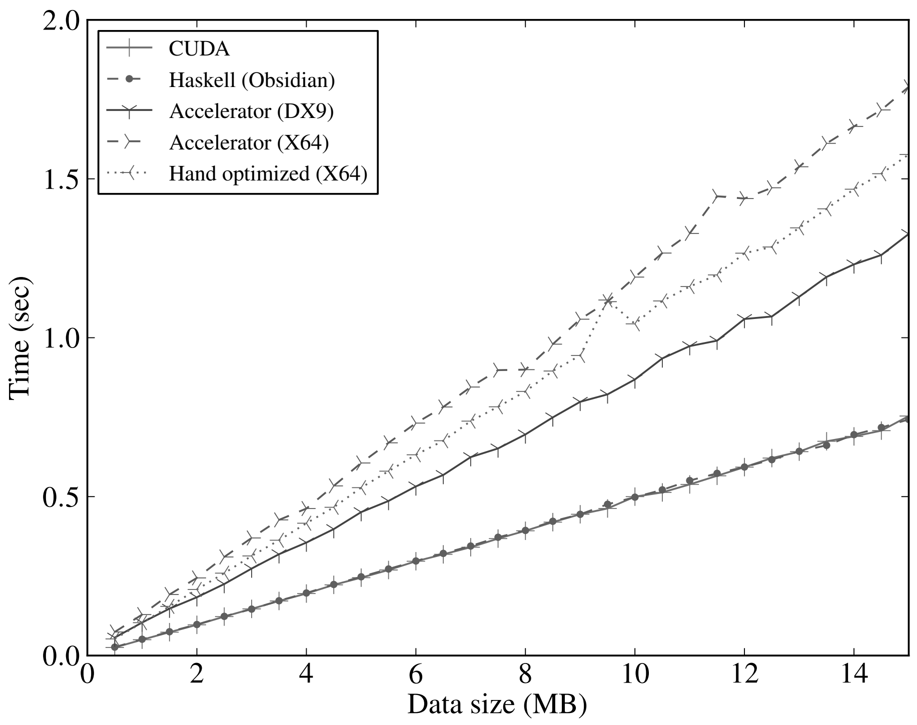

Figure 5. Graph of all targets for a level 1 UDWT filter.

These requirements should explain some of the design decisions made during this work, and while the PEGGY DSL presented above was a little lacking in some areas, the results are still believed to justify this Mass Spectrometry research direction.

Acknowledgements

The authors would like to thank Satnam Singh, now at Google, who assisted greatly in the earlier stages of this research project.

References

[1] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. *SIGPLAN Not.*, 34(1):174–184, September 1998.

[2] Krzysztof Czarnecki, John ODonnell, Jrg Striegnitz, and Walid Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-25935-0_4.

[3] Andy Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM.

[4] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. *SIGPLAN Not.*, 45(11):67–78, September 2010.

[5] Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 156–173. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-24452-0_9.

[6] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[7] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS XII*, page 325, San Jose, California, USA, 2006.

[8] Alex Cole, Alistair McEwan, and Satnam Singh. An Analysis of Programmer Productivity versus Performance for High Level Data Parallel Programming. In Peter H. Welch, Adam T. Sampson, Jan Bækgaard Pedersen, Jon M. Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *33rd Communicating Process Architectures Conference (CPA 2011)*, volume 68, pages 111–130, 2011.

[9] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[10] NVIDIA. *NVIDIA CUDA Programming Guide 3.2*. 2009.

[11] Aaftan Munshi. *The OpenCL Specification v1.2*. Khronos Group, 2011.

[12] Microsoft Corporation. Compute Shader Overview. `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx`, 2012. Accessed 07/2012.

[13] Simon Peyton Jones, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 Language and Libraries. `http://www.haskell.org/onlinereport/`, 2002. Accessed 06/2012.

[14] Victor Podlozhnyuk. Image Convolution with CUDA. `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf`, 2007. Accessed 06/2012.

[15] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2004. Accessed 06/2012.

[16] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. Morgan Kaufmann, 2010.

[17] Kevin R. Coombes, Spiridon Tsavachidis, Jeffrey S. Morris, Keith A. Baggerly, Mien-Chie Hung, and Henry M. Kuerer. Improved peak detection and quantification of mass spectrometry data acquired from surface-enhanced laser desorption and ionization by denoising spectra with the undecimated discrete wavelet transform. *PROTEOMICS*, 5(16):4107–4117, 2005.

[18] E. De Hoffmann and V. Stroobant. *Mass Spectrometry: Principles and Applications*. J. Wiley, 2007.

[19] David N. Perkins, Darryl J. C. Pappin, David M. Creasy, and John S. Cottrell. Probability-based protein identification by searching sequence databases using mass spectrometry data. *ELECTROPHORESIS*, 20(18):3551–3567, 1999.

[20] Alexey I. Nesvizhskii. A survey of computational methods and error rate estimation procedures for peptide and protein identification in shotgun proteomics. *Journal of Proteomics*, 73(11):2092 – 2123, 2010.