Declarative Programming Techniques for Many-Core Architectures

Satnam Singh
Microsoft Research
7 JJ Thomson Avenue
Cambridge, Cb3 0FB
United Kingdom
satnams@microsoft.com

Abstract

Future manycore architectures are likely to have heterogeneous computing resources which will include conventional CPUs as well as variants of today's GPUs and reconfigurable logic like FPGAs. Many of the techniques that the reconfigurable computing community has championed will find new applications in mainstream applications. One challenge posed by such manycore architectures is the requirement to target multiple parallel computing resources from a single description (or code). This paper proposes a design methodology for the specification and implementation of data parallel computations that can be mapped to either circuits on FPGAs or pixel-shader code on GPUs from exactly the same description. These descriptions exploit higher order combinators and polymorphism to provide powerful glue for composing data parallel descriptions in a generic way. We present two implementations of these combinations: one in F# which is a variant of the ML functional language and the other in the C# language which uses generics and C# delegates to achieve the effect of higher order combinators. We present the results of implementations that execute on Virtex FPGAs and ATI graphics cards.

1. INTRODUCTION

Reconfigurable computing fabrics like FPGAs stand to enjoy an important role in future manycore architectures that will comprise heterogeneous processors, programmable data paths and special hardware acceleration architectures (like FPGAs). In this paper we argue for the need for high level data parallel descriptions that can be compiled to FPGA-based hardware; to GPUs; or to multi-threaded software for execution on multiple CPU cores. We predict that future chips will have a great variety of parallel computing resources which will be used as market differentiators for price and performance. However, software will need to execute on each of these configurations, albeit with different performance characteristics. This will require components of software systems to be designed in such a way as to allow execution on CPUs, GPUs (and related parallel data-path oriented architectures) and FPGAs. Designing a different implementation for a module for each target is impractical and not all possible targets may be known at compile time. Instead, we argue that it is far more productive to cast data parallel descriptions in a form that can be readily mapped to a variety of data parallel computing targets including multi-core processors, GPUs and FPGAs. This allows data parallel descriptions to target different kinds of parallel computing resources and it also helps to dynamically migrate calculations from one resource to another. We advocate the view that many of the techniques and principles that have been applied to run-time reconfiguration will also find mainstream applicability in future manycore architectures.

The contribution of this paper is a data parallel formalism based on higher order combinators that provides a highly composable way of building up re-targetable data parallel descriptions. This paper illustrates the proposed approach with a sorting example. The technique we propose is largely language neutral and we outline two implementations in very different languages: one in F# which is a variant of the ML functional language which is interoperable with the .NET framework on the other in C# which is a modern object-oriented language.

Other researchers such as Cheung, Luk and Mencer have presented results or undertaken work which show that GPUs are already an interesting parallel computing resource which can outperform FPGAs for certain tasks. The Brook [1] system from Stanford also shows how viable GPUs are as general purpose computing engines. The objective of this paper is not to present further evidence about the potential of GPUs but instead to show how GPUs and FPGAs can be targeted from a single description. We believe that the techniques described in this paper have an important role to play in the search for design methodologies that can target the heterogeneous parallel computing resources of future manycore architectures.

2. FUTURE MANYCORE ARCHITECTURES

The path of least imagination for future processor design is to stamp out many copies of a CPU die and connect them together with a coherent memory. However, it is far from certain that this is a sensible approach for a variety of reasons including the lack of scalability of cache coherence, lack of parallel programming models and power consumption. Instead, we propose that future manycore architectures may look like the Metropolis system described in Figure 1.

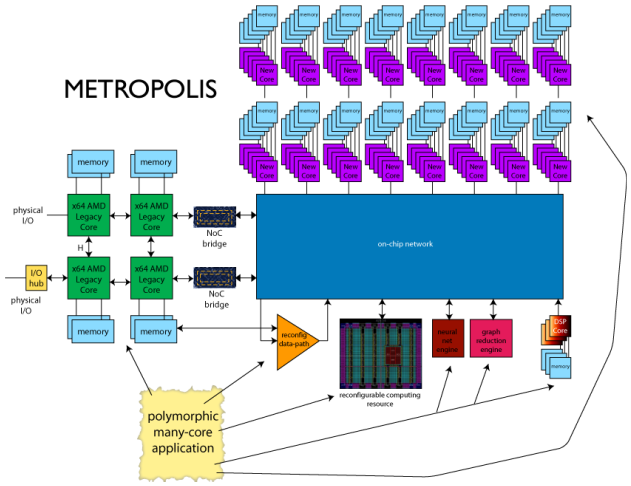


Figure 1: A potential manycore architecture

Such an architecture comprises conventional legacy processors, many smaller processors, an on-chip network, an FPGA-like computing resource and an evolution of today's GPUs in the form of a highly programmable parallel data-path. A key requirement for such an architecture is the ability to program the various computing resources from a single 'polymorphic' description that can take on different forms depending on the implementation target.

3. Data Parallel Descriptions

Data parallel descriptions can take many forms varying from explicit and fairly static descriptions of data-flow computing resources to very indirect descriptions in an imperative language which many involve significant analysis to extract parallelism from nested loops and complex array indexing. In this paper we advocate descriptions from the former category because these are more amenable for translation into circuits for realization on FPGAs (or other reconfigurable computing resources); pixel-shader code for GPUs; or for multi-core CPUs.

We advocate data parallel descriptions which have the following characteristics:

Higher order. This means that the data parallel descriptions are made up of elements that take computations as inputs and return computations as results. This is an important ingredient which is needed to allow us to devise a highly *composable* data parallel design methodology. An example of a higher order operation is one that takes an algorithm for solving a problem, some problem set and then returns a new algorithm that uses two parallel instances of the original algorithm on the given input. Examples of such higher order functions appear later in this paper.

Polymorphic. The data parallel descriptions should be general enough to range over many kinds of input which allows such descriptions to be applicable to many kinds of implementation target. Examples in this paper show how we exploit polymorphism in F# and generics in C# to achieve this effect.

Data-parallel. The descriptions should make it easy to spot how calculations are applied in parallel to a given input stream. Either the same operation is applied to each element of a stream (in SIMD style) or different operations are performed for each input. Our approach supports either style.

The remainder of this paper presents an example of a parallel sorter which can be implemented on FPGAs, GPUs or multi-core CPUs from the same description based on the principles we have just described.

4. Data Parallel Code in F# and C#

In this paper we present higher order polymorphic and recursive data parallel descriptions in two languages: F# and C#. In F# we make use of the polymorphic type system and we define useful combinators to give what looks like a domain specific language for describing data parallel networks. In C# we exploit the generic facility to achieve a similar effect to polymorphism and we make heavy use of parameterized static methods, delegates and anonymous delegates which are used to define combinators for parallel descriptions.

In F# we provide a serial composition operator which has a circuit simulation behavior given by this definition:

```
let (>->) f1 f2 i = f2 (f1 i)
```

This lets us write `A >-> B` to mean connect the output of computation (or circuit `A`) to the input of computation (or circuit) `B`. We also introduce a useful library of list operations for tasks like halving and zipping lists.

Higher order functions are a primitive feature of F# but in C# we need to define the type of higher order parameters using delegate types. We also use generics to help us define a very general notion of a function that takes one value and returns a result of a possibly different value:

```
public delegate T2 unaryFn<T1, T2>(T1 v);
```

We can now use the type `unaryFn` to describe higher order parameters in C#. This describes a function (or method) that takes as input some parameterized type T1 and returns a parameterized type T2.

5. A Parallel Sorter

This section describes how to build a data parallel sorter circuit using butterfly networks which are carefully placed to ensure high performance. The sorter circuit is made by recursively merging the results of sub-sorts. A top-level schematic of the circuit that we present in this section is shown below.

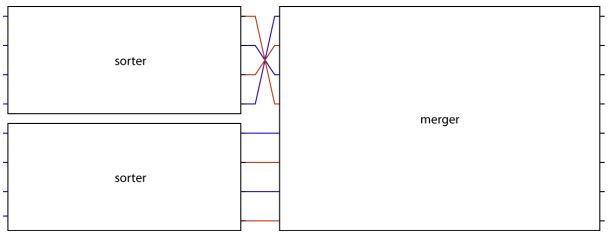
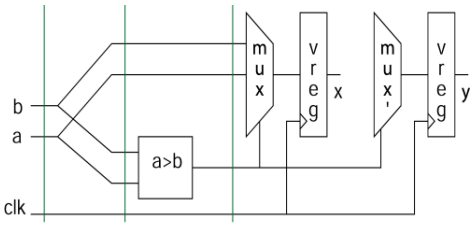


Figure 2: The recursive structure of the sorter

The merger that we present is bitonic which requires the first half of the input list to be increasing and the second half decreasing (or vice versa). The result of the top sorter is reversed to accommodate this requirement.

Given the ability to sort two numbers and the diagram above we have a recursive formula for making sorters of any size. A data parallel description of the two sorter is:



twoSorter clk = fork2 >=> fsT comparator >-> condSwap clk

The FPGA the implementation of the two sorter is given below:

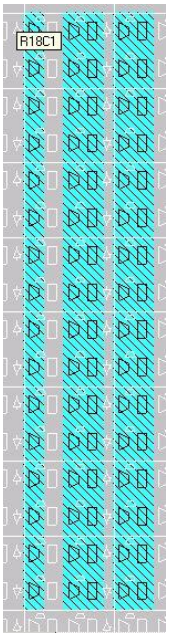


Figure 3: The FPGA layout of the two-sorter.

For the CPU version the two sorter simply takes streams of values and sorts them directly. The GPU version used a system called Accelerator [7] from Microsoft to define a two-sorter component:

```
public static List<FloatParallelArray>
sort2(List<FloatParallelArray> l)
{ FloatParallelArray cf = l[0] - l[1];
  FloatParallelArray o1 =
    FloatParallelArray.Cmp(cf, l[1], l[0]);
  FloatParallelArray o2 =
    FloatParallelArray.Cmp(cf, l[0], l[1]);
  List<FloatParallelArray> r
    = new List<FloatParallelArray>();
  r.Add(o1);
  r.Add(o2);
  return r;
}
```

This code results in a pair of pixel shaders being generated (one for each output):

```
s_2_0
dcl_2d s0
dcl_2d s1
dcl t0.xy
texld r0, t0, s0
texld r1, t0, s1
sub r2, r0, r1
cmp r0, r2, r1, r0
mov oC0, r0
```

```
ps_2_0
dcl_2d s0
dcl_2d s1
dcl t0.xy
texld r0, t0, s0
texld r1, t0, s1
sub r2, r0, r1
cmp r1, r2, r0, r1
mov oC0, r1
```

Each pixel shader takes the same two streams as input (s0 and s1). The input streams are loaded as textures into registers r0, r1, and r2. After the two-sorter computation (involving a subtraction and a comparison) values are streamed back to the texture memory. Larger sorters result in larger pixel shaders which do not involve so many round-trips to memory. We use texture memory for the sorter input streams [8].

A merger called Batcher's Bitonic Merger can be made by using a butterfly of two sorters. Here is an example of a specific butterfly network of two sorters (written as 2S) which merges eight numbers:

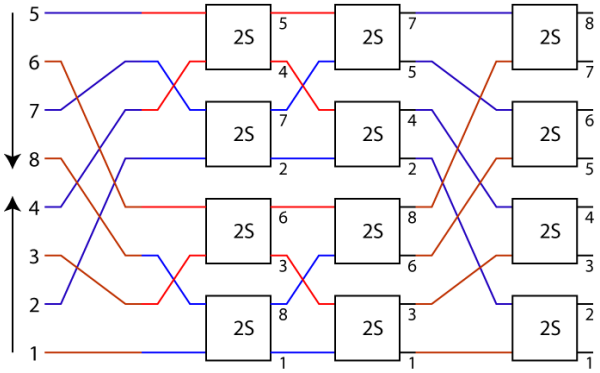


Figure 4: A bitonic merger

To help describe such butterfly networks in a data parallel way a few useful circuit combinators are introduced. From the top level description we see that a reverse operation is required and we can simply reverse function of the host language:

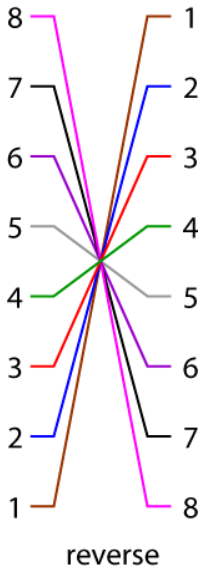


Figure 5: The reverse combinator

Another very useful wiring combinator is called riffle and an instance of it is shown below:

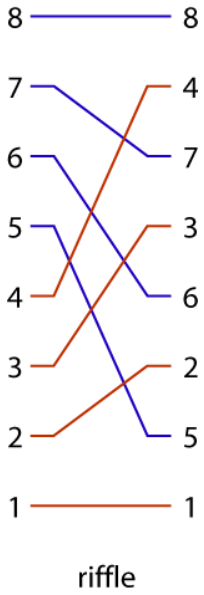


Figure 6: The riffle combinator

This wiring combinator interleaves the odd and even elements of the input list (shown on the left). It can be defined in F# as:

```
let riffle = halveList >-> zipList >-> unPair
```

The halve function splits a list into two halves which are returned in a two element tuple. The ziP combinator takes a pair of lists and returns a new list of pairs by associating each element in the first list with the corresponding element in the second list. The unpair function then flattens this list of pairs into a list.

The definition of riffle in C# makes use of a generic List collection to hold the elements of the list (rather than the polymorphic list in F#). We define riffle as a static method which most closely resembles a function and the definition makes use of other static methods for halving, zipping and unpairing lists.

```
public static List<T> riffle<T>(List<T> l)
{ return  unpair(zip(halve(l))); }
```

It is also useful to be able to perform the inverse function of riffle called unriffle. This circuit can be thought of as the reflection of the riffle circuit along a vertical axis as shown below.

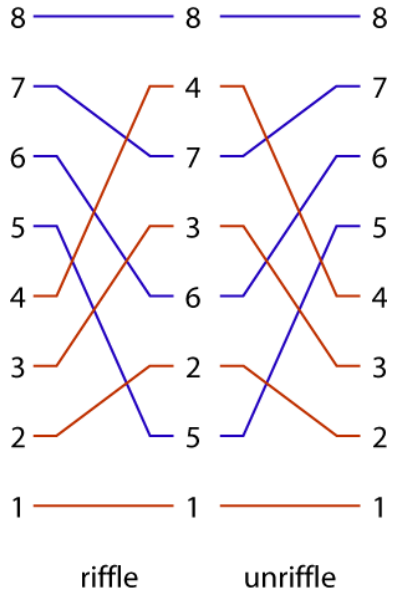


Figure 7: Unriffle

In F# this is described as:

```
let unriffle = pair >-> unzipList >-> unhalveList
```

and there is a corresponding definition for C#.

Sometimes a bus containing *n* elements is processed by using two copies of a circuit such that the first copy of the circuit operates on the bottom half of the input and the second copy of the circuit operates on the top half of the input as shown below for a four input bus:

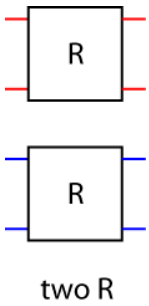


Figure 8: The two higher order combinator

The combinator that performs this task is called **`two`** and is easily defined in F# in terms of the more primitive **par** combinator:

```
let two r = halve >-> par2 r r >-> unhalve
```

The par combinator has different interpretations for different targets. For CPU targets it spwans off two threads each of which compute R for their input. For GPUs it tries to create a pixel shader which has the code for R which is applied in parallel to two streams. For FPGAs it causes two R circuits to be instantiated.

In C# the **two** higher order combinator can be defined using a delegate type:

```
public static List<T2> two<T1, T2> (unaryFn<List<T1>, List<T2>> r, List<T1> l)
{ return unhalve(par2(r, r,halve(l))); }
```

Another combining form that uses two copies of the same circuit is **`ilv`** (pronounced "interleave"). This combinator has the property that the bottom circuit processes the inputs at even positions and the top circuit processes the inputs at the odd positions. An instance of **`ilv`** R for an eight input bus is shown below.

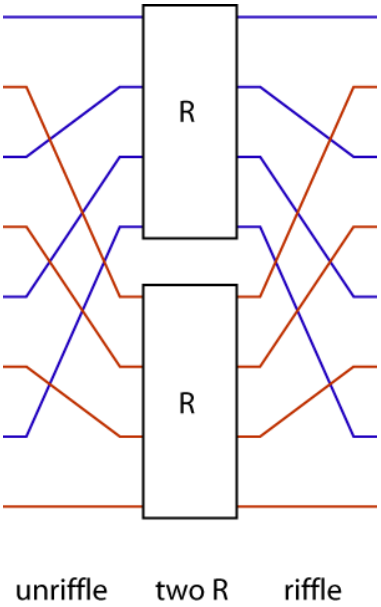


Figure 9: The **ilv** higher order combinator

The **ilv** combinator can be defined by noticing the it is the composition of an unriffle, two R and riffle. In F# this combinator is defined as:

```
let evens f = chop 2 >-> map f >-> concat
```

In C# the definition once again makes use of a delegate:

```
public static List<T2> ilv<T1, T2>(unaryFn<List<T1>, List<T2>> f, List<T1> l)
{ return riffle(two(f, unriffle(l))); }
```

The evens combinator chops the input list into pairs and then applies copies of the same circuit to each input. The argument circuit for evens must be a pair to pair circuit. An instance of evens two_sorter over an eight input list is shown below.

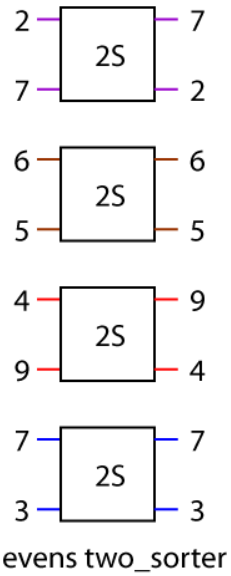


Figure 10: The evens higher order combinator

This combinator is defined in F# as:

```
let evens f = chop 2 >-> map f >-> concat
```

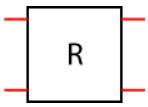
Using the combinators shown above we can now describe in F# a butterfly network of some circuit r (such that r is a pair to pair circuit or calculation):

```
let rec bfly r n =
   match n with
   1 -> r
 | n -> ilv (bfly r (n-1)) >-> evens r
```

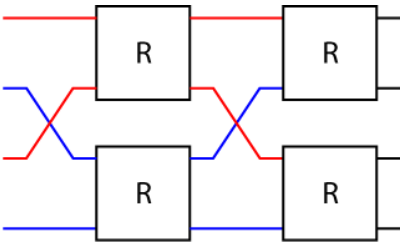
The C# description requires the use of an anonymous delegate to achieve the effect of the partial application in the description shown above.

```
public static List<T> bfly<T>(unaryFn<List<T>, List<T>> r, List<T> l)
{ if (l.Count == 2)
   return r(l);
  else
   return evens(r, ilv<T, T>(delegate(List<T> i)
                { return bfly(r, i); }, l));
}
```

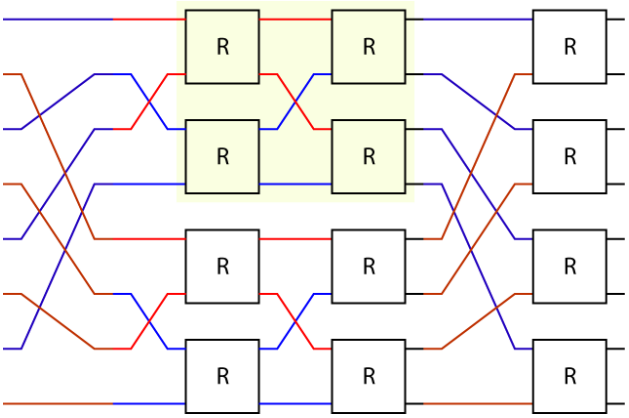
This is a recursive butterfly description. Here is a picture of bfly r 1:



This makes sense in the case of a two sorter since a butterfly of size 1 has 2 inputs which can be sorted by a single two sorter. The layout for bfly r 2 is:



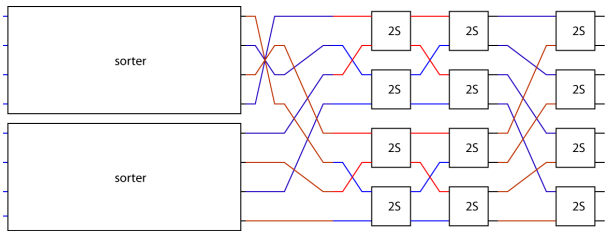
The left hand side of this picture shows an interleave of R and the right hand side shows evens R. The layout for bfly r 3 is:



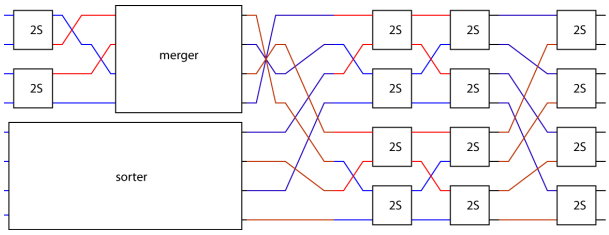
Note that a sub-butterfly of size 2 has been identified with a pale background. It can be instructive to unfold the bfly r 3 expression and then try and spot where the various combinators occur in the picture.

```
bfly r 3
  = ilv (bfly r 2)) >-> evens r
  = ilv (ilv r >-> evens r) >-> evens r
```

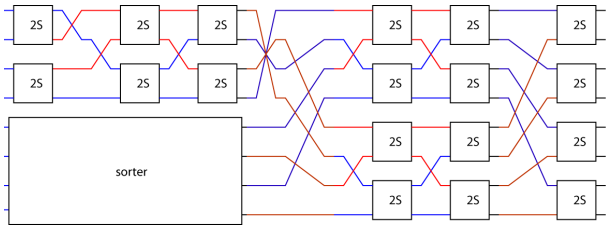
To make a merger all we need to do is to instance this butterfly with a two sorter. Here is a picture of bfly r two_sorter shown before. This solves the right hand side of the sorter architecture since bfly two_sorter makes a bitonic merger:



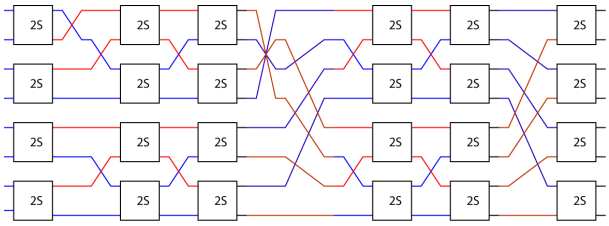
The two remaining sorters can be recursively decomposed using exactly the same technique used to decompose the top level sorter. For example, the upper sorter can be implemented by using a merger (shown on the right) and then sorting the two sub-lists. Since each sub-list contains just two elements we get to the base case of the recursion and deploy a two sorter.



But how is the merger realized? As before, it is just a butterfly of two sorters, in this case bfly 2 two_sorter:



Applying the same technique to the lower sorter gives the complete architecture for a size 3 sorter (i.e. 2^3 inputs = 8):



Although it is not at all obvious this circuit sorts eight numbers it has been systematically derived from a simple procedure which can be codified in F# as:

```
let rec bsort n =
  match n with
  1 -> sort2
  | n -> two (bsort (n-1)) >-> sndList rev >->
         bfly sort2 n
```

This description says that a sorter of degree 1 (i.e. 2 inputs) can be made using a two sorter. A larger sorter is made by using two small sorters, then reversing the result of the upper sort, and then merging these sub-sorts using a butterfly of two sorters. Note that this sorter description is parameterized on the specific sorter to be used and this a key feature which allows this description to be used for so many different targets.

The layout of a 32-input sorter on a Virtex XC2V3000 part is shown below. The netlist generators infers layout information from the combinators used to compose the data parallel description and this information results in a densely packed layout that is faithful to the layouts shown for the butterfly networks. This implementation sorts over 165 million 16-bit numbers arriving in 32 streams per second.

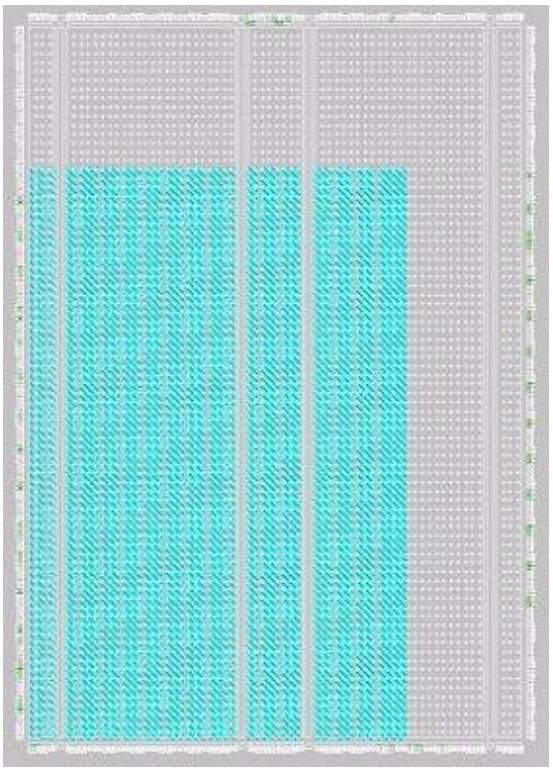


Figure 11: Implementation of sorter on Xilinx XC2V3000

A GPU implementation of several sorters using exactly the same higher order data parallel operations shows that for a given card there seems to be an 8X degree of parallelism:

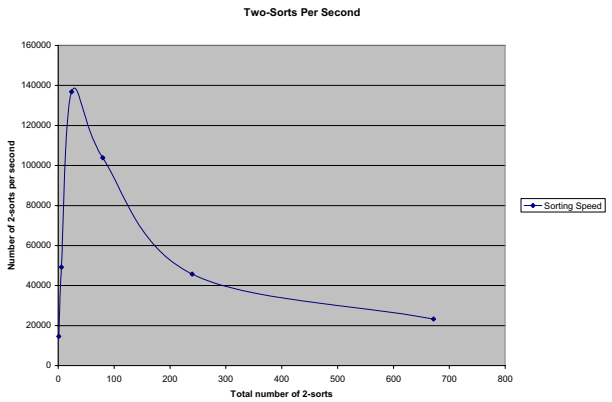


Figure 12: GPU Performance

This graph shows work (the number of two-sorts needed) on the x-axis and parallelism on the y-axis (the number of two-sorts per second) and includes the time required to compile and JIT the code from C# or F# into pixel shader code as well as memory transfer to and from the GPU from the CPU. The graphics card used was an ATI Radeon 9600 and we found similar results on other cards. Each stream contained 2048 values. It shows that for 8 inputs there is roughly an 8X improvement in performance which suggests that sorters of this size are well suited to the pixel shaders on the chard. As the size of the inputs grows we believe the need to transfer memory to and from the card becomes too large an overhead.

6. Related Work

Mencer has developed the ASC [6] stream based system which also seems very suitable for compilation to multiple targets including GPUs and FPGAs.

GPUTeraSort [1] at is another example of a GPU-based sorter which also used a bitonic sorter which works on large databases composed of billions of records and wide keys. Benchmarks have shown that the GPU-based sorter outperforms high-end workstations. Other interesting applications for data-parallel calculations for FPGAs and GPUs include biological sequence analysis [1]. Linear algebra is also another domain that seems suited to GPU [2][4].

7. Future Work

The approach we used for programming GPUs involved using a DirectX-based system which makes use of proprietary information about how to exactly configure the pixel shaders (we do not use the vertex shaders). We now propose to prototype our own designs inspired by GPU hardware onto FPGA devices but with an architecture and programming model which is better suited for more general purpose data parallel computing.

The kind of restrictions that we would like to overcome are things like: limited numbers of registers; slow bandwidth back o the main processor; and more direct SIMD-style descriptions.

8. Conclusions

This paper shows the viability of using data parallel descriptions based on combinators that exploit higher order functions and polymorphism to give descriptions which can be effectively mapped onto FPGAs and GPUs. The ability to target different parallel computing devices from the same description will become more important as future manycore architectures take on heterogeneous computing resources onto the same die as processors. We believe that architectures based on GPUs and FPGAs will be prime candidates for the manycore era and the programming model described here will help to fully exploit future computing architectures.

REFERENCES

[1] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. SIGGRAPH, 2004.

[2] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix- multiplication. Graphics Hardware, 2004.

[3] Naga K. Govindaraju; Jim Gray; Ritesh Kumar; Dinesh Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. Microsoft Technical Report MSR-TR-2005-183. December 2005.

[4] J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. SIGGRAPH, 2003.

[5] O. Mencer, D. J. Pearce, L. W. Howes, and W. Luk. Design space exploration with A Stream Compiler. IEEE International Conference on Field Programmable Technology (FPT), Dec 2003.

[6] O. Mencer. ASC, a stream compiler for computing with FPGAs. IEEE Transactions on CAD, 2006.

[7] David Tarditi; Sidd Puri; Jose Oglesby. Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism. MST Technical Report MSR-TR-2005-184. December 2005.

[8] S. Venkatasubramanian. The graphics card as a stream computer. SIGMOD-DIMACS Workshop on Management and Processing of Data Streams, 2003.

[9] G. Voss, A. Schröder, W. Müller-Wittig, B. Schmidt, Using Graphics Hardware to Accelerate Biological Sequence Analysis, IEEE Tencon 2005, Melbourne, Australia, 2005