Optimising Obsidian Programs Through Derivations

Joel Svensson

June 2008

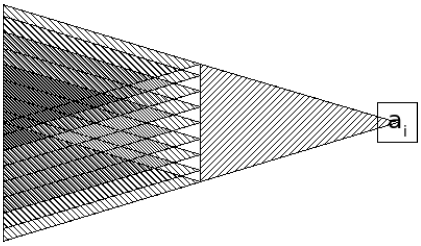
1 Introduction

Obsidian is an embedded language for data-parallel programming. In Obsidian data-parallel programs are written in a style similar to the one used in Lava [2]. Lava is an embedded language for hardware description and verification developed by Mary Sheeran and Koen Claessen at Chalmers and Satnam Singh at Xilinx. The main idea of Obsidian is to use combinators to specify the parallel computations and from these high level descriptions generate C code. As a target platform we have chosen NVIDIA GPUs (Graphics Processing Units) using *CUDA*. CUDA is NVIDIA's system for general purpose data-parallel programming on their high end GPUs. The programmers view of the GPU using CUDA is a device capable of executing a high number of threads in parallel. These threads are serviced by a number of SIMD processors each containing 8 processing elements [1].

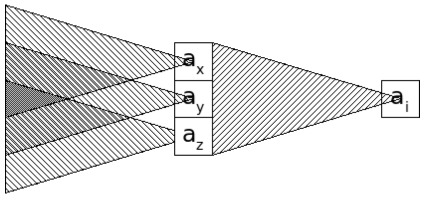
2 Obsidian

Obsidian programs describe computations on arrays. An array has the type `Arr a`. The operations on these arrays are implicitly parallel. In the code generated by Obsidian each element in the target array is computed by one thread.

An important function in Obsidian is the `sync` function. This function assigns work to threads. Figure 1 illustrates the role of `sync`. The figure shows `sync` together with two functions `f` and `g`. Function composition `->-` is also used in the figure. The composition operator has been given this appearance to look like the one in Lava. Here `(f ->- g) x` means `g(f(x))`. There is a shorthand available so that `f ->- sync ->- g` can be written `f ->>- g`.



(a) `f ->- g`



(b) `f ->- sync ->- g`

Figure 1: Shows the difference between (`f ->- g`) and (`f ->- sync ->- g`). The shaded triangles represent the expressions for f and g. In figure b the redundant computations from figure a is avoided by storing the result of f in memory.

2.1 Programming example

The first example program, `pAdd`, takes an array, A, of integers as input and produces an array, R, where for every odd index i, $R[i] = A[i-1] + A[i]$. For even indices i, $R[i] = A[i]$.

To implement this the Obsidian functions `pair` and `unpair` will be used:

```
pair :: Arr a -> W (Arr (a,a))
unpair :: Arr (a,a) -> W (Arr a)
```

The `pair` function pairs the first element with the second, the third with the forth and so on. The `unpair` function performs the converse operation relative to `pair`. Composing `pair` and `unpair` gives the identity function:

$$\texttt{pair ->- unpair} \Rightarrow \texttt{id}_{\texttt{Arr a -> W (Arr a)}} \tag{1}$$

$$\texttt{unpair ->- pair} \Rightarrow \texttt{id}_{\texttt{Arr (a,a) -> W (Arr (a,a))}} \tag{2}$$

One more building block is needed to be able to implement this first example program. The function `fun` has type `(a -> b) -> Arr a -> W (Arr b)`, it applies a function to every element of a given array. Using these functions `pAdd` can be implemented as follows:

```
pAdd :: Arr IntE -> W (Arr IntE)
pAdd = pair ->- fun (\(x,y) -> (x,x+y)) ->- unpair
```

In the C code generated from this example, which is of type `Arr IntE -> W (Arr IntE)`, there will be one thread per array element. Threads with an even Id just pass their value over into the result array. Threads with an odd Id reads two elements, compute their sum and store the result.

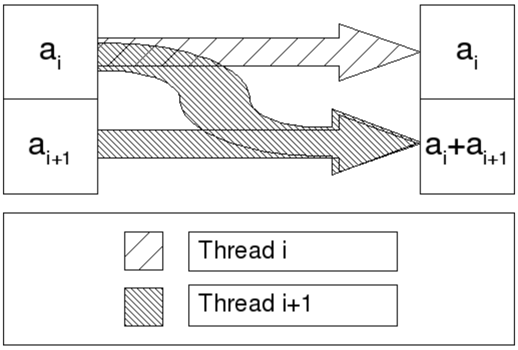


Figure 2: Pairwise addition of elements in an array

Obsidian provides a function called `execute` that takes as argument an Obsidian program and a Haskell list. This function generates C code from the first argument and compiles this using the NVIDIA CUDA C compiler. The compiled program is then executed on the GPU and the result presented to the user. Here a related function called `executeT` will be used. This function takes an extra argument which is the number of threads to instantiate for running the program. The applicaiton of `sync` below is there for purely technical reasons. It is at this `sync` point that the representation of the program is generated. An alternative would be to let the function `executeT` apply this top-level `sync` automatically.

```
> executeT 8 (pAdd ->- sync) [1,2,3,4,5,6,7,8]
[1,3,3,7,5,11,7,15]

> executeT 8 (pAdd ->- sync) [1,1,1,0,0,1,0,0]
[1,2,1,1,0,1,0,0]
```

The list supplied to the `executeT` function is just to be considered data for the program to use. In the current version it is completely decoupled from the type of the Obsidian program handed to `executeT`. For example, later `executeT` will be used to run an Obsidian function with type `Arr (IntE,IntE) -> W (Arr (IntE,IntE))` but the data will still be passed to `executeT` as a list of type `[Int]` and not `[(Int,Int)]` which is expected. Correcting these issues are high on the todo list.

3 Optimising Obsidian programs

The C code generated by Obsidian in the above example, `pAdd`, runs one thread per array element. In the `pAdd` example `sync` is of the type `Arr IntE -> W (Arr IntE)` and assigns one thread to each element of the array. In this case this means that only every second thread does any real work, the other threads just pass their value through. This is shown in figure 2. An optimisation of the program would be to run only half as many threads and let all of them operate on pairs of elements.

A new implementation of `pAdd` will now be derived from the previous one. In the C code generated from the new implementation every thread will operate on pairs on elements. The first step in deriving this optimisation is to define a function `pfy`, short for pairify. This pairify function takes an Obsidian function of type `Arr a -> W (Arr a)` and returns a function `Arr (a,a) -> W (Arr (a,a))`:

```
pfy :: (Arr a -> W (Arr a)) -> Arr (a,a) -> W (Arr (a,a))
pfy f = unpair ->- f ->- pair
```

Now to derive the pairwise version of `pAdd`, `pfy` is applied to `pAdd`. Here it is important that it is at the `sync` points that work is assigned to threads. The body of pAdd contains no sync, instead it is applied when actually executing the program.

```
pfy pAdd
=>
unpair ->- pAdd ->- pair
=> (definition of pAdd)
unpair ->- pair ->- fun (\(x,y) -> (x,x+y)) ->- unpair ->- pair
=> (unpair ->- pair = id )
fun (\(x,y) -> (x,x+y))
```

So the optimized version of `pAdd` here called `pAdd'` is implemented as follows:

```
pAdd' :: Arr (IntE,IntE) -> W (Arr (IntE,IntE))
pAdd' = fun (\(x,y) -> (x,x+y))
```

In the program `pAdd' ->- sync`, `sync` has type `Arr (IntE,IntE) -> W (Arr (IntE,IntE))`. Using this implementation the same result as in `pAdd` can be obtained using half the number of threads.

```
> executeT 4  (pAdd' ->- sync) [1,2,3,4,5,6,7,8]
[1,3,3,7,5,11,7,15]

> executeT 4  (pAdd' ->- sync) [1,1,1,0,0,1,0,0]
[1,2,1,1,0,1,0,0]
```

Figure 3 shows how the new `pAdd'` computes the same result as `pAdd` but using half as many threads. The thread with Id x will compute the results at index $2 * x$ and $2 * x + 1$ in this example.

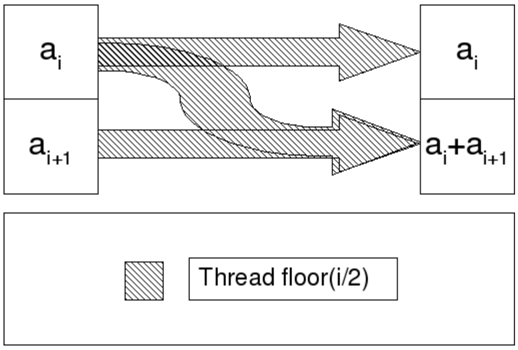


Figure 3: One thread per pair implementation of the pairwise addition.

4 Sorting

The goal of this section is to implement a periodic sorter. A periodic sorter sorts an array by repeatedly applying the same function on the data [3]. Here a periodic sorter will be implemented using a merger and a permutation function.

4.1 Implementing the merger

First the merger will be implemented. In the first implementation it will generate code that uses one thread per array element. A two element per thread implementation will then be derived from the first implementation.

This merger is based on the *shuffle-exchange* network and takes an array of length $2 * n$ as input where the first n values are sorted and the last n are reversly sorted. Figure 4 shows a merger with 8 inputs based on the shuffle-exchange network. The permutation used in this network is called `riffle`.

In Obsidian `riffle` is implemented by splitting the array into 2 and then shuffling them perfectly. The functions `halve` and `shuffle` are provided as primitives:

```
riffle = halve ->- shuffle
```

The other building block needed to implement the merger is a function that applies a function to pairs of elements. This has already been done in the `pAdd` example but will be generalised here in a function called `evens`.

```
evens f = pair ->- fun f ->- unpair
```

In the merger, `evens` will be used to apply a 2-sorter called `cmpSwap` to pairs of elements in the array. The 2-sorter is implemented using a conditional:

```
cmpSwap op (a,b) = ifThenElse (op a b) (a,b) (b,a)
```

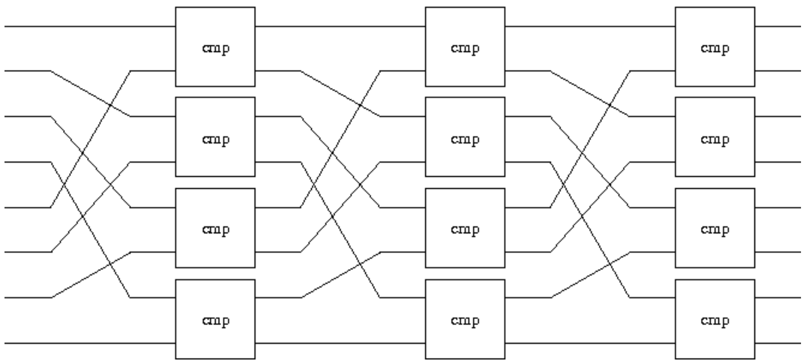


Figure 4: Merger based on the shuffle-exchange network with 8 inputs. There are 2-sorters in the nodes (compare and swap operations).

The body of the merger can now be described as follows. Here the `->>-` operator is used rather than the `->-` in combining the `riffle` and the `evens`. This choice is based on experimental results, in this particular case using `->>-` leads to a faster merger:

```
merge_body = riffle ->>- evens (cmpSwap (<*)) ->- sync
```

The `merge_body` function can be used to implement a merger by repeatedly applying it to the input array. The following program implements the merger:

```
merger :: Arr IntE -> W (Arr IntE)
merger arr = repE k merge_body arr
    where
      k = log2i (len arr)
      merge_body = riffle ->>- evens (cmpSwap (<*)) ->- sync
```

This program uses the `repE` combinator that repeats a program a number of times. In this case the program is repeated a number of times depending on the length of the input array.

Running `merger` on well-formed input gives:

```
> executeT 16 merger   [0,2,4,6,8,10,12,14,15,13,11,9,7,5,3,1]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

4.2 Deriving the pairwise merger

Again `pfy` will be used to derive the pairwise implementation of the merger. But the merger is a bit more complex than the example with `pAdd` in the beginning. In particular two more rules concerning `pfy` is needed:

$$\texttt{pfy (a ->- sync)} \Leftrightarrow \texttt{pfy a ->- sync} \tag{3}$$

$$\texttt{pfy (a ->>- b)} \Leftrightarrow \texttt{pfy a ->>- pfy b} \tag{4}$$

These rules can be justified using the identity rules from a previous section:

```
pfy (a ->- b)
==> expand pfy
unpair ->- a ->- b ->- pair
==> identity
unpair ->- a ->- id ->- b ->- pair
==> (pair ->- unpair = id)
unpair ->- a ->- pair ->- unpair ->- b ->- pair
==> contract pfy
pfy a ->- pfy b
```

Setting `b = sync` in the above derivation, rule 3 is obtained using the following equivalence:

$$\texttt{pfy sync}_{\texttt{Arr a -> W (Arr a)}} \Leftrightarrow \texttt{sync}_{\texttt{Arr (a,a) -> W (Arr (a,a))}} \tag{5}$$

Rule 4 is obtained directly from the derivation by just swapping −>− for −>>−. The result of `f ->- g` and `f ->>- g` is the same. However, doing it one way rather than the other may have performance implications.

Now these intuitions can be applied to the `merge_body` to derive a more efficient implementation of the merger. The associativity of −>− is also used without comments. The idea is again to move applications of `pfy` inwards until all applications of `sync` are of the desired type:

```
pfy (merge_body)
=> (expand merge_body)
pfy ( riffle ->>- evens (cmpSwap (<*)) ->- sync )
=> (distributivity over ->>-)
pfy riffle ->>- pfy (evens (cmpSwap (<*)) ->- sync)
=> equation 3
pfy riffle ->>- pfy (evens (cmpSwap (<*))) ->- sync
=> (specification of evens)
pfy riffle ->>-
  pfy (pair ->- fun (cmpSwap (<*) ->- unpair)) ->- sync
=> (expand pfy)
unpair ->- riffle ->- pair ->>-
  unpair ->- pair ->-
    fun  (cmpSwap (<*)) ->- unpair ->- pair ->- sync
=> equation 2
unpair ->- riffle ->- pair ->>- fun  (cmpSwap (<*) ->- sync
```

The function `repE` has type `Exp Int -> (Arr a -> W (Arr a)) -> Arr a -> W (Arr a)` and can thus be used unchanged with the new and more efficient `merge_body`. The new merger is implemented like this:

```
merger2 :: Arr (IntE,IntE) -> W (Arr (IntE,IntE))
merger2 arr = repE k merge_body arr
    where
      k = log2i (2 * len arr)
      merge_body = unpair ->- riffle ->- pair ->>-
                   fun (cmpSwap (<*)) ->- sync
```

```
> executeT 8 merger2 [0,2,4,6,8,10,12,14,15,13,11,9,7,5,3,1]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

4.3 Implementing the sorter

The merger described so far can be used to implement a periodic sorter. This is done by using the merger repeatedly together with a specific permutation function. The permutation function is called `tau` and is similar to a permutation used in [4]. The `tau` permutation is obtained by composing `unriffle` with a function that reverses half the array. Here this function will be implemented as `one rev`. The function `one` applies its first input to half of the elements in the array. The name `one` comes from [5]. Below is an example execution of `one rev`:

```
one_rev = one rev ->- sync :: Arr IntE -> W (Arr IntE)

> executeT 8 one_rev [1..8]
[1,2,3,4,8,7,6,5]
```

The `tau` permutation is defined as follows:

```
tau  = unriffle ->>- one rev
```

The first sorter, that uses one thread per element, is implemented like this using `tau`, `merge` and `repE`:

```
sorter arr = repE k periodicMerge arr
    where k = log2i (len arr)
          periodicMerge = tau ->>- merger
```

Below is a test run of the sorter on 8 elements just as an example.

```
> executeT 8 sorter [1,4,3,7,8,2,5,6]
[1,2,3,4,5,6,7,8]
```

To implement the sorter that uses one thread per pair of elements it is also necessary to define a pairwise tau. The pairwise `tau` can be derived from `tau` by applying `pfy`:

```
pfy tau
==> (expand tau)
pfy (unriffle ->>- one rev)
==> (distribute pfy)
pfy unriffle ->>- pfy (one rev)
==> (expand pfy)
unpair ->- unriffle ->- pair ->>- unpair ->- one rev ->- pair
```

Lets call this version of `tau` `tau2`

```
tau2 = unpair ->- unriffle ->- pair ->>-
          unpair ->- one rev ->- pair
```

Using `tau2` and `merger2` the two element per thread sorter is implemented:

```
sorter' arr = repE k periodicMerge arr
    where k = log2i (2 * len arr)
          periodicMerge = tau2 ->>- merger2
```

Below is an example of running this version of the sorter. Here using only 4 threads:

```
> executeT 4 sorter' [1,4,3,7,8,2,5,6]
[1,2,3,4,5,6,7,8]
```

5 Conclusions

Being able to do more work per thread is important since there is a limitation to the number of threads that can be executed within a `block` on the target platform. A block is group of threads executing on one of the SIMD multiprocessors within the GPU. It is only within a block that threads can communicate and synchronise during the execution of a single kernel. The upper limit for the number of threads per block is 512, but depending on the resource demands of the program this might be lower. Executing about 256 threads per block seems to be an ideal.

Mergers generated from the obsidian descriptions above has been used in the implementation of a sorter for 1 Million elements. At the moment using a merger that merges 512 elements using 256 threads this sorter sorts 1 Million elements in 155 ms on a NVIDIA GeForce 9800 GX2 using one of its GPUs. Using the single element per thread merger the run time is 193 ms. This can be compared to quick-sort on the CPU (Intel Core 2 2.4GHz using one core) taking 322 ms.

sorter	193ms	1 thread per element
sorter'	155ms	1 thread per pair
quick-sort	322ms	1 core

Table 1: Run times for a 1 million element sorter implemented using mergers and sorters adapted from the ones implemented here.

6 Future work

To make the derivations described above possible changes had to be made to Obsidian. The nature of these changes has been very ad-hoc and currently it is for example not possible to go further than from elements to pairs of elements. Generalising the methods presented here to arbitrary length sub arrays is currently a work in progress.

References

[1] Nvidia cuda compute unified device architecture: Programming guide version 1.1, November 2007. www.nvidia.com/cuda.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming, ICFP*, pages 174–184. ACM, 1998.

[3] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, Lecture Notes in Computer Science. Springer Verlag, 2001.

[4] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The periodic balanced sorting network. *J. ACM*, 36(4):738–757, 1989.

[5] G. Jones and M. Sheeran. *Collecting Butterflies.* Oxford University Programming Research Group, 1991. ISBN 0-902928-69-4.