Counting and Occurrence Sort for GPUs using an Embedded Language

Josef Svenningsson Bo Joel Svensson Mary Sheeran

Dept, of Computer Science and Engineering
Chalmers University of Technology
{josefs, joels, ms}@chalmers.se

Abstract

This paper investigates two sorting algorithms: counting sort and a variation, occurrence sort, which also removes duplicate elements, and examines their suitability for running on the GPU. The duplicate removing variation turns out to have a natural functional, data-parallel implementation which makes it particularly interesting for GPUs.

The algorithms are implemented in Obsidian, a high-level domain specific language for GPU programming.

Measurements show that our implementations in many cases outperform the sorting algorithm provided by the library Thrust. Furthermore, occurrence sort is another factor of two faster than ordinary counting sort. We conclude that counting sort is an important contender when considering sorting algorithms for the GPU, and that occurrence sort is highly preferable when applicable. We also show that Obsidian can produce very competitive code.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Code generation

Keywords Sorting, embedded language

1. Introduction

Sorting is an ever important, ever fascinating field of computer science. The introduction of GPUs has introduced new challenges in designing fast sorting algorithms.

This paper focuses on counting sort together with a variation which removes duplicate elements. We call this variation *occurrence sort*. Counting sort is a non-comparing sort suitable for parallel implementation. The occurrence sort variation presented here is interesting because it seems to be a particularly good fit for executing on the GPU. It has a very natural functional, data-parallel implementation. We believe we are the first to study this variation in the literature.

These algorithms are explored in Obsidian [3, 6], a domain specific language targeting GPU programming. The goal of Obsidian is to strike a balance between high-level constructs and low-level control. We want to provide the programmer with a set of tools for low-level experimentation with the details that influence performance when implementing GPU kernels. The counting sort case study shows that Obsidian generates competitive kernels with relatively little programmer effort. That Obsidian is an embedded language allows us to rapidly experiment with the addition of features and with varying programming idioms. The version used in this case study adds global arrays and atomic instructions to Obsidian, see section 4.1.

The contributions of this paper are:

- We provide measurements (section 7) showing that counting sort is a competitive algorithm for sorting keys on the GPU, outperforming the sorting implementation in the library Thrust[15] in many cases.
- Occurrence sort is shown to be particularly suitable for implementing on the GPU (section 6) and performs well (section 7).
- The Obsidian implementation of the two sorting algorithms is detailed along with the generated CUDA (sections 4 and 5).

1.1 Related work

Sorting has applications in the computer graphics field [17]. Example instances of sorting and duplicate element removal in computer graphics can be identified in references [16] and [7]. Another example of where sorting and the removal of duplicates have applications is in databases [10].

The paper [9] implements counting sort in CUDA, and also optimisations that overlap computations with transferring data to and from the GPU. Our work differs by being implemented in a high-level embedded language and also by implementation of the occurrence sort variant of counting sort. We have not been concerned with trying to overlap computations and data transfer, but have instead focused solely on computations within the GPU.

Obsidian is a high-level embedded language for GPU programming. Other such approaches include Accelerate [2] and Nikola [12]. Accelerate and Nikola both take an even higher level approach compared to Obsidian and abstract more from GPU details. Another example of providing a high-level interface to programming the GPU is the C++ library Thrust [15].

2. Counting Sort

Counting sort, like radix sort, relies on array indexing rather than comparison to order elements. The key idea of counting sort is to count the number of occurrences of each element, and then perform a prefix sum over all the counts. The prefix sum generates an array which, for each element in the original array, will point to its position in the final sorted array. In order for this to work,

FHPC '13, September 23 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2381-9/13/09…$15.00.
http://dx.doi.org/10.1145/2502323.2502325

```
countingSort :: (Int,Int)
                -> Array Int Int
                -> Array Int Int
countingSort range input =
  reconstruct $
  scanlArray (+) 0 $
  histogram range input

histogram :: (Int,Int)
             -> Array Int Int
             -> Array Int Int
histogram range =
  accumArray (\i _ -> i+1) 0 range . elems . fmap dup

reconstruct :: Array Int Int -> Array Int Int
reconstruct arr =
  array (0,arr!l-1)
    [ (a,i)
    | (i,e) <- init (assocs arr)
    , a <- [ e .. arr!(i+1) - 1] ]
  where (_,l) = bounds arr

dup a = (a,a)
```

Figure 1. A Haskell specification of Counting Sort. The function `scanlArray` is not standard but works much like `scanl` for lists.

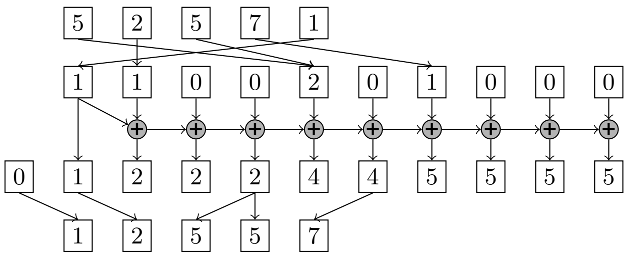

Figure 2. An example run of counting sort in diagrammatic form

counting sort needs a lower and upper bound on the values of the elements in the input array. Counting sort is typically explained in terms of sorting integers and we will do the same here.

The array `{5,2,5,7,1}` will be used as a running example. In addition, the range `(1,10)` will be the approximation of the range of the keys in the input. A diagrammatic illustration of our running example is presented in figure 2.

Counting sort consists of three steps: histogram creation, prefix sum and reconstruction.

The histogram creation step computes a new array which records how many times an integer occurs in the input array, i.e. the new array is a histogram of the input array. The size and indices of the histogram array are determined by the range which approximates the lower and upper bounds of the keys.

Computing the histogram of `{5,2,5,7,1}` with the range `(1,10)`, results in `{1,1,0,0,2,0,1,0,0,0}`. Indexing in this array starts at `1` because that's the lower bound given in the range. Each element in the histogram indicates how many times the corresponding value occurs in the input array. For example, `5` occurs twice and `8` occurs zero times.

The second step of the algorithm computes the prefix sum (or inclusive scan with plus) of the histogram array, and prepends a zero. For input array `{5,2,5,7,1}` and range `(1,10)`, the histogram array is `{1,1,0,0,2,0,1,0,0,0}` and its prefix sum is `{0,1,2,2,2,4,4,5,5,5,5}`.

The array computed by the prefix sum is called the position array. It indicates where in the final sorted array each value is

```
countingSort :: (Int,Int)
                -> Array Int Int
                -> Array Int Int
countingSort range input =
  reconstruct $
  scanlArray (+) 0 $
  occurs range input

occurs :: (Int,Int)
          -> Array Int Int
          -> Array Int Int
occurs range =
  accumArray (\_ _ -> 1) 0 range . elems . fmap dup
```

Figure 3. A Haskell specification of occurrence sort

placed. The indices of the position array correspond to values in the final array and the elements correspond to positions in the final array. The reconstruct step takes care of distributing the values into the final sorted array. For each index in the position array, the difference between the element at that index and the index plus one is computed. This difference indicates how many times the value occurs in the final array.

The position array in our running example is `{0,1,2,2,2,4,4,5,5,5,5}`. Indexing starts at `1` just as with the histogram. As an example, element `1` occurs once in the final array because the difference between the element at position `1` and `2` in the position array is precisely one. Also, `1` is placed on index `0`. Furthermore, element `5` occurs twice in the final array (the difference between `2` and `4` in the position array), on indices `2` and `3`. Continuing this process with all elements will result in the following final array: `{1,2,5,5,7}`.

A functional description of counting sort can be found in figure 1. The main function is `countingSort` which takes two arguments: a range with a safe approximation of the lower and upper bounds of the values in the input array, and the input array itself. The histogram step is implemented by the function `histogram`, the prefix sum with `scanlArray` and the reconstruct step with the function `reconstruct`.

2.1 Occurrence sort: Removing duplicates

There is an interesting variation of counting sort which removes duplicate elements. It seems to be folklore; it is mentioned on Wikipedia [1]. However, we haven't found any description of it in the literature. It is particularly interesting because it allows for an efficient data-parallel implementation.

Figure 3 shows the changes we make to counting sort in order to obtain occurrence sort. Only the `histogram` function needs changing so that instead of computing an actual histogram which counts the number of occurrences of an element, it only records if an element occurs or not with a `1` or a `0`. Since the function no longer computes a histogram it has been renamed to `occurs`.

The prefix sum and the reconstruction step can be kept intact. They will still compute the correct result.

Compared to the full counting sort, occurrence sort has some interesting properties which make it more suited for parallel execution. In particular, the histogram construction phase and the reconstruction phase allow for less synchronisation.

When constructing the histogram in the counting sort algorithm, an index has to be incremented each time the corresponding value is found in the input array. When parallelising this operation, it is important that the increments are done atomically, which incurs a cost. The occurs phase presented above doesn't need to employ any special mechanism to ensure atomicity, since it writes a single word with the value 1 to memory.

Given the array {5,2,5,7,1} as input and (1,10) as range, the occurs step will now compute {1,1,0,0,1,0,1,0,0,0}. The prefix sum will in turn produce the following position array: {0,1,2,2,2,3,3,4,4,4,4}. The reconstruct step computes {1,2,5,7}. In particular, there is only one occurrence of 5 in the final array.

3. GPUs and CUDA

The GPUs we target in this paper and with Obsidian are NVIDIA GPUs which support CUDA. Here we provide some background information on GPUs that we hope will aid in the understanding of example programs throughout this paper.

GPUs supporting CUDA are based on a scalable design. The GPU consists of a number of so-called multiprocessors (MPs) each consisting of a number of processing elements (PEs); sometimes these are referred to as cores or as stream processors. An MP also contains a local shared memory through which threads running on the PEs can communicate. The key that makes the architecture scalable is that a GPU consists of one or more of these MPs. This also influences the programming model significantly; since threads can only communicate in a synchronised manner using the local memory (or in a very limited way using atomic operations) the programmer must partition the computation in such a way that its communication patterns follow this architectural constraint. The GPU also has access to a larger memory called the *global* or *device* memory. The sizes of these memories and the number of MPs and PEs per MP vary slightly between generations of GPUs. For example, the GPU used in the benchmarks (section 7) has a total of 1344 single-precision floating point CUDA cores and has 2 GB global memory. The very latest GPU architectures from NVIDIA slightly digress from what is outlined here but mostly still adhere to the same programming model. For exact details refer to [14].

The programming model that CUDA exposes is called single program multiple threads (SPMT) and is a slight variation of the SPMD concept. This programming model reflects the GPU architecture. In CUDA, the programmer writes a single program that is executed by many threads. Because of the scalable architecture, with its one or more MP, these threads are grouped into *blocks* (now up to 1024 threads per block). A block of threads is executed on an MP; thus only threads within that block can communicate using the shared memory. The same CUDA program can be executed on any GPU along the scale, from the smallest with only one MP to the largest. The only difference is in the number of blocks of threads that can be executed in parallel. This constraint implies that all blocks must be free of any sequential dependencies. The collection of all blocks is called a *Grid*.

A CUDA program has two parts; there is a coordination program that runs on the CPU and there are kernels that execute on the GPU MPs. The program describing a kernel is parameterised over a blockId and a threadId so that decisions can be made from that information during execution. A typical CUDA program starts out by loading data into local memory using some function of blockId and threadId. It then computes on the local memory and uses a syncthreads primitive when exchanging values between threads. When the local computation is done the final results are written back into global memory again using some function of blockId and threadId.

For more CUDA and GPU specifics see references [13, 14].

4. Obsidian

Obsidian is a programming language for expressing general purpose GPU kernels, compute kernels, in a high level and functional style. Obsidian is embedded in Haskell. Specifically, Obsidian is a library of functions for generating and combining abstract syntax trees. By using features of Haskell such as overloading and

```
__global__ void addv(float *i0,
                     float *i1,
                     float *r){
  unsigned int ix =
     blockIdx.x * blockDim.x + threadIdx.x;

  extern __shared__ float sm[];
  sm[threadIdx.x] = i0[ix] + i1[ix];
  r[ix] = sm[threadIdx.x];
}
```

Figure 4. The code illustrates elementwise addition of vectors. Shared memory (the sm array) is used to allow the addv program to be run with the result r pointing to the same memory area as either of the inputs.

```
#define BLOCK_SIZE 32
#define BLOCKS 4
#define N (BLOCKS * BLOCK_SIZE)

int main(int argc, char **argv){
  float *v1, *v2, *r;
  float *dv1, *dv2, *dr;

  v1 = (float*)malloc(N*sizeof(float));
  v2 = (float*)malloc(N*sizeof(float));
  r = (float*)malloc(N*sizeof(float));

  //Generate or read input data.
  ...

  //Allocate arrays in Global memory
  cudaMalloc((void**)&dv1, sizeof(float) * N );
  cudaMalloc((void**)&dv2, sizeof(float) * N );
  cudaMalloc((void**)&dr, sizeof(float) * N );

  //Copy data into Global memory
  cudaMemcpy(dv1, v1, sizeof(float) * N,
                      cudaMemcpyHostToDevice);
  cudaMemcpy(dv2, v2, sizeof(float) * N,
                      cudaMemcpyHostToDevice);

  //Launch the vector add kernel.
  addv<<<BLOCKS, BLOCK_SIZE, BLOCK_SIZE * sizeof(float)>>>
      (dv1,dv2,dr);

  //Launch further kernels on the data.
  ...

  cudaMemcpy(r, dr, sizeof(float) * N ,
                    cudaMemcpyDeviceToHost);

  cudaFree(dv1);
  cudaFree(dv2);
  cudaFree(dr);

  //Show or further process results on the CPU.
  ...
}
```

Figure 5. This code starts a CUDA kernel on the GPU. The syntax <<<nb,nt,sm>>> is used to set up a kernel launch configuration. The nb, nt and sm quantities refer to the number of blocks, the number of threads/block and the amount of shared memory.

higher-order-functions, the library/language border is blurred and we call the result an embedded language. From the abstract syntax trees generated while running an Obsidian program, NVIDIA CUDA code is generated.

4.1 A brief history of Obsidian

In the Obsidian project we experiment with combinators and language features for GPU kernel implementation. We seek to strike a balance between low-level control and useful high-level abstractions. Over time, this has led to a number of different versions of Obsidian. In reference [6], two versions of Obsidian differing by having a *monadic* or a limited *arrow* like interface are described. We have settled on using the monadic style.

One of the array representations that Obsidian uses, the Pull array, was present already in the earlier work but went under a different name. It was not until discovering a complementary array representation, that we call push arrays, that the traditional array was renamed to pull.

The Obsidian implementation of pull arrays was influenced by Pan's representation of images as functions from coordinates to colour values [4]. A pull array is a function from index to value and an associated length.

Push arrays were added to Obsidian first in reference [3]. A push array specifies where elements are to end up in a result array, rather than where they come from (as in a pull array). In essence, a push array is a computation which writes an array to memory, but is parameterised on the *write function* which writes a single element into memory. The push array function can then use the write function zero, one, or several times.

Push arrays were added to Obsidian with the purpose of solving some performance issues that we had been struggling with for a long time. These problems were mainly concerned with concatenating or combining arrays in an efficient way. Push arrays have also been adopted by others and are used in the implementation of filters in reference [11] and in the new Nikola[1].

Both push and pull arrays are virtual, in the sense that they do not directly correspond to any data in a region of memory. This virtual nature of the array representations gives us fusion of operations for free. The programmer has to explicitly force an array (using a `force` function) to actually compute the values and store them in memory and to prevent operations from fusing.

In the version of Obsidian used in this paper[2], new features were added. We introduce two new variants of push and pull arrays called `GlobPush` and `GlobPull`. This introduces a distinction between local arrays, short enough to be computed by a block of the GPU, and global arrays that represent computations spread over blocks of the GPU. This is an important step when it comes to Obsidian's capabilities; we can now express in what order a kernel fetches elements from global memory. Earlier versions of Obsidian were limited to straight-line block indexing. A kernel generated by older Obsidian accessed blocks of elements in a fixed way (thread block i accessed elements using `(i * blocksize) + f tid`, where `f` can permute indices within a block but there is no similar way to permute the actual blocks).

In order to implement histograms we also need to add atomic operations to the language. The addition of atomic operations here is done in a rather ad hoc way and pushes us to program in a very low-level and imperative style. We strive to incorporate much of CUDA's low-level functionality into Obsidian and adding atomic operations is an attempt in that direction.

The version of Obsidian used in this paper also introduces a new monadic representation of GPU programs. We call the monad `Program`. The `Program` type has a parameter that represents at what level in the GPU hierarchy it executes. There are sequential *Thread* programs and parallel *Block* and *Grid* programs. Type synonyms are available, `TProgram`, `BProgram` and `GProgram`.

Many of the new features we mention above have been further refined and are present in versions of Obsidian following the one we use in this paper. For example, in the master branch of Obsidian[3] we build on the ideas in this paper. There, global and local arrays are represented by the same data type. That work also goes further by implementing hierarchy generic functions that are applicable at different levels of the Thread, Block and Grid hierarchy. We are also experimenting with the addition of mutable arrays in order to better integrate the atomic operations[4].

4.2 Obsidian programming example

In the different variants of counting sort that are developed in this paper, the prefix sum operation is the same. It is therefore natural to use as an introductory example of how Obsidian programs are written.

We need to operate on arrays that are too long to fit in a single block. The resulting prefix sum is implemented as shown in reference [5], see figure 6. There are three steps involved. First local prefix sums are computed; the rightmost elements elements are also stored in a separate, auxiliary, array. Following this, the prefix sum of the auxiliary array is computed. The prefix sums used in phase A and B of the figure can both be generated from the same Obsidian program. Finally the local prefix sums and the result of the auxiliary prefix sum are combined, forming a large prefix sum over the entire array.

The figure also shows what kernels we need to produce. First, something that performs a number of local prefix sums in parallel over the full length of an array is needed. The prefix sum algorithm that we implement here is based on divide and conquer and originates from Sklansky [18].

The code below can be thought of as a program generator. Given an integer, n, it gives a prefix computation for 2^n elements. Here we go directly to generating the prefix sums operations (with the operator + hard coded) rather than a higher order function that can be used to more generically.

```
sklansky :: Int
            -> Pull EWord32
            -> BProgram (Pull EWord32)
sklansky 0 arr = return arr
sklansky n arr = do
    let arr1 = binSplit (n-1) fan arr
    arr2 <- force arr1
    sklansky (n-1) arr2
```

The `sklansky` code above is short but still contains much that needs explanation. First, the type specifies that the function takes a normal Haskell Int and an input array (`Pull EWord32`). The result is a `BProgram` that produces a result array. The function `binSplit` is in the Obsidian library and is used to implement divide and conquer algorithms. This function splits an array recursively a given number of times and then applies some computation on all parts. In this case the `fan` operation implemented below is used.

```
fan arr =  a1 ‘conc‘  fmap (+ (last a1)) a2
    where
      (a1,a2) = halve arr
```

The `fan` operation splits an array down the middle and then adds the element at the highest index of the first half to all elements of the second. Then the two parts are concatenated back together.

[1] github.com/mainland/nikola/blob/master/src/Data/Array/Nikola/Repr/Push.hs

[2] github.com/svenssonjoel/Obsidian/branches/February2013

[3] github.com/svenssonjoel/Obsidian

[4] github.com/svenssonjoel/Obsidian/tree/mutable

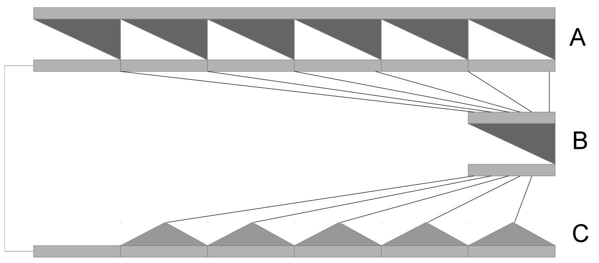

Figure 6. Illustration of how to combine many local prefix sum computations into a large one. The algorithm has three steps: A compute local prefix sums, B recursively compute prefix sums on the maximums of the local prefix sum calculations, C distribute summed maximums over the local results.

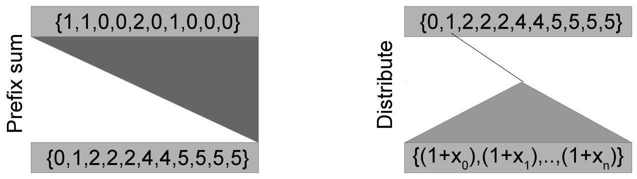

Figure 7. Close-up on the prefix sum and distribute operations used in Figure 6.

In order to obtain both the local prefix sum results and the maximum element of the block, a small wrapper function is created.

```
sklanskyMax :: Int
               -> Pull EWord32
               -> BProgram (Pull EWord32,
                            Pull EWord32)
sklanskyMax n arr = do
  res <- sklansky n arr
  return (res,singleton (last res))
```

All this wrapper does is compute the prefix sum followed by returning the result and a singleton array containing the maximum, `last`, element.

This local prefix sum is run on sub-parts of the global array. The function `mapDist` splits up the array, runs a local computation on each part using the MPs of the GPU; when it finishes, the result of that computation is an array that remains distributed across the local memories of the GPU MPs.

```
sklanskyG :: Int
             -> GlobPull EWord32
             -> GProgram (GlobPush EWord32,
                          GlobPush EWord32)
sklanskyG logbsize input =
  toGProgram (mapDist (sklanskyMax logbsize)
                      (2^logbsize)
                      input)
```

In the `sklanskyG` function the local prefix combinator implemented so far is turned into a global prefix sum computation. The `toGProgram` function takes a function from blockId to a `BProgram` and gives back a `GProgram`, that is a program running on the full GPU (across many MPs).

The CUDA code generated from `sklanskyG` is shown in figure 8. Writing that type of CUDA code by hand is both tedious and error prone. The Obsidian code is both shorter, more compositional and easier to read.

```
__global__ void sklansky(uint32_t *input0,
                         uint32_t *output0,
                         uint32_t *output1){
  extern __shared__ unsigned char sbase[];
  ((uint32_t *)sbase)[tid] =
    (((tid&0x1)<0x1)
      ? input0[((bid*32)+((tid&0xFFFFFFFE)|(tid&0x1)))]
      : (input0[((bid*32)+((tid&0xFFFFFFFE)|0x0))]+
         input0[((bid*32)+((tid&0xFFFFFFFE)|(tid&0x1)))]));
  __syncthreads();
  ((uint32_t *)(sbase + 128))[tid] =
    (((tid&0x3)<0x2)
      ? ((uint32_t *)sbase)[((tid&0xFFFFFFFC)|(tid&0x3))]
      : (((uint32_t *)sbase)[((tid&0xFFFFFFFC)|0x1)]+
         ((uint32_t *)sbase)[((tid&0xFFFFFFFC)|(tid&0x3))]));
  __syncthreads();
  ((uint32_t *)sbase)[tid] =
    (((tid&0x7)<0x4)
      ? ((uint32_t *)(sbase+128))[((tid&0xFFFFFFF8)|(tid&0x7))]
      : (((uint32_t *)(sbase+128))[((tid&0xFFFFFFF8)|0x3)]+
        ((uint32_t *)(sbase+128))[((tid&0xFFFFFFF8)|(tid&0x7))]));
  __syncthreads();
  ((uint32_t *)(sbase + 128))[tid] =
    (((tid&0xF)<0x8)
      ? ((uint32_t *)sbase)[((tid&0xFFFFFFF0)|(tid&0xF))]
      : (((uint32_t *)sbase)[((tid&0xFFFFFFF0)|0x7)]+
         ((uint32_t *)sbase)[((tid&0xFFFFFFF0)|(tid&0xF))]));
  __syncthreads();
  ((uint32_t *)sbase)[tid] =
    ((tid<0x10)
      ? ((uint32_t *)(sbase+128))[tid]
      : (((uint32_t *)(sbase+128))[0xF]+
         ((uint32_t *)(sbase+128))[tid]));
  __syncthreads();
  ((uint32_t *)(sbase + 128))[tid] = ((uint32_t *)sbase)[tid];
  if (tid<1){
    ((uint32_t *)(sbase + 256))[tid] = ((uint32_t *)sbase)[31];
  }
  __syncthreads();
  output0[((bid*32)+tid)] = ((uint32_t *)(sbase+128))[tid];
  if (tid<1){
    output1[(bid+tid)] = ((uint32_t *)(sbase+256))[tid];
  }
}
```

Figure 8. CUDA code generated from the Obsidian `sklanskyG` prefix sum program. This example shows the 32 element version of the prefix sum computation

One kernel remains to be implemented, `distribute`.

```
distribute :: EWord32
              -> GlobPull EWord32
              -> GlobPull EWord32
              -> GlobPull EWord32
distribute bs maxs inputs = zipWithG (+) maxs' inputs
  where
    maxs' = repeat bs maxs
    repeat n = ixMap (\ix -> ix `div` n)
```

The `distribute` kernel takes three inputs, a number, n, and two global pull arrays. The elements of the first array are repeated n times and then the result is element wise added to the second input array. The code generated from this kernel is shown in figure 9. The `distribute` kernel is used in the implementation of a large prefix sum (over more elements then what can fit in shared memory) from local prefix sum computations. This is indicated in figure 6.

```
__global__ void distribute(uint32_t s0,
                                uint32_t *input1,
                                uint32_t *input2,
                                uint32_t *output0){
  output0[((bid*bDim)+tid)] =
      (input1[(((bid*bDim)+tid)/s0)]+
       input2[((bid*bDim)+tid)]);

}
```

Figure 9. CUDA code generated from the Obsidian `distribute` function.

4.3 A note about generated code

All generated code shown in this paper has been edited slightly by hand. This is partly to increase readability but also to conserve space. The changes that have been made are entirely cosmetic. Line breaks have been inserted in too long lines. All occurrences of `blockIdx.x`, `blockDim.x` and `threadIdx.x` have been replaced with `bid`, `bDim` and `tid`. Some decimal constants have been changed to hexadecimal. No other changes have been made.

5. Implementing counting sort in Obsidian

5.1 Histogram

When computing the histogram, atomic operations are needed. Index i in the histogram array is incremented for each occurrence of the value i in the input array, and if there are multiple occurrences of i then multiple threads will try to increment, leading to a possible race. CUDA atomic operations must be applied to data stored at an actual memory location. This does not fit very well into our setting with virtual arrays (arrays that do not necessarily represent data in memory). But in Obsidian it is possible to drop down to a low enough level of abstraction to still implement this function. However, we are searching for suitable higher level abstractions to apply here.

```
histogram :: GlobPull EInt32
             -> GProgram ()
histogram gpull = do
  global <- Output $ Pointer Word32
  forAllT $ \gix ->
     atomicOp global
              (int32ToWord32 (gpull ! gix))
              AtomicInc
```

Below, the code generated from this Obsidian program is shown.

```
__global__ void histogram(int32_t *input0,
                          uint32_t *output0){

  atomicInc(output0+(uint32_t)(input0[((bid*bDim)+tid)]),
            0xFFFFFFFF);

}
```

The generated code uses `atomicInc` to increment a value in an array. The function is conditional, incrementing only if the value in memory is larger than the second argument. Since we always want to increment, no matter the value in memory, we've supplied `0xFFFFFFFF`, which is the largest possible value.

The Obsidian code and the CUDA code are very similar in size and complexity. However, an important advantage of the Obsidian code is that it composes better than the CUDA code. If the input to `histogram` was a `GlobPull` array produced by some other Obsidian function, then the two functions would be fused, typically generating much faster code and not allocating memory for the fused array.

5.2 Reconstruct

The kernel for the reconstruct step uses one thread per element in the input array. Each thread is implemented as a loop to write its corresponding element as many times as it should occur in the output.

```
reconstruct :: GlobPull EWord32
               -> GlobPush EInt32
reconstruct (GlobPull ixf) = GlobPush f
  where f k =
    do forAllT $ \gix ->
         let startIx = ixf gix
         in  SeqFor (ixf (gix + 1) - startIx) $ \ix ->
               k (word32ToInt32 gix) (ix + startIx)
```

The generated CUDA code for `reconstruct` can be seen below. Modulo syntax and some index manipulation, the code is very similar to the Obsidian code.

```
__global__ void reconstruct(uint32_t *input0,
                            int32_t *output0){

  for (int i1 =  0;
       i1 < (input0[(((bid*bDim)+tid)+1)]-
             input0[((bid*bDim)+tid)]);
       i1++)
  {
    output0[(i1+input0[((bid*bDim)+tid)])] =
      (int32_t)(((bid*bDim)+tid));

  }

}
```

Again, the difference between Obsidian and CUDA might seem small, but just as above, the code in Obsidian composes better.

6. Implementing occurrence sort in Obsidian

Occurrence sort uses an `occurs` kernel instead of a histogram. The result of the occurs computation is an array with a one at indices corresponding to values occurring in the input array. The reconstruction of the sorted (and duplicate free) array is also done slightly differently.

6.1 Occurs

The occurs kernel is implemented by using the `scatterGlobal` function from the Obsidian library. This function takes two arrays, one of indices to write to and a second of elements to write. In this case, it is applied to the input array and an array containing all ones (`replicateG 1`).

```
occurs :: GlobPull EInt32 -> GlobPush EWord32
occurs elems =
  scatterGlobal (fmap int32ToWord32 elems) (replicateG 1)
```

The CUDA code generated from this function is shown below.

```
__global__ void occurs(int32_t *input0,
                       uint32_t *output0){

  output0[(uint32_t)(input0[((bid*bDim)+tid)])] = 1;

}
```

When running this code, it is possible that many threads write to the same target element. Since all are writing a one, no atomic operations are needed; the result will still be one.

It is worth comparing the Obsidian code for the `histogram` kernel and the `occurs` kernel. The `histogram` kernel is highly imperative in nature, and requires atomic operations to manage synchronisation between threads. The `occurs` kernel, on the other hand, has a very straightforward data-parallel implementation. Not only is it simpler, but as we will see in section 7, it is also significantly faster.

6.2 Reconstruct

The reconstruct kernel for the occurrence sort algorithm is almost identical to the kernel used for standard counting sort. The only difference is that a conditional can be used instead of a loop. This can be seen as a slight optimisation; the previous version of reconstruct could still be used in its place.

```
reconstruct :: GlobPull EWord32
            -> GlobPush EInt32
reconstruct (GlobPull ixf) = GlobPush f
  where f k =
    do forAllT $ \gix ->
         let startIx = ixf gix
         in  Cond ((ixf (gix + 1) - startIx) ==* 1) $
               k (word32ToInt32 gix) startIx
```

This variant of reconstruct results in the generated code below.

```
__global__ void reconstruct(uint32_t *input0,
                            int32_t *output0){

  if ((input0[(((bid*bDim)+tid)+1)]-
       input0[((bid*bDim)+tid)])==1){

  output0[input0[((bid*bDim)+tid)]] =
     (int32_t)(((bid*bDim)+tid));

  }

}
```

7. Performance evaluation

This section presents the experimental results. Our implementations of counting sort are compared to the implementation of sorting from the Thrust library [15]. Thrust is a C++ library, developed by NVIDIA, which provides abstractions similar to the standard template library, but targets the GPU. The reason we have chosen to compare with Thrust is that it has similar goals to Obsidian: to provide high-level constructs while generating efficient GPU code.

All the data being sorted in the measurements are 32 bit unsigned integer valued keys. The convention used in the charts is that the y-axis represents time in ms. The different experiments are called *sorted* for sorting of an already sorted array, and *unique* for sorting of randomised arrays where each element occurs exactly once. There are also a number of experiments called Tx for a number x; the arrays used in these experiments contain elements from an interval of numbers (0 to $2^x - 1$).

Figures 10 and 11 show the results of comparing our full counting sort to Thrust sort and occurrence sort to Thrust's `sort` and `unique` functions. Our implementation of counting sort consistently outperforms Thrust, except for small ranges of values. The reason counting sort performance degrades is that the number of threads used in the reconstruct step is proportional to the range, and therefore at small ranges the execution becomes more sequential.

We compare occurrence sort to sort followed by unique from the Thrust library because that seems to be the current best practice [10]. Our implementation is always at least twice as fast but in many cases it outperforms Thrust by more than a factor of four. Our method is clearly beneficial for removing duplicates.

```
float total_time;

cudaEventRecord(start,0);
for (int i = 0; i < 1000; ++i) {
    // perform counting sort.
}
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&total_time,start,stop);
```

Figure 13. The timing methodology used in performance experiments. A CUDA event is recorded before and after the execution on the GPU. Their respective recorded times are then compared.

We also compare occurrence sort and counting sort to each other. The result of this is shown in figure 12. We can see that the occurrence sort is typically a factor of two faster than the regular sorting algorithm. The speedup comes from the fact that the `occurs` kernel doesn't need to perform any kind of synchronisation, whereas the `histogram` kernel needs to use atomic operations.

The charts included in this paper show the performance when sorting eight respectively 32 million (2^{23} resp. 2^{25}) elements. However, we ran all benchmarks with a number of elements, N, equal to 2^n for n between 20 and 25. We found that the interesting comparison is not in what happens as N grows larger but rather what happens as we vary the ranges of the elements contained in the arrays.

All timing was performed on a system with an NVIDIA GTX670 GPU.

7.1 CUDA framework for the benchmarks

To obtain the timing measurements above, a framework[5] written in CUDA was used. This framework looks very similar to the CUDA code in figure 5. The main difference is that a number of kernels are run in sequence, as shown in figure 14. The timing is performed on the GPU computation only, as illustrated by figure 13.

8. Discussion

It is instructive to compare the code of the two variants of counting sort. The standard algorithm requires mutations and its parallel implementation is rather complicated. The occurrence sort variation avoids key synchronisations, making the algorithm data-parallel and this gives it a performance advantage. However, in applying the trick to remove synchronisation we have changed the semantics of the function. In this case, it is still useful, albeit less generally applicable. The idea of removing synchronisation is a general and important idea for speeding up algorithms, but it is not clear how general the method used in this paper is. We currently do not know of any other algorithms which could benefit from a similar trick.

If the desired effect is to sort and remove duplicates (for an example of this see [10]) then there is a clear benefit to using occurrence sort over the `sort` combined with `unique` method.

In this paper, we are entirely focused on the implementation and performance of kernels. Reference [9] goes into the details of performing data transfers in parallel with ongoing computations. We have not addressed this issue at all in our implementations. This is also the reason why we haven't performed any benchmarks against the algorithm in [9], because it would compare very different things. As a piece of future work, it would be interesting to add

[5] Source code needed to repeat our experiments is available at `www.cse.chalmers.se/~joels/csort.html`

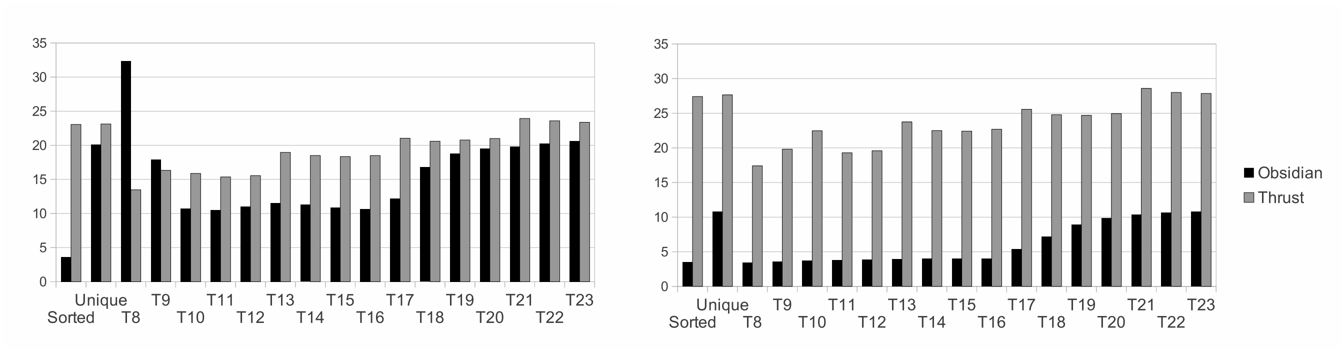

Figure 10. On the left, counting sort is compared to the Thrust library. On the right, occurrence sort is compared to the `sort` followed by `unique` function in Thrust. Both charts show running time for 8 Million elements.

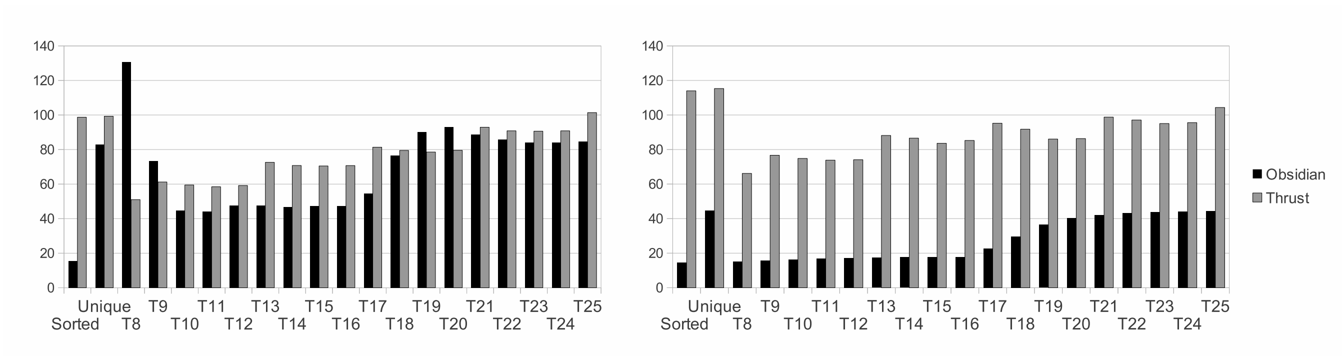

Figure 11. This chart shows same comparison as figure 10, but for 32 Million elements.

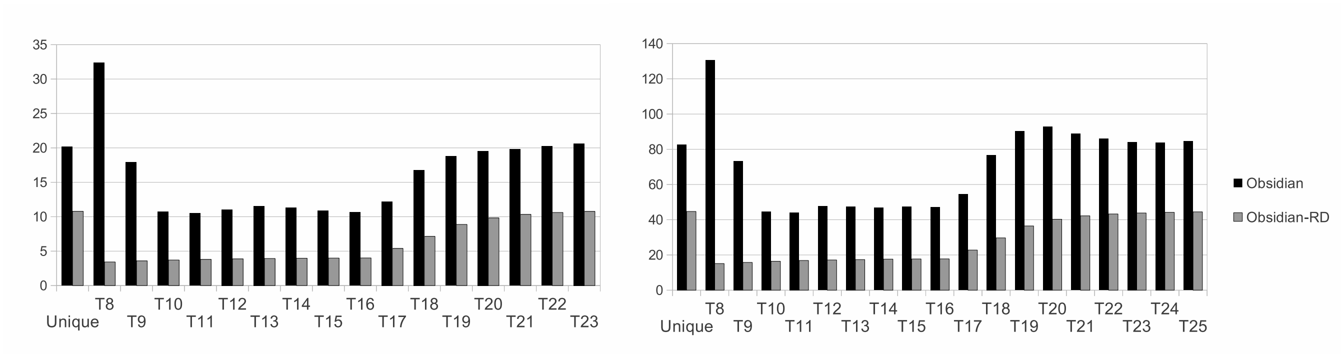

Figure 12. These two charts compares occurrence sort to counting sort. The left chart is for 8 Million elements and the right chart is for 32 Million elements.

the capability of streaming data from the main memory to the GPU to our algorithms.

In counting sort, an auxiliary array holding the count of how many times each element occurs in the input array, the histogram array, is created. The size of this array is based on the range of elements occurring in the input. This means that more memory is needed for larger ranges and may imply that counting sort is suitable only up to some point. A related algorithm, known as radix sort [8], addresses this issue by using multiple counting sort like passes, one for each digit position and thus limits this auxiliary memory usage to an array of size related to the radix (number of unique digits) used in the representation of the values being sorted. During our exploration of the counting sort algorithm on a GPU, we were not much concerned by memory usage; all data and auxiliary arrays fit easily in the device memory. The charts (10, 11 and 12) indicate that our implementation offers the most benefit in the ranges labelled as $T10$ to $T17$; these are in fact relatively small ranges (1024 to 131072 unique values). Our current hypothesis is that the decrease in benefit of counting sort from range $T18$ and onward is due to more and more complicated memory access patterns in the reconstruction phase. However, more in-depth profiling is needed to understand where the decrease in benefit comes from.

Obsidian is work in progress but this paper shows a step forward in its capabilities. New kinds of kernels that operate on global arrays can be implemented. We also added atomic operations to the language. However, when dealing with atomic operations, we

```
histogram<<<NB,BS,0>>>(dinput,dhistoutput);

scan256<<<NB,BS,2052>>>
    (dhistoutput,dscanoutput+1,dmaxs+1);

//dmaxs needs to be scaned (dmaxs2 is ignored)
scan256<<<SMALL,BS,2052>>>
    (dmaxs+1,dmaxs+1,dmaxs1+1);

scan16<<<1,SMALL,132>>>
    (dmaxs1+1,dmaxs1+1,dmaxs2);

// distribute can be in-place regarding dresult
// since it reads and writes in the exact
// same location (per thread)
distribute<<<SMALL,BS,0>>>
    (BS,dmaxs1,dmaxs+1,dmaxs+1);
distribute<<<NB,BS,0>>>
    (BS,dmaxs,dscanoutput+1,dscanoutput+1);

reconstruct<<<NB,BS,0>>>
    (dscanoutput,dresult);
```

Figure 14. The kernel launch sequence used in the timing of counting sort for 1 Million elements. Any other input data size would vary only some of the prefix sum (scan in the code) sizes.

express the Obsidian program at a very low level. One task for future work is to find some way to raise the level of abstraction in that area. Another future direction in exploring the counting sort algorithm is to consider further optimisations. One possibility is to work with sequentiality (more work per thread). Continuing exploration in that direction could lead to further improvements of Obsidian when it comes to low-level control.

In the current version of Obsidian, we need to write some CUDA code by hand in order to implement the algorithm we describe. In the future we want to be able to do all coding without leaving Haskell.

Acknowledgments

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

References

[1] Wikipedia article: Counting sort. http://en.wikipedia.org/wiki/Counting_sort.

[2] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Proc. of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11. ACM, 2011.

[3] 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*, DAMP '12, 2012.

[4] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. URL http://conal.net/papers/jfp-saig/.

[5] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, 2007.

[6] Joel Svensson. Obsidian: GPU Kernel Programming in Haskell. Technical Report 77L, Computer Science and Enginering, Chalmers University of Technology, Gothenburg, 2011. Thesis for the degree of Licentiate of Philosophy.

[7] T. Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *Proc. of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12. Eurographics Association, 2012.

[8] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. ISBN 0-201-89685-0.

[9] V. Kolonias, A. G. Voyiatzis, G. Goulas, and E. Housos. Design and implementation of an efficient integer count sort in CUDA GPUs. *Concurrency And Computing: Practice And Experiance*, 23(18), Dec. 2011.

[10] J. Krueger, M. Grund, I. Jaeckel, A. Zeier, and H. Plattner. Applicability of GPU Computing for Efficient Merge in In-Memory Databases. In *The Second International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures (ADMS 11)*, 2011. http://www.adms-conf.org/p19-KRUEGER.pdf.

[11] A. Kulkarni and R. R. Newton. Embrace, Defend, Extend: A Methodology for Embedding Preexisting DSLs, 2013. Functional Programming Concepts in Domain-Specific Languages (FPCDSL'13).

[12] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. *SIGPLAN Not.*, 45(11), 2010.

[13] NVIDIA. CUDA C Programming Guide, . URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[14] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110, . URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[15] NVIDIA. NVIDIA Thrust Library, . URL https://developer.nvidia.com/thrust.

[16] O. Olsson, M. Billeter, and U. Assarsson. Clustered deferred and forward shading. In *Proc. of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics*, EGGH-HPG'12. Eurographics Association, 2012.

[17] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10), 2008.

[18] J. Sklansky. Conditional Sum Addition Logic. *Trans. IRE*, EC-9(2), June 1960.