Efficient GPU Implementation of Affine Index Permutations on Arrays

Efficient GPU Implementation of Affine Index Permutations on Arrays

Mathis Bouverot-Dupuis
ENS Paris
Paris, France
mathis.bouverot@ens.psl.eu

Mary Sheeran
Chalmers University
Gothenburg, Sweden
mary.sheeran@chalmers.se

Abstract

Optimal usage of the memory system is a key element of fast GPU algorithms. Unfortunately many common algorithms fail in this regard despite exhibiting great regularity in memory access patterns. In this paper we propose efficient kernels to permute the elements of an array. We handle a class of permutations known as Bit Matrix Multiply Complement (BMMC) permutations, for which we design kernels of speed comparable to that of a simple array copy. This is a first step towards implementing a set of array combinators based on these permutations.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; **Parallel programming languages**; • **Software and its engineering** → **Software performance**.

Keywords: GPU, data-parallelism, functional languages

ACM Reference Format:
Mathis Bouverot-Dupuis and Mary Sheeran. 2023. Efficient GPU Implementation of Affine Index Permutations on Arrays. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '23), September 4, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3609024.3609411

1 Introduction

In GPU algorithms, memory access is often the performance bottleneck. Consider the following low-level GPU kernel that transforms an array of size 2^n :

```
kernel(int* input, int* output)
{
  i <- thread_id();
  x <- input[bit-rev(n, i)];
  output[i] <- f(x);
}
```


FHPNC '23, September 4, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0296-9/23/09…$15.00
https://doi.org/10.1145/3609024.3609411

Here, 2^n threads are launched that each read and write a single element, modifying it using a function f. The read position is computed using the `bit-rev` function that performs bit-reversal on the index i viewed as a list of n bits, so that `bit-rev`$(4, 7)$ transforms $7 = 0b0111$ to $14 = 0b1110$.

Behind this deceivingly simple access pattern lies terrible performance; the read is typically an order of magnitude slower than the write on modern GPUs. Despite the great degree of regularity present in this memory access pattern, it yields uncoalesced memory accesses that force the reads from different threads to be serialized.

While bit reversal often has a hardware or low level implementation, many other transformations (such as those in sorting networks) exhibit a similar degree of regularity that is not fully exploited by GPUs. To this end, we use an alternative way of describing array indexing that allows many regular access patterns to be compiled to efficient GPU code. We view indices into an array of size 2^n as binary vectors of size n (vectors in F_2^n) and focus on affine transformations in F_2^n, the so-called Binary Matrix Multiply and Complement (BMMC) transformations [4, 7]. We study the BMMC permutations because they enable reasoning about and implementation of the sets of combinators that we have earlier considered for both software and hardware design [2, 3].

The contributions of this paper are as follows :

- We show how to efficiently implement a specific class of array permutations where the mapping between indices is given by a BMMC[1].
- We conduct an empirical evaluation of our kernels, both in the worst case and the average case.
- We show preliminary work on using BMMC permutations to compile high level array combinators.

More precisely, we show how to implement a subclass of BMMC permutations - namely *tiled* BMMC permutations - almost as fast as a simple array copy, and how to factorize any BMMC as the product of at most two tiled BMMCs.

2 Background : GPU Programming

2.1 A Simple GPU Model

This section presents the relevant parts of a simple GPU model that we will use to justify our optimizations. There

[1]The code to generate and benchmark our CUDA kernels is publicly available at https://github.com/MathisBD/bmmc-perms-gpu.

are two key aspects to this model : the execution model and the memory hierarchy. For a more in depth discussion of a similar machine model we refer the reader to chapter 4 "Parallelism and Hardware Constraints" of Henriksen's thesis on Futhark [9].

Regarding terminology, there are unfortunately two distinct sets of terms; we will be using the CUDA set, which differs from but also overlaps with the OpenCL set.

The execution model follows an SIMT (single instruction multiple threads) design; a large number of threads are launched concurrently, all executing the same code. Threads are uniquely identified by a thread identifier, which often dictates how they will behave. They are organized according to the following hierarchy :

Kernels are the top-level scheduling unit : all threads in a kernel execute the same code. To obtain good performance it is necessary that a kernel have many threads (typically at least a 100 thousand), and in general there is no kernel-level synchronization possible between threads. A GPU program consists of one or several kernels that are run sequentially.

Thread blocks are the unit at which thread synchronization - whether it be memory or execution synchronization - can happen. In kernels where the threads do not need synchronization (map-like kernels), the thread group is mostly irrelevant. Maximum thread block size is hardware dependent : typical sizes are 256 and 1024 threads for AMD and NVIDIA GPUs respectively.

Warps form the basic unit for execution and scheduling. Threads inside a single warp execute instructions in lockstep, including memory access instructions, so that all memory transactions of a warp must have completed before it can advance to the next instruction. Warp size is hardware dependent, although 32 threads is typical.

Kernels usually launch many more threads than can be run concurrently. In this case, threads are launched one thread block at a time, with new thread blocks being swapped in as previous blocks finish execution. The order in which blocks are scheduled is by increasing thread identifier : this means that at any given time the threads currently in flight cover a contiguous subset of the thread identifiers.

The other side of the coin is the GPU memory hierarchy, which reflects the thread hierarchy :

Global memory is large off-chip memory (typically on the order of several GiB). This is where the CPU copies data to and from, and is where the inputs and outputs to a kernel reside. If accessed properly global memory has a much larger bandwidth than usual CPU RAM.

Shared memory is smaller and shared by all threads in a thread group. It usually functions as a cache used by thread blocks : however unlike traditional caches, the programmer is responsible for loading data in and out of shared memory.

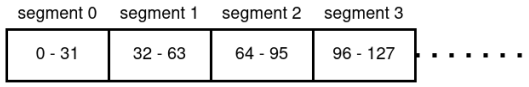
Registers are small bits of memory private to each thread. Although very fast, the number of registers per thread is limited. Kernels that require many registers per thread will cause fewer threads to run concurrently.

2.2 Optimizing Memory Access

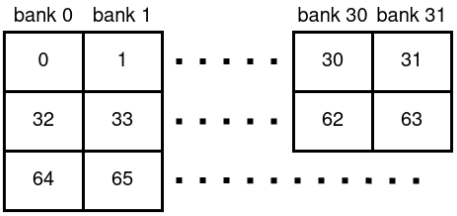
In contrast to CPUs, GPU programmers must manually manage most of the memory hierarchy in order to get the best performance. Hardware managed caches, while also present on GPUs, are of less importance; most performance benefits come from mechanisms that allow certain memory transactions to be answered concurrently, known as *coalesced* and *bank conflict free* memory accesses.

Global memory is divided into contiguous segments - typically 32, 64 or 128 bytes - that form the basic unit for memory transactions (see Figure 1a). The size of a segment is much larger than what can be accessed by a single thread in a given instruction, and in general the memory transactions needed for the individual threads in a warp are serialized. However modern GPUs ensure that the memory accesses from a given warp that fall in the same segment are *coalesced* into one transaction (the order of addresses within a segment does not matter). To obtain optimal memory performance the set of segments accessed by a warp must be as small as possible. Memory access patterns that fail to exploit coalescing can lead to over an order of magnitude decrease in bandwidth.

Shared memory is divided into *banks* (typically 32). Contrary to global memory segments, shared memory banks are not contiguous but rather interleaved at the 32-bit word granularity : see Figure 1b for an illustration. Accesses by a warp that fall in the same memory bank must be serialized, but accesses to different banks can be answered concurrently. If threads within a warp access the memory banks in an imbalanced way, a bank conflict occurs, potentially causing a decrease in shared memory bandwidth of up to 32 times.



(a) Global memory layout assuming each segment is of size 128 bytes.



(b) Shared memory layout.

Figure 1. Typical layouts for global and shared memory. The numbers correspond to addresses of 32-bit words.

2.3 An Example : Matrix Transposition

To help gain some intuition on GPU programming we walk through an example kernel. Let M be a two-dimensional matrix of size (N, N). We would like to write a kernel that performs matrix transposition on $M : M[i, j] \leftarrow M[j, i]$ for all i and j, and M is stored in row major order in both the input and output.

If we assume that N is a power of two, we can write the following kernel (in CUDA-like pseudocode) :

```
kernel transpose_naive(int* input, int* output)
{
  size_t i = blockIdx.y * blockDim.y + threadIdx.y;
  size_t j = blockIdx.x * blockDim.x + threadIdx.x;

  output[j * N + i] = input[i * N + j];
}
```

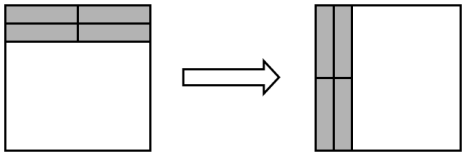
The variables `blockIdx`, `blockDim` and `threadIdx` are three-dimensional vectors that store, for each thread, the corresponding block index, block size and thread index within its block.

We invoke this kernel with a grid of $(N/32) * (N/32)$ thread blocks with each thread block being of size $32 * 32$. When using a two-dimensional indexing scheme for thread blocks (as is done here) the index of a thread within its block is given by `threadIdx.y * blockDim.x + threadIdx.x`, and warps correspond to bundles of 32 threads that have contiguous indices. In this case, each warp corresponds to a single value for `i` and 32 contiguous values for `j`. This means that the first memory access (reading the input) is fully coalesced, but the second memory access (writing the output) is not.

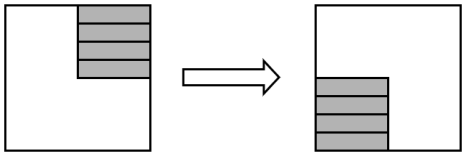
To ensure that both memory accesses are coalesced we can make use of shared memory. Each thread block will process a square tile of the input of size $32 * 32$ (compare this to the naive kernel where each block processes a contiguous patch of the input, see figure 2 for an illustration). When reading in the tile, each warp will process a single row of the tile, but when writing out the tile, each warp will process a single column of the tile :

```
kernel transpose_tiled(int* input, int* output)
{
  shared tile[32*32];

  size_t block_i = blockIdx.y * blockDim.y;
  size_t block_j = blockIdx.x * blockDim.x;
  size_t i = threadIdx.y;
  size_t j = threadIdx.x;

  tile[i * 32 + j] =
      input[(block_i + i) * N + block_j + j];
  synchronize();
  output[(block_j + i) * N + block_i + j] =
      tile[j * 32 + i];
}
```

Each thread group uses an array `tile` of size $32 * 32$ in shared memory. We have to manually `synchronize()` threads within a thread block so that the tile for this block is fully populated before we start writing out. The tile processed by a given block has its upper left corner at position (`block_i`, `block_j`) in the input, which corresponds to the tile with upper left corner (`block_j`, `block_i`) in the output.



(a) Naive transpose.



(b) Tiled transpose.

Figure 2. The shaded area represents the part of matrix M that is accessed by a single thread group, in both input and output. This area is further divided into regions that are accessed by individual warps (for visual clarity, we drew only 4 warps per thread group; in reality there would be 32).

We measured the performance of the above transpose kernels for matrices of size $2^{15} * 2^{15}$ on an NVIDIA RTX4090 GPU. The effective memory bandwidth achieved in each case is computed by comparing the running time to that of a simple copy kernel :

kernel	running time	effective bandwidth
copy	9.3 ms	100%
naive transpose	26.4 ms	35.2%
tiled transpose	12.2 ms	76.2%

The tiled version is over twice as fast as the naive version. Further optimizations can bring the running time even closer to the copy kernel : we refer the interested reader to the NVIDIA tutorial [8].

3 Key Ideas

Viewing indices into arrays of size 2^n as binary vectors of length n allows us to restrict our attention to certain well-behaved transformations on indices. Arguably the simplest transformations according to this point of view are linear and affine mappings, i.e. mappings between source indices x and target indices y such that :

$$y = Ax + c$$

where A is an (n, n) binary matrix, c is a binary vector of length n and all arithmetic is done modulo 2 (i.e. in F_2 the finite field with two elements). If we expand this formula,

each bit of y is given by :

$$y_i = \left(\sum_{0 \leq j < n} a_{ij} x_j \right) + c_i$$

Many common transformations on indices can in fact be expressed in this way. For instance, transposing a matrix of size $4 * 4$ can be expressed as follows :

$$y_i = x_{(i+2) \ \% \ 4} \quad \texttt{i.e.} \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The above matrix has exactly one non-zero entry per row and per column. Invertible matrices of this form are called *permutation matrices* and simply permute the bits of the input index. In the above example, the index with bits $[x_0, x_1, x_2, x_3]$ is mapped to $[x_2, x_3, x_0, x_1]$, so that index $6 = 0b0110$ is mapped to index $9 = 0b1001$.

When the matrix A is a permutation matrix and the complement vector c is 0 we call (A, c) a Bit Permute (BP) transformation. Bit-reversal is thus a BP transformation :

$$y_i = x_{n-1-i} \quad \texttt{i.e.} \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The complement vector is also useful. Here is an example of using it to define a transformation that reverses an array of size 16 :

$$y_i = x_i + 1 \quad \texttt{i.e.} \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

In this case the matrix A is also a permutation matrix (corresponding to the identity permutation) but the complement vector c is non-zero : we call such a transformation a Bit Permute Complement (BPC).

In the most general case, A is any invertible matrix (over F_2) and c is any vector, giving a Bit Matrix Multiply Complement (BMMC) transformation. The invertibility requirement for A ensures that the transformation defines a permutation on arrays. While not all permutations on arrays can be expressed in such a way, the preceding examples should convince the reader that this class includes many of the common cases. Note that permutations on an array whose size is not a power of 2 do not fall in the BMMC class. For instance transposing a matrix of size $(7, 5)$ is not a BMMC permutation, whereas transposing a matrix of size $(16, 8)$ is.

BMMCs were studied in the context of data-parallel programming in the 1990s by Cormen, Edelman and their coauthors. Both exploited the power of linear algebra, such as various matrix decompositions or Gaussian elimination, inspiring this work [4, 7]. For example, Cormen proposed asymptotically optimal implementations for BMMC permutations on the disk I/O model [4].

We aim to use BMMC permutations to provide an efficient implementation for high-level combinators that allow the programmer to describe data access patterns concisely. In this paper, we give one example of such a combinator, called `parm`. The expression `parm mask f xs` partitions the array `xs` of size 2^n into two equally sized subarrays according to the n bit `mask`, applies `f` to each subarray and stitches the resulting arrays back together according to the `mask`. The element at index `i` is assigned to the first or second subarray according to the dot product `i` $*$ `mask` in F_2 : see Figure 3 for examples.

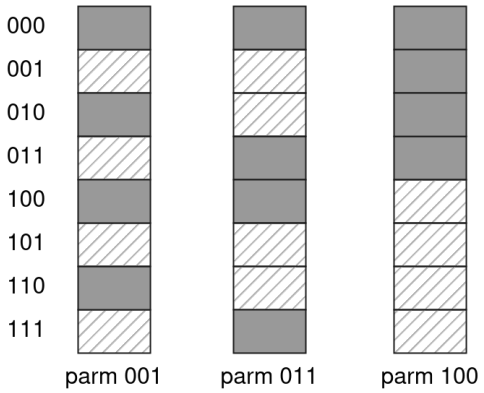


Figure 3. Applying `parm` to an array of size 8 with different masks written in binary. The first and second subarrays are represented respectively with a solid and dashed background.

In section 7.2, we show how to efficiently implement the `parm` combinator in terms of BMMC permutations. In fact, for any mask m we can find a matrix A such that :

$$\texttt{parm}\ m\ f = \texttt{bmmc}\ (A^{-1}, 0) \circ \texttt{parm}\ 2^{n-1}\ f \circ \texttt{bmmc}\ (A, 0)$$

where $\texttt{parm}\ 2^{n-1}\ f$ applies f to the first and second halves of the input array, and composition is from right to left. For instance, the BMMC corresponding to `parm 0b0011` is

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

We are currently working on implementing `parm` and related combinators in the Futhark programming language, a high-level functional language that can compile to efficient GPU code [10]. These combinators benefit greatly from fusion rules such as the following :

$$\texttt{bmmc}\ (A, c) \circ \texttt{bmmc}\ (B, d) = \texttt{bmmc}\ (AB, Ad + c)$$

Some challenges arise when compiling uses of the bmmc combinator in nested parallel code. In fact for any BMMC

(A, c) of size n and any mask m we can find another BMMC (A', c') of size $n+1$ such that :

$$\texttt{parm}\ m\ (\texttt{bmmc}\ (A, c)) = \texttt{bmmc}\ (A', c')$$

The main contribution of this paper is to give an efficient implementation for a class of BMMC permutations that we call *tiled* BMMC permutations. These include all BPC permutations, such as transpose and bit reverse. We show how to generalize the matrix transposition kernel to tiled BMMCs and compare the impact of different optimizations in sections 4 and 6. The final kernels we obtain are fully coalesced and bank-conflict free, reaching on average 90% of the maximum effective memory bandwidth.

Finally, we show in section 5 how to use linear algebra techniques to decompose any BMMC A as a product $A = T_1 T_2$ of two tiled BMMCs. The permutation defined by A can then be efficiently realized by first applying the kernel for T_2 followed by the kernel for T_1.

It should be noted that we assume an offline setting, i.e. that the BMMC matrix and complement vector are known in advance (before generating the CUDA code for the kernel). This is in accordance with our aim to implement the techniques described in this paper in the Futhark compiler.

4 Implementing BPC Permutations

In this section, we explain how to generalize the transpose kernel from section 2.3 to arbitrary BPC permutations. We start by introducing simple tiling to enable coalesced memory access before gradually adding further optimizations. As a running example the reader can inspect the different kernels generated for the bit-reverse permutation in the appendix.

4.1 Ensuring Coalesced Accesses

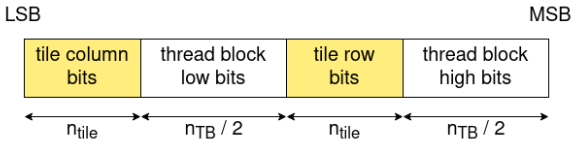
The first step is to define the notion of tile for an arbitrary BPC (p, c), where p is a permutation on $\{0, \ldots, n-1\}$ and c is a complement vector. We start by partitioning the bits of input indices as follows :

- The tile column bits are the n_tile lower bits.
- The tile row bits are the n_tile bits such that $p(bit\_index) < n\_tile$.
- The tile overlap bits are the n_over bits that are both tile column and tile row bits.
- The thread block bits are the n_TB remaining bits.

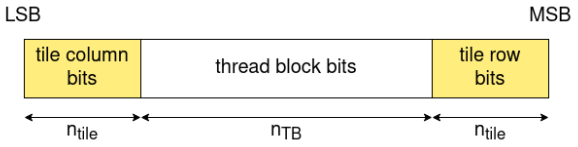
See Figure 4 for an illustration. In our implementation, we choose n_tile to be equal to the logarithm of the warp size : $n\_tile = 5$.

We also define some notation for dealing with indices :

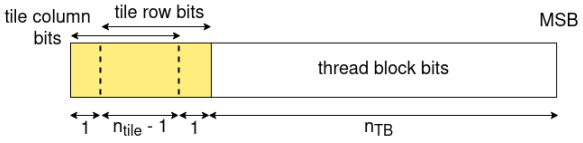
- `stitch_col(col, TB, row)` forms an index by using `col` for the n_tile tile column bits, `TB` for the n_TB thread block bits and `row` for the $n\_tile - n\_over$ remaining tile row bits.



(a) Transpose (for a square matrix).



(b) Bit reverse : $p(i) = n - 1 - i$



(c) Cyclic shift : $p(i) = \begin{cases} i-1 & \texttt{if } i > 0 \\ n-1 & \texttt{if } i = 0 \end{cases}$

Figure 4. Partition of input index bits for different permutations. In the first two cases $n\_over = 0$, and in the third case $n\_over = n\_tile - 1$. Note that in general the tile row bits need not be contiguous, and so do the thread block bits.

- `stitch_row(col, TB, row)` forms an index by using `row` for the n_tile tile row bits, `TB` for the n_TB thread block bits and `col` for the $n\_tile - n\_over$ remaining tile column bits.
- `stitch_tile_col(col, row)` forms an index as in `stitch_col`, but deletes the thread block bits.
- `stitch_tile_row(col, row)` forms an index as in `stitch_row`, but deletes the thread block bits.

We show some examples of using these functions for the cyclic shift permutation of Figure 4. This permutation shifts the bits of the input index by one position towards the LSB and moves the LSB to the MSB, so that :

$$\texttt{cyclic\_shift}(\texttt{0b11001}, 5) = \texttt{0b11100}$$

In these examples $n = 10$ and $n\_tile = 5$ (thus $n\_over = 4$ and $n\_TB = 4$), and we follow the usual convention for binary literals of writing the LSB to the right and MSB to the left :

```
stitch_col(11010, 1100, 1) = 1100111010
stitch_tile_col(11010, 1) = 111010
stitch_row(0, 1011, 00110) = 1011001100
stitch_tile_row(0, 00110) = 001100
```

Note that when $n\_over = 0$, `stitch_col` and `stitch_row` are identical, and `stitch_tile_col` and `stitch_tile_row` are also identical. We refer the reader to the appendix for some intuition on how these stitching functions are translated to CUDA instructions.

Fixing the thread block bits and choosing every possible combination of tile bits defines a single tile : the input array is thus covered by 2^{n_TB} disjoint tiles. As in the transposition case, we launch one thread block per tile, each of size $2^{n\_tile} * 2^{n\_tile-n\_over}$.

```
kernel bpc_permutation(int* input, int* output)
{
  shared tile[2^(2*n_tile - n_over)];
  size_t block = blockIdx.x;
  size_t warp = threadIdx.y;
  size_t thread = threadIdx.x;

  // Read the tile.
  tile[stitch_tile_col(thread, warp)] =
      input[stitch_col(thread, TB, warp)];

  // Synchronize
  syncthread();

  // Write the tile
  output[p(stitch_row(warp, TB, thread)) XOR c] =
      tile[stitch_tile_row(warp, thread)];
}
```

This kernel uses only coalesced memory accesses. We can easily see that when reading the input tile each warp reads 2^{n_tile} consecutive elements. This is less clear when writing the output tile. Notice that using p to permute the bits of `stitch_row(warp, TB, thread)` moves the bits of `thread` to the n_tile lower bits of the index : each warp thus writes 2^{n_tile} consecutive elements.

4.2 Avoiding Bank conflicts

The previous kernel solved the coalescing problem, but unfortunately it introduced shared memory bank conflicts, specifically in the second access to the tile in shared memory.

At this point there are two natural ways of viewing the two-dimensional tile in shared memory : we could view it as a $2^{n\_tile} * 2^{n\_tile-n\_over}$ matrix, or as a $2^{n\_tile-n\_over} * 2^{n\_tile}$ matrix. We choose the latter option as it yields an easier analysis of bank conflicts. Note that the tile is in general not square : it can have fewer rows than columns.

We can now analyse both accesses to the tile using this new lens (see Figure 5 for an illustration) :

- In the first access each warp writes a single row in the tile. Since we chose $2^{n\_tile} = 32$ this is always bank conflict-free.
- In the second access each warp reads 2^{n_over} distinct columns from the tile : in particular when $n\_over = 0$ each warp reads a single column. Note that the accessed columns are not necessarily evenly spaced.

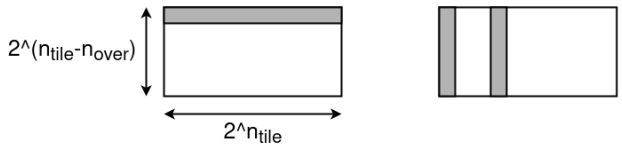


Figure 5. The two-dimensional tile in shared memory when $n\_over = 1$. The shaded region corresponds to the elements accessed by a single warp : on the left for the first access and on the right for the second access.

Accessing a matrix column-wise in shared memory results in a bank conflict. In this case, the second access is serialized into $2^{n\_tile-n\_over}$ conflict-free reads, one for each row.

To fix this conflict we change slightly the way the tile matrix is stored in shared memory : we shift each row by a given amount to the right. Elements that overflow the end of the row wrap around to the start of the row. More formally, the element at row i and column j is stored at index :

$$i * 2^{n\_tile} + (\texttt{shift\_i} + j \mod 2^{n\_tile})$$

We choose the shift for each row depending on the permutation p, but note that no matter how we choose the shifts the first access to the tile will always remain conflict-free. We make the following choice :

$$\texttt{shift\_i} = \texttt{stitch\_tile\_row}(i, 0)$$

For instance when $n\_over = 0$ we have `shift_i` $= i$. We can now analyse the second access again. Each thread accesses the shared memory tile at position (i, j) where :

$$\begin{aligned} i &= \texttt{stitch\_tile\_row}(\texttt{warp}, \texttt{thread}) \,/\, 2^{n\_tile} \\ &= \texttt{thread} \,/\, 2^{n\_over} \\ j &= \texttt{stitch\_tile\_row}(\texttt{warp}, \texttt{thread}) \mod 2^{n\_tile} \\ &= \texttt{stitch\_tile\_row}(\texttt{warp}, \texttt{thread} \mod 2^{n\_over}) \end{aligned}$$

This element is in the following bank (modulo 2^{n_tile}) :

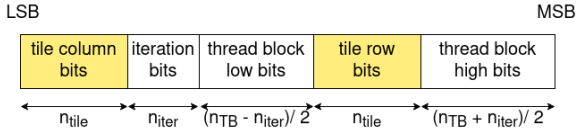
$$\begin{aligned} &\texttt{bank}(\texttt{warp}, \texttt{thread}) \\ &= \texttt{shift\_i} + j \\ &= \texttt{stitch\_tile\_row}(i, 0) + j \\ &= \texttt{stitch\_tile\_row}(\texttt{thread} \,/\, 2^{n\_over}, 0) + \\ &\quad \texttt{stitch\_tile\_row}(\texttt{warp}, \texttt{thread} \mod 2^{n\_over}) \\ &= \texttt{stitch\_tile\_row}(\texttt{warp}, 0) + \\ &\quad \texttt{stitch\_tile\_row}(\texttt{thread} \,/\, 2^{n\_over}, \texttt{thread} \mod 2^{n\_over}) \end{aligned}$$

The final call to `stitch_tile_row` is a bit permutation of `thread`. This means that in the second access each warp accesses every bank once. The resulting kernel is fully conflict-free.

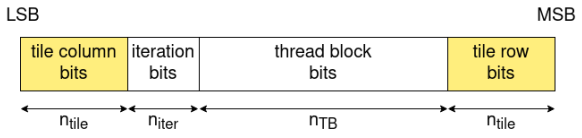
4.3 Amortizing Index Computations

The running time of the transpose kernel shown in the introduction is almost completely spent on memory operations. This is not the case for more complex permutations (for instance when $n\_over > 0$ or when the tile row bits are not contiguous); the scalar instructions performed by each thread to stitch the bits of input and output indices account for a non-negligeable portion of the running time. We can reduce this overhead by having each thread process 2^{n_iter} input indices instead of only one (typically $n\_iter = 3$).

We modify the partition of input index bits by splitting the thread block bits into two parts : the lower n_iter bits become the iteration bits and the upper bits become the new thread block bits (see Figure 6 for an illustration). The `stitch_row` and `stitch_col` functions are modified accordingly.



(a) Transpose (for a square matrix).



(b) Bit reverse.

Figure 6. Partition of input index bits for different permutations, accounting for the iteration bits. The shaded areas represent the tile bits.

Each thread block processes 2^{n_iter} tiles : it reads the tiles sequentially, synchronizes the threads, and writes the tiles sequentially. For instance the read step becomes :

```
// Read the tiles.
for (int iter = 0; iter < 2^n_iter; iter++) {
    tiles[iter][stitch_tile_col(thread, warp)] =
        input[stitch_col(thread, iter, TB, warp)];
}
```

The advantage of writing the kernel this way is that most index computations can be pulled out of the `for` loop. Only the parts that depend on `iter` need remain in the loop (see the appendix for an example). The average amount of scalar instructions per input element is thus greatly reduced.

5 Implementing BMMC Permutations

5.1 Tiled BMMCs

It is straightforward to extend the kernels developed in the previous section to a class of BMMCs slightly larger than BPCs, namely tiled BMMCs. A tiled BMMC (A, c) is a BMMC corresponding to a permutation that can be implemented using the tiled kernel outlined above. The minimal requirements on the matrix A are that we can find a set of columns $i_1, \ldots, i_{n\_tile}$ such that :

- The sub-matrix formed by the first n_tile rows and the columns $i_1, \ldots, i_{n\_tile}$ is invertible.
- The sub-matrix formed by the last $n - n\_tile$ rows and the columns $i_1, \ldots, i_{n\_tile}$ is equal to 0.

See Figure 7 for an illustration. Note that a BPC is always a tiled BMMC; in this case the columns $i_1, \ldots, i_{n\_tile}$ are exactly the indices of the tile row bits.

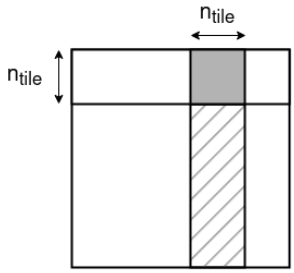


Figure 7. Decomposition of a tiled BMMC. The shaded sub-matrix is invertible and the dashed sub-matrix is equal to 0. In this example the columns $i_1, \ldots, i_{n\_tile}$ are contiguous.

When implementing the tiled kernel, the bits of each input index are now partitioned in the following way :

- The tile column bits are the n_tile lower bits.
- The tile row bits are the n_tile bits $i_1, \ldots, i_{n\_tile}$.

The tile overlap bits and thread block bits are defined as previously. The only modification we have to make to the kernel is to change the calculation of the output address

```
p(stitch_row(...)) XOR c
```

to use a matrix multiplication instead :

```
A * stitch_row(...) XOR c
```

Bank conflicts can now be eliminated in the same way as for BPC permutations. However, the next optimization (amortizing the cost of index computations) cannot be applied to tiled BMMC permutations, as it relies on the sparseness of BPC matrices. We show the exact performance impact in section 6.

5.2 Factorizing BMMCs into Tiled BMMCs

The main use case for tiled BMMCs is to provide an implementation for arbitrary BMMC permutations. Using the Lower-Upper (LU) decomposition we show that any BMMC can be factorized into a product of at most two tiled BMMCs.

Let (A, c) be a BMMC. There exist matrices U, L and P such that :

$$A = ULP$$

where U is an upper triangular matrix, L is a lower triangular matrix and P is a permutation matrix.

Observe that U is the matrix of a tiled BMMC (using the first n_tile columns) and P is the matrix of a BPC, but L has no such property. We can factorize A in a slightly different way, using the matrix R corresponding to bit-reverse (see section 3) such that $R_{ij} = 1$ exactly when $i + j = n - 1$ (thus R^2 is the identity matrix) :

$$A = (UR)(RLP)$$

Both factors in this new decomposition are matrices of tiled BMMCs (see Figure 8) :

- UR using the columns $n - n\_tile, \dots, n-2, n-1$.
- RLP using the columns $p(n-n\_tile), \dots, p(n-2), p(n-1)$.

The permutation defined by (A, c) can thus be realized by first permuting using $(RLP, 0)$ and then using (UR, c).

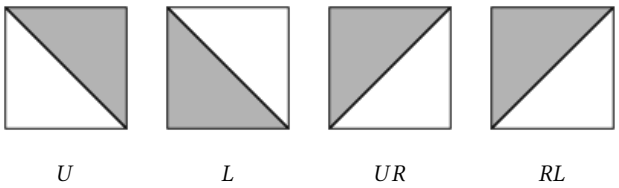


Figure 8. The non-zero entries in each matrix can only occur in the shaded area.

6 Results

We implemented the kernels outlined above in CUDA : we use Haskell to generate a CUDA kernel for each permutation. We refer the reader to the appendix for an example of the naive and various optimized BPC permutation kernels.

We used CUDA events to measure the running time of each kernel on a NVIDIA RTX4090 GPU and averaged each measurement across 1000 runs. Unless otherwise noted, all arrays contain 32-bit elements. We report the impact of different optimizations in Figure 9 :

- The *tile* optimization refers to the tiling optimization described in section 4.1.
- The *banks* optimization refers to the shared memory bank conflict optimization described in section 4.2.
- The *iters* optimization refers to the iteration optimization described in section 4.3. As explained at the end of section 5.1 this is only applicable to BPC permutations, not to tiled or arbitrary BMMC permutations.

The *tile* optimization yields the largest speedup. For the other two optimizations, we report only the additional speedup when they are added to *tile*.

Our optimized BPC permutation (*tile* + *banks* + *iters*) is about as fast as a simple copy, whereas our optimized BMMC permutation (*tile* + *banks*) is about half as fast as a simple copy. This is because a BMMC permutation is implemented as two tiled kernels and thus does twice the work of a BPC permutation which is implemented as a single tiled kernel. The cost of the binary matrix-vector product performed by each thread in the tiled BMMC kernel accounts for only a few percent of the total running time.

The first column (corresponding to the naive kernels) deserves some explanation. On average, a BPC permutation is much faster than the worst case (corresponding to bit-reversal). This is because a random BPC permutation is likely to have $n\_over > 0$, which means that with the naive kernel each warp writes to only 16 (when $n\_over = 1$) or even 8 (when $n\_over = 2$) global memory segments instead of 32 in the worst case : the naive kernel is already somewhat coalesced. On the contrary, when choosing a random BMMC permutation and factorizing it as in section 5.2, the resulting tiled BMMC permutations almost always have n_over equal to 0, meaning that with the naive kernel each warp writes to 32 global memory segments.

Figure 10 shows that our kernels are close to optimal in terms of memory bandwidth : the optimized BPC and BMMC permutations reach respectively 92% and 86% of the maximum effective bandwidth. Note that memory bandwidth is a measure of how well a memory-bound GPU program uses the memory system and does not directly reflect the program's running time, as the latter also depends on how much data is transferred to and from memory. Recall that the BMMC implementation does twice as much memory transfers as the BPC implementation, which explains why the last two columns are similar although BMMC permutations are twice as slow.

Figure 11 shows the speedup we obtain using all optimizations compared to the naive version for different array sizes. Compared to Figure 9, for arrays of size smaller than 2^{24} we get a lower speedup in the random BMMC and bit-reverse case but a higher speedup in the random BPC case (in all cases the speedup is greater than 1). We do not report data for arrays of size smaller than 2^{20} :

- For arrays of size smaller than 2^{20}, the running time of permutation kernels - both naive and optimized - is only a couple microseconds, which is very close to the GPU clock precision (half a microsecond according to the CUDA Runtime API [13], section 6.5 "Event Management").
- GPUs need a very large amount of threads to be *saturated*, i.e. to be able to hide global memory latency by switching threads. This is not anymore the case when permuting a single small array : for instance with the optimized BPC permutation kernel and an array of size 2^{18} we would launch $2^{15} = 32768$ threads, which is not enough to saturate the RTX4090 GPU used for benchmarking.

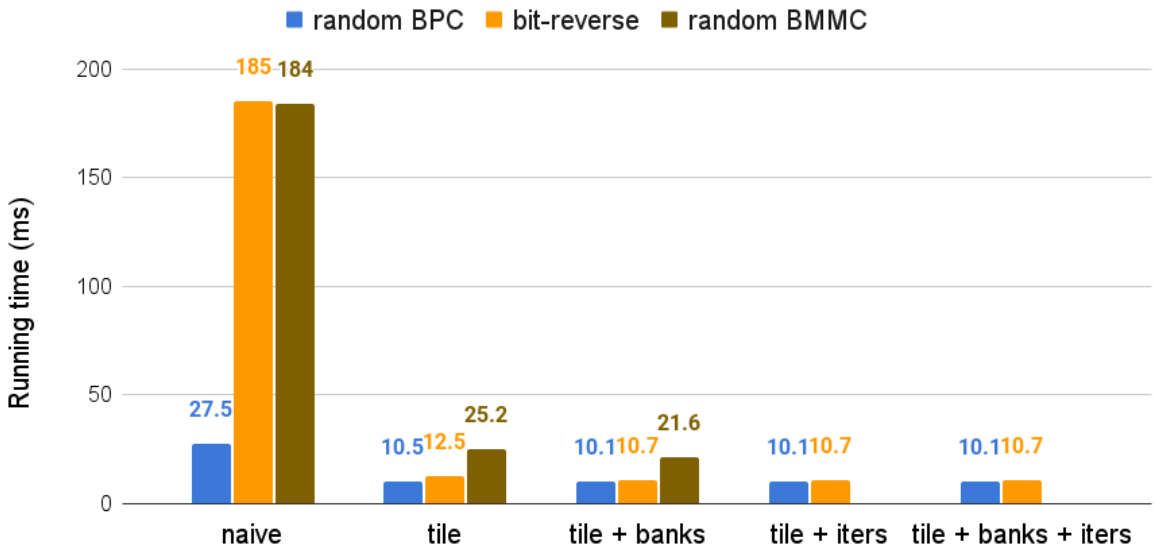


Figure 9. Impact of the different optimizations on running time, for random BPC and BMMC permutations, and for a particular BPC permutation (bit-reversal, the slowest BPC permutation) on arrays of size 2^{30}. For comparison, the running time of a copy kernel was 9.3 ms. The *iters* optimization does not apply to BMMC permutations (see section 5.1).

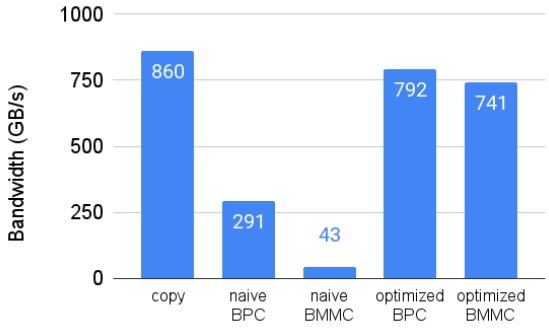


Figure 10. Global memory bandwidth of our kernels (both naive and with all optimizations), measured on arrays of size 2^{30}. The first column shows that the maximum effective bandwidth of 860 GB/s is lower than the maximum theoretical bandwidth, which is 1008 GB/s for our GPU.

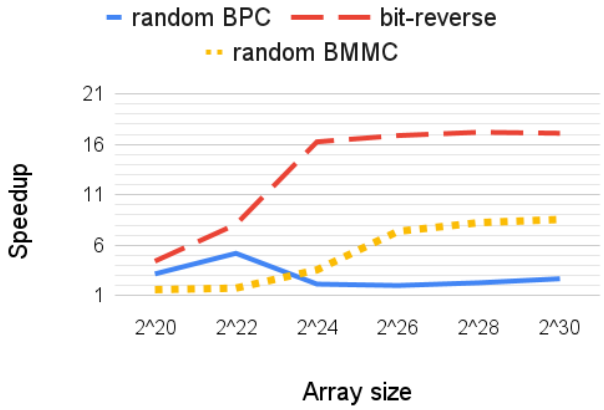


Figure 11. Speedup for different array sizes.

Our current approach for implementing BMMC permutations does have several limitations. We elaborate on the main ones here. Array sizes are restricted to powers of 2 : we have not yet found a satisfactory way to extend our results to arrays of arbitrary size. We also work in an offline setting, i.e. we assume that the BMMC matrix and complement vector are known at compile time. Extending our approach to work in an online setting would raise some difficulties :

- The decomposition of a BMMC matrix into a product of tiled BMMCs can be a costly operation for large arrays, and is poorly suited to GPUs.
- Implementing the bit-stitching functions used in section 4 in an online setting could lead to slowdown due to the additional scalar instructions we would have to generate. While this might not be an issue for BPC permutations since we can use the optimization outlined in section 4.3 to alleviate the cost of scalar instructions, this would certainly result in at least a minor slowdown for arbitrary tiled BMMC permutations.

All the measurements in this article were performed on a NVIDIA RTX4090 GPU. We could not reproduce them on an AMD GPU : we ran into some unexpected slowdowns related to global memory. Despite being fully coalesced, the running

time of our tiled permutation kernels depended heavily on the given BPC or BMMC matrix. This can be reproduced even with a kernel as simple as a tiled transpose : see Figure 12 for an example using the Futhark transpose kernel. This phenomenon only occurs when array sizes are powers of two, and as such is not an issue for most Futhark programs, but is an issue for the algorithms in this paper.

There seem to be differences in the memory architecture between AMD and NVIDIA. Our guess is that they have a different address mapping scheme and that our kernels trigger global memory bank conflicts on AMD cards, however we have not been able to prove or disprove this intuition and are open to suggestions. We refer the reader to [12] for a discussion on GPU address mapping schemes that coincidentally makes use of BMMCs.

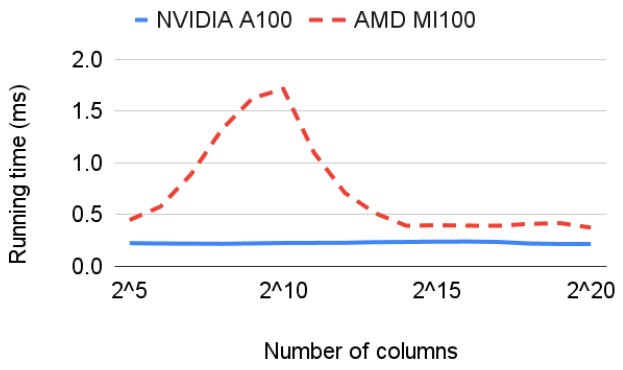


Figure 12. Running time of an optimized transpose kernel on an AMD and NVIDIA GPU, for matrices of various sizes. We keep the number of elements constant equal to 2^{25} and vary the number of columns (always a power of 2).

7 Application to the Parm Combinator

7.1 Using the Parm Combinator

As a use case of BMMC permutations we describe how they can be used to implement a high level combinator called `parm`. This is not the only useful combinator that is related to BMMCs : other examples are outside the scope of this paper, but we do plan on studying these combinators further in future work. We refer the reader to [3] for another paper using similar combinators.

Let us remind how the `parm` combinator works : it takes as input an array xs of size 2^n, an n-bit binary mask and a function f that maps arrays of size 2^{n-1} to arrays of size 2^{n-1}. The input array xs is partitioned into two sub-arrays xs0 and xs1 depending on the mask as follows (see Figures 3 and 14) :

$$\texttt{sub-array(i)} = \begin{cases} \texttt{xs0} & \texttt{if i} * \texttt{mask} = 0 \\ \texttt{xs1} & \texttt{if i} * \texttt{mask} = 1 \end{cases}$$

Where i is the index of the given element in xs and $*$ denotes the dot product in F_2. We then apply f to each sub-array and stitch them back together in exactly the same way.

We now show how to use `parm` to implement a simple sorting network, inspired by Batcher's bitonic sorting network [1] and the balanced periodic merger [6]. There has been previous effort to generate efficient GPU code for such networks : see [3] for an approach that focusses on small networks operating on arrays that fit in shared memory.

The network we study in this example is a variant of merge sort: the elements at even and odd indices are sorted separately before being merged. The following function sorts its input xs of size 2^n :

```
sort 0 xs = xs
sort n xs = let ys = parm 1 (sort (n-1)) xs
            in merge n ys
```

The merge function takes as input an array in which the two sub-arrays formed by the elements at even and odd indices are sorted and produces a sorted output. We choose to use a balanced periodic merger : Figure 13 illustrates the merging network. Data flows from left to right along the 16 horizontal lines. The vertical lines operate on two inputs and place the minimum on the top and the maximum on the bottom. Here is the corresponding pseudocode :

```
merge 0 xs = xs
merge n xs = let ys = vcolumn n xs
             in parm (2^(n-1)) (merge (n-1)) ys
```

The `vcolumn` function in turn builds a single V-shaped column with 2^n inputs in the merging network. This can be accomplished by simply interleaving two half-size V-columns using a mask equal to $3 = 0b11$ (see also Figure 3).

```
vcolumn 0 xs = xs
vcolumn 1 [x1, x2] =
  if x1 <= x2 then [x1, x2] else [x2, x1]
vcolumn n xs = parm 3 (vcolumn (n-1)) xs
```

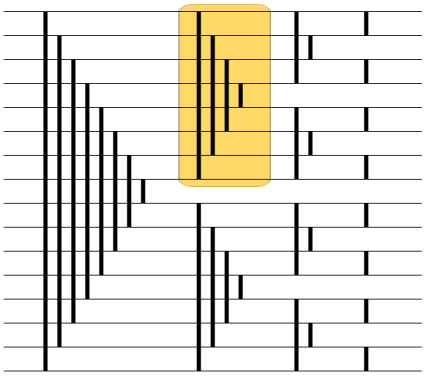


Figure 13. A 16 input balanced periodic merger. The shaded region corresponds to an 8 input V-column.

The `parm` combinator shines here because it allows the programmer to specify the sorting network in a declarative

style, leaving many opportunities for the compiler to optimize the program (in this case using BMMCs to permute arrays and obtain coalesced memory accesses).

7.2 Compiling Parm using BMMC Permutations

While the above example shows the expressiveness of `parm`, a straightforward implementation - in which the function f we apply to each sub-array reads its inputs directly from `xs` and writes directly to the output array - is not suited to GPUs. To gain some intuition on why consider the case where f makes only fully coalesced reads and writes. For most masks (think for instance of `mask = 1`) the resulting function `parm mask f` will not make fully coalesced accesses, and in fact will require twice as much memory transactions as a coalesced version would. Now take into account that `parm` is often nested many times (as in the sorting network example) and we loose all coalescing.

Our solution for compiling `parm mask f xs` while retaining coalescing is to first permute the array `xs` such that the two subarrays `xs0` and `xs1` form the first and second half of the resulting array, apply f to each half and then permute the array back. When applying f, the two sub-arrays are contiguous in memory : any coalescing behaviour of f will therefore be retained. Permuting the array twice (before and after applying f) of course adds some overhead : however these permutations are in fact BMMC permutations, allowing for an efficient implementation.

We now explain how to construct a matrix A such that :

$$\texttt{parm}\ m\ f = \texttt{bmmc}\ (A^{-1}, 0) \circ \texttt{parm}\ 2^{n-1}\ f \circ \texttt{bmmc}\ (A, 0)$$

Permuting `xs` using the BMMC $(A, 0)$ should put `xs0` into the first half and `xs1` into the second half, while preserving the order of elements within each sub-array. More formally, an element at index x in `xs` should have the index y in the result such that :

$$y_{0..n-2} = \texttt{sub-index}(x)$$
$$y_{n-1} = \texttt{sub-array}(x)$$

Where `sub-array(x)` is equal to 0 if x is in the first sub-array and 1 otherwise, and `sub-index(x)` is the new index of the element at position x in its sub-array (see Figure 14 for an example).

Notice that `sub-array(x)` is simply equal to $i * \texttt{mask}$. Finding an expression for `sub-index(x)` is slightly harder. It turns out that it is sufficient to remove the bit at index `lsb(mask)` from x, where `lsb(mask)` is the index of the least significant bit of the mask. The reader is invited to check this fact in Figure 14. This yields the following relation between x and y from which it is straightforward to construct the matrix A (a similar formula can be derived for A^{-1}) :

$$y_i = \begin{cases} x_i & \texttt{if } i < \texttt{lsb(mask)} \\ x_{i+1} & \texttt{if lsb(mask)} \leq i < n-1 \\ x * \texttt{mask} & \texttt{if } i = n-1 \end{cases}$$

index x	sub-array	sub-index
0	0	0
1	0	1
2	1	0
3	1	1
4	1	2
5	1	3
6	0	2
7	0	3

(a) The `sub-array` and `sub-index` functions.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

(b) The matrix A.

Figure 14. Constructing the matrix A for an 8 element array and a mask equal to 6 = 0b110. In this case `lsb(mask)` = 1.

It should be noted that `parm` and `bmmc` give rise to a rich set of rewrite rules that allow us to reduce the number of BMMC permutations performed in most cases, especially when nesting applications of `parm`.

8 Related Work

BMMCs were first studied by Cormen in the setting of the parallel disk I/O model introduced by Vitter and Schriver [14]. This model consists in a processor (or multiprocessor) connected to several storage devices which can be accessed in parallel, and places an emphasis on the memory system rather than on the processor. Performance in this model is measured in terms of I/O accesses. Cormen showed how to perform BMMC and BPC permutations for large on-disk arrays and proved optimality results for his implementations in terms of number of memory accesses [4, 5]. This inspired our current work, which tackles the same problem but in the context of GPUs, for which memory access performance is just as important as in the context of parallel disk I/O.

There have been previous attempts at performing permutations efficiently on GPUs : Kasagi et al. [11] show how to implement arbitrary permutations (also in an offline setting) in a fully coalesced and bank-conflict free manner, and additionally provide specialized kernels for specific permutations such as transpose or bit-reverse. Their method has similar theoretical guarantees in terms of bandwidth as ours, but they use 5 kernels per permutation whereas we use only one and two kernels for BPC and BMMC permutations respectively. The result is that while our permutations reach roughly 50% (for BMMCs) and 100% (for BPCs) of the speed of a copy, their fastest algorithm is 5 times slower than a copy. Kasagi's method is additionally limited by shared memory size : for an input array of N elements, it requires that $\sqrt{N}$ elements can fit in shared memory, which is typically only a few kilobytes on modern GPUs. Their method can thus only handle input arrays of up to roughly 2^{24} 32-bit elements.

More recently, BMMCs have been used to design GPU address mapping schemes [12]. To the programmer, GPU global

memory is presented as a single contiguous block of memory. The translation between a memory address and actual hardware parameters (involving a bank index, channel index and so on) is handled by a so-called address mapping scheme. Liu et al. represent this mapping using a BMMC mapping : in essence, they implement a fixed BMMC permutation directly in GPU hardware.

9 Conclusions

We have shown an efficient CUDA implementation of BMMC permutations, a class that includes many interesting permutations. The benchmark results are promising, especially for BPC permutations which are basically as fast as they can get, reaching upwards of 90% of the maximum effective bandwidth.

We also explained how inserting BMMC permutations in GPU code at the right places can allow for fully coalesced memory accesses. In some sense, this generalizes an optimization present in the Futhark compiler in which multidimensional arrays are automatically transposed in memory to create opportunities for coalescing when possible ([10] section 5.2 "Optimizing Locality of Reference"). In both cases this does create a tradeoff between the speedup from coalescing and the slowdown from executing additional permutations. Our aim moving forward is to implement `parm` and several related combinators in the Futhark compiler to measure the net gains of this tradeoff. These combinators come with a rich fusion algebra which should permit further optimizations.

Acknowledgement

This research was funded by a Swedish Research Council grant "An algebra of array combinators and its applications", proj. no. 2021-05491. We would also like to thank Troels Henriksen for providing the data for Figure 12.

A Generated CUDA Kernels

This appendix shows the complete CUDA kernels generated for the bit-reverse permutation. For all kernels in this section, the parameters are as follows :

n = 15 n_tile = 5 n_over = 0

The number of scalar instructions in these kernels might be higher than expected : we deliberately do not use CUDA intrinsic functions such as `__brev()` to speed up index computations as this approach would not work for arbitrary bit permutations. We do however perform a simple optimization to reduce the instruction count. When setting bits $i_0 < \cdots < i_k$ in a destination variable using bits $j_0 < \cdots < j_k$ respectively in an input variable, if the offsets $i_1 - i_0, \ldots, i_k - i_{k-1}$ are equal to the offsets $j_1 - j_0, \ldots, j_k - j_{k-1}$, we set all the bits in a single operation (corresponding to a single line in the kernels below). We measured the impact of this optimization and found that on average it reduced by 50% the number of scalar instructions that were generated.

Here is the naive kernel with no tiling :

```
__global__ void bit_reverse_naive(
    const int* input, int* output) {
  size_t in_addr = blockIdx.x * blockDim.x + threadIdx.x;

  // Compute the output address
  size_t out_addr = 0;
  out_addr |= (in_addr & 1ULL) << 14;
  out_addr |= (in_addr & 2ULL) << 12;
  out_addr |= (in_addr & 4ULL) << 10;
  out_addr |= (in_addr & 8ULL) << 8;
  out_addr |= (in_addr & 16ULL) << 6;
  out_addr |= (in_addr & 32ULL) << 4;
  out_addr |= (in_addr & 64ULL) << 2;
  out_addr |= in_addr & 128ULL;
  out_addr |= (in_addr & 256ULL) >> 2;
  out_addr |= (in_addr & 512ULL) >> 4;
  out_addr |= (in_addr & 1024ULL) >> 6;
  out_addr |= (in_addr & 2048ULL) >> 8;
  out_addr |= (in_addr & 4096ULL) >> 10;
  out_addr |= (in_addr & 8192ULL) >> 12;
  out_addr |= (in_addr & 16384ULL) >> 14;
  output[out_addr] = input[in_addr];
}
```

Here is the tiled kernel :

```
__global__ void bit_reverse_tiled(
    const int* input, int* output) {
  __shared__ int tile[1024];
  size_t block = blockIdx.x;
  size_t warp = threadIdx.y;
  size_t thread = threadIdx.x;

  // Read the input tile
  size_t in_addr = 0;
  size_t itile_addr = 0;
  in_addr |= (block & 31ULL) << 5;
  in_addr |= thread & 31ULL;
  in_addr |= (warp & 31ULL) << 10;
  itile_addr |= thread & 31ULL;
  itile_addr |= (warp & 31ULL) << 5;
  tile[itile_addr] = input[in_addr];

  // Synchronize
  __syncthreads();

  // Write the output tile
  size_t out_addr = 0;
  size_t otile_addr = 0;
  out_addr |= (block & 1ULL) << 9;
  out_addr |= (block & 2ULL) << 7;
  out_addr |= (block & 4ULL) << 5;
  out_addr |= (block & 8ULL) << 3;
  out_addr |= (block & 16ULL) << 1;
  out_addr |= (thread & 1ULL) << 4;
  out_addr |= (thread & 2ULL) << 2;
  out_addr |= thread & 4ULL;
  out_addr |= (thread & 8ULL) >> 2;
  out_addr |= (thread & 16ULL) >> 4;
  out_addr |= (warp & 1ULL) << 14;
  out_addr |= (warp & 2ULL) << 12;
  out_addr |= (warp & 4ULL) << 10;
```

```
  out_addr |= (warp & 8ULL) << 8;
  out_addr |= (warp & 16ULL) << 6;
  otile_addr |= (thread & 31ULL) << 5;
  otile_addr |= warp & 31ULL;
  output[out_addr] = tile[otile_addr];
}
```

Here is the tiled kernel, bank-conflict free :

```
__global__ void bit_reverse_banks(
    const int* input, int* output) {
  __shared__ int tile[1024];
  size_t block = blockIdx.x;
  size_t warp = threadIdx.y;
  size_t thread = threadIdx.x;

  // Read the input tile
  size_t in_addr = 0;
  size_t itile_addr = 0;
  size_t ishift = 0;
  in_addr |= (block & 31ULL) << 5;
  in_addr |= thread & 31ULL;
  in_addr |= (warp & 31ULL) << 10;
  itile_addr |= thread & 31ULL;
  itile_addr |= (warp & 31ULL) << 5;
  ishift |= (itile_addr & 992ULL) >> 5;
  tile[(itile_addr & 992) +
    ((ishift + itile_addr) & 31)] =
      input[in_addr];

  // Synchronize
  __syncthreads();

  // Write the output tile
  size_t out_addr = 0;
  size_t otile_addr = 0;
  size_t oshift = 0;
  out_addr |= (block & 1ULL) << 9;
  out_addr |= (block & 2ULL) << 7;
  out_addr |= (block & 4ULL) << 5;
  out_addr |= (block & 8ULL) << 3;
  out_addr |= (block & 16ULL) << 1;
  out_addr |= (thread & 1ULL) << 4;
  out_addr |= (thread & 2ULL) << 2;
  out_addr |= thread & 4ULL;
  out_addr |= (thread & 8ULL) >> 2;
  out_addr |= (thread & 16ULL) >> 4;
  out_addr |= (warp & 1ULL) << 14;
  out_addr |= (warp & 2ULL) << 12;
  out_addr |= (warp & 4ULL) << 10;
  out_addr |= (warp & 8ULL) << 8;
  out_addr |= (warp & 16ULL) << 6;
  otile_addr |= (thread & 31ULL) << 5;
  otile_addr |= warp & 31ULL;
  oshift |= (otile_addr & 992ULL) >> 5;
  output[out_addr] =
    tile[(otile_addr & 992) +
      ((oshift + otile_addr) & 31)];
}
```

Here is the tiled kernel, using iterations (but susceptible to bank conflicts). We choose $n\_iter = 3$ for this example :

```
__global__ void bit_reverse_iters(
    const int* input, int* output) {
  __shared__ int tile[8192];
  size_t block = blockIdx.x;
  size_t warp = threadIdx.y;
  size_t thread = threadIdx.x;

  // Read the input tiles
  size_t in_addr = 0;
  size_t itile_addr = 0;
  itile_addr |= thread & 31ULL;
  itile_addr |= (warp & 31ULL) << 5;
  in_addr |= (block & 3ULL) << 8;
  in_addr |= thread & 31ULL;
  in_addr |= (warp & 31ULL) << 10;
  for (size_t iter = 0; iter < 8; iter++) {
      in_addr &= ~224ULL;
      in_addr |= (iter & 7ULL) << 5;
      tile[(iter << 10) + itile_addr] =
        input[in_addr];
  }

  // Synchronize
  __syncthreads();

  // Write the output tiles
  size_t out_addr = 0;
  size_t otile_addr = 0;
  otile_addr |= (thread & 31ULL) << 5;
  otile_addr |= warp & 31ULL;
  out_addr |= (block & 1ULL) << 6;
  out_addr |= (block & 2ULL) << 4;
  out_addr |= (thread & 1ULL) << 4;
  out_addr |= (thread & 2ULL) << 2;
  out_addr |= thread & 4ULL;
  out_addr |= (thread & 8ULL) >> 2;
  out_addr |= (thread & 16ULL) >> 4;
  out_addr |= (warp & 1ULL) << 14;
  out_addr |= (warp & 2ULL) << 12;
  out_addr |= (warp & 4ULL) << 10;
  out_addr |= (warp & 8ULL) << 8;
  out_addr |= (warp & 16ULL) << 6;
  for (size_t iter = 0; iter < 8; iter++) {
      out_addr &= ~896ULL;
      out_addr |= (iter & 1ULL) << 9;
      out_addr |= (iter & 2ULL) << 7;
      out_addr |= (iter & 4ULL) << 5;
      output[out_addr] =
        tile[(iter << 10) + otile_addr];
  }
}
```

References

[1] KE Batcher. 1968. Sorting networks and their applications. *AFIPS Spring Joint Computer Conference* 32 (1968), 307–314.

[2] Koen Claessen, Mary Sheeran, and Satnam Singh. 2001. The Design and Verification of a Sorter Core. In *Correct Hardware Design and Verification Methods*, Tiziana Margaria and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 355–368.

[3] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming* (Philadelphia, Pennsylvania, USA) *(DAMP '12)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/2103736.2103740

[4] Thomas H. Cormen. 1993. Fast permuting on disk arrays. *J. Parallel and Distrib. Comput.* 17, 1-2 (1993), 41–57.

[5] Thomas H. Cormen and Leonard F. Wisniewski. 1993. Asymptotically Tight Bounds for Performing BMMC Permutations on Parallel Disk Systems. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures* (Velen, Germany) *(SPAA '93)*. Association for Computing Machinery, New York, NY, USA, 130–139. https://doi.org/10.1145/165231.165248

[6] M. Dowd, Y. Perl, M. Saks, and L. Rudolph. 1983. The Balanced Sorting Network. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/800221.806719

[7] Alan Edelman, Steve Heller, and S. Lennart Johnsson. 1994. Index Transformation Algorithms in a Linear Algebra Framework. *IEEE Trans. Parallel Distrib. Syst.* 5, 12 (1994), 1302–1309.

[8] Mark Harris. 2013. An Efficient Matrix Transpose in CUDA C/C++. https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/

[9] Troels Henriksen. 2017. *Design and implementation of the Futhark programming language*. Ph. D. Dissertation. University of Copenhagen, Faculty of Science [Department of Computer Science].

[10] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[11] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. 2014. Offline Permutation on the CUDA-enabled GPU. *IEICE TRANSACTIONS on Information and Systems* 97, 12 (2014), 3052–3062.

[12] Yuxi Liu, Xia Zhao, Magnus Jahre, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Lieven Eeckhout. 2018. Get out of the Valley: Power-Efficient Address Mapping for GPUs. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Los Angeles, California, 166–179. https://doi.org/10.1109/ISCA.2018.00024

[13] NVIDIA. 2023. CUDA Runtime API :: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[14] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. 1994. Algorithms for parallel memory, I: Two-level memories. *Algorithmica* 12 (1994), 110–147.

Received 2023-05-31; accepted 2023-06-28