Configurable XOR hash functions for banked scratchpad memories in GPUs

Gert-Jan van den Braak, Juan Gómez-Luna, José María González-Linares, Henk Corporaal, Nicolás Guil

Abstract—Scratchpad memories in GPU architectures are employed as software-controlled caches to increase the effective GPU memory bandwidth. Through the use of well-known optimization techniques, such as privatization and tiling, they are properly exploited. Typically, they are banked memories which are addressed with a mod(2^N) bank indexing scheme. Although their bandwidth is fully exploited for linear memory accesses, their performance is burdened when non-unit strides appear in memory access patterns because they provoke bank conflicts. This paper explores the use of configurable *bit-vector* and *bitwise* XOR-based hash functions to evenly distribute memory addresses of the access patterns over the memory banks, reducing the number of bank conflicts. An exhaustive, but lightweight, search is used to configure bit-vector hash functions. Bitwise hash functions are configured with heuristics. Hardware and software implementations are carried out. For the hardware approach, the experimental results show 24% performance speed-up for 22 benchmarks on GPGPU-Sim, a Fermi-like simulator. Bank conflicts are reduced by 96% with bit-vector hash functions, and 97% with bitwise hash functions using our proposed Minimum Imbalance Heuristic. The software approach, using bit-vector hash functions, demonstrates 23% speed-up and 96% bank conflict reduction on a Fermi GPU, and 33% speed-up and 99% bank conflict reduction on a Kepler GPU.

Index Terms—Computer architecture, Graphics processing units, Memory architecture

1 Introduction

In the last decade, GPU computing has burst into the field of High Performance Computing as a booming trend, thanks to the outstanding horsepower of GPUs and the relative easiness of programming with CUDA and OpenCL. Many programmers have adopted GPUs to obtain impressive speed-ups on their codes.

One key factor for GPU programmers to achieve such improvements is taking advantage of the on-chip scratchpad memories (i.e., shared memory in CUDA, local memory in OpenCL), which are used as software-controlled caches. Optimization techniques such as tiling and privatization [1] are widely-used to leverage the on-chip memory, in order to improve data locality and reduce thread contention, respectively.

Unfortunately, a proper use of these on-chip memories requires programmers to have a certain level of expertise. As these are interleaved memories with a power-of-two number of banks, their bandwidth is only properly exploited when all the banks are read or written concurrently. Thus, avoiding bank conflicts is fundamental to not hamper the performance. An m-way bank conflict degree means that m concurrent threads access the same bank. Consequently, these m accesses are serialized by the hardware.

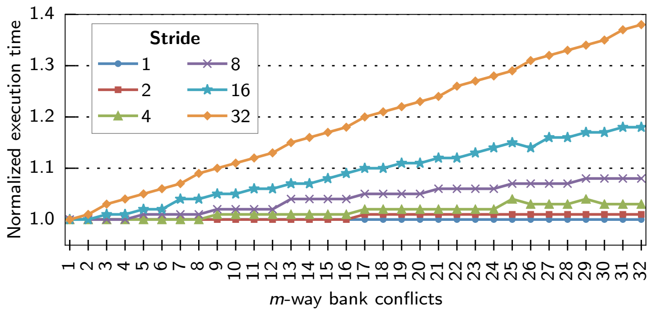
Hash functions are used in processors for memory address mapping to increase the bandwidth of multibank memories and caches [2], [3]. The main aim of these functions is to spread evenly the memory accesses of a running application among the memory banks, reducing, in this way, the bank conflict degree. Choosing the most suitable hash function for a specific application should take into account how well the memory references are spread and the impact of the hash function in the final memory latency. The selection of bank indexing bits can be performed by exhaustive search or with heuristics [3].

In this work, we propose the use of configurable hash functions to access the on-chip scratchpad memories. The aim is to reduce the number of bank conflicts without explicit actions from the programmer's perspective. The main contributions of this paper are the following:

- A formulation that describes scratchpad memory access patterns is presented, as well as a classification that establishes the most common access patterns.
- Hash functions based on the permutation of memory bank addressing bits are introduced. Bit-vector and bitwise XOR operators are added to cope with complex and combined access patterns.
- A new heuristic, the Minimum Imbalance Heuristic (MIH), has been developed in order to discover the best configuration of the bitwise hash functions. It outperforms the Givargis Heuristic [3].
- A study of the hardware cost of the proposed hash functions is carried out.
- A framework that applies the proposed hash functions to increase the performance of GPU kernel execution is proposed. This framework extracts the access patterns present in the kernel, calculates the hash function configuration that reduces the bank conflicts, and reconfigures the addressing hardware.

- *Gert-Jan van den Braak and Henk Corporaal are with the Eindhoven University of Technology, the Netherlands.*
  E-mail: {g.j.w.v.d.braak, h.corporaal}@tue.nl.
- *Juan Gómez-Luna is with the University of Córdoba, Spain.*
  E-mail: el1goluj@uco.es.
- *José María González-Linares and Nicolás Guil are with the University of Málaga, Spain.*
  E-mail: {jgl, nguil}@uma.es.

Manuscript received April 9; revised July 15; accepted August 2, 2015.



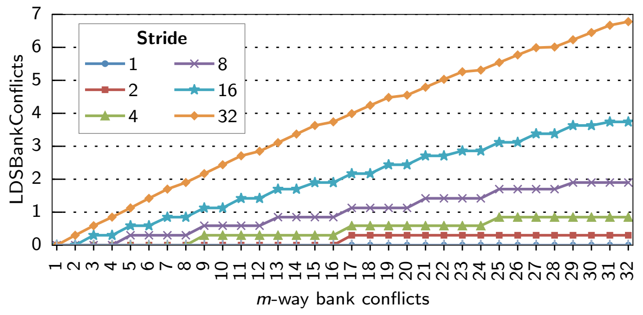


Fig. 1. Execution results on AMD Hawaii for access patterns to scratchpad memory with different bank conflict degrees (m-way) and strides. (Left) Normalized execution time; (Right) Profiling results.

The use of software hash functions that avoids hardware modifications is also discussed.

- Exhaustive experimentation is accomplished using 22 kernels from the NVIDIA CUDA SDK, Rodinia and Parboil benchmark suites. The results show that our approach is able to eliminate 97% of the bank conflicts achieving a geometric mean 24% speed-up.

The rest of the paper is organized as follows. First, a motivational experiment in Section 2 shows that scratchpad memory bank conflicts can harm performance significantly, and that removing these conflicts is essential to achieving peak GPU performance. In Section 3 a memory access pattern classification for scratchpad memory accesses based on [4], [5] is shown. Four different classes of configurable hash functions are introduced in Section 4, including an evaluation of their hardware costs. Heuristics are used to configure these hash functions, as shown in Section 5. In this section the new Minimum Imbalance Heuristic is presented. Section 6 illustrates a framework that applies hash functions to kernels running on GPU architectures. This framework proposes the implementation of the calculated hash function not only in hardware but also in software. The results of the different hash functions and heuristics are presented in Section 7. Related work is discussed in Section 8. Finally, conclusions and future work are stated in Section 9.

2 Motivation

Through an illustrative experiment, we show in this section the impact of bank conflicts on two modern GPUs, AMD Hawaii and NVIDIA K20. We have used a simple microbenchmark to evaluate this impact:

```
int index = GenIndex(tid, way, stride);
for(int i = 0; i < repeat; i++)
      index = ScratchpadMemory[index];
```

where the function `GenIndex` returns an index value calculated as follows:

$$GenIndex(tid, way, stride) = \begin{cases} tid * stride & \text{if } tid < way \\ tid & \text{elsewhere} \end{cases}$$

where tid is the threadID and way is the number of consecutive threads employing a strided access of value $stride$.

For instance, if way = 4 and $stride$ = 32, 32 consecutive threads (a warp in NVIDIA devices, or a half-wavefront in AMD devices) will access the following addresses: [0 32 64 96 4 5 6 . . . 31].

By changing way and $stride$, we can analyze the impact of bank conflicts on performance. Fig. 1 (left) shows the normalized execution time on AMD Hawaii. This GPU contains 64 kB of scratchpad memory, called LDS, per compute unit. The LDS has 32 banks, and each bank is 4 bytes wide. Thus, power-of-two strides provoke bank conflicts. Fig. 1 (right) presents the corresponding results of the performance counter `LDSBankConflicts` as given by CodeXL profiler [6].

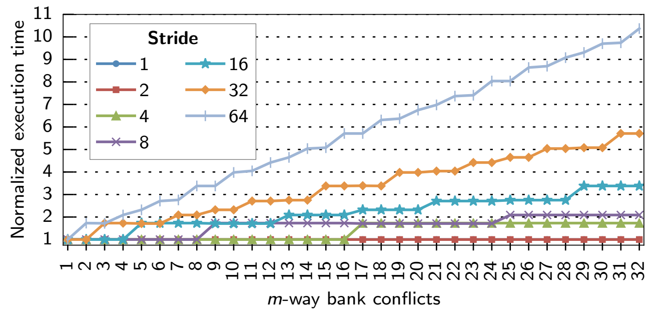
Fig. 2 shows the same experiments on NVIDIA K20. This GPU has up to 48 kB of shared memory per streaming multiprocessor, where there are 32 banks and each bank is 8 bytes wide. This helps to decrease the number of bank conflicts when accessing 4-byte data elements (e.g., a stride of 2 does not provoke bank conflicts). On the right, the results of the performance counter `shared_load_replay`, given by CUDA profiler [7], are shown.

In real world benchmarks the performance penalty caused by bank conflicts not only depends on the conflict degree, but also on the relative number of scratchpad memory instructions in the application. For example, in the first MRI benchmark from Section 7 (MRI-grid-1) only 7.6% of the instructions are scratchpad memory instructions (194 loads and 141 stores). The average conflict degree is 16.3, which results in a large potential performance gain when the bank conflicts are removed. Anther benchmark, convolution (conv-2), has a high number of scratchpad memory instructions of 35% (2560 loads and 640 stores), but the average bank conflict degree is only 2.2. Still, performance can be improved significantly when these conflicts are avoided, as shown in the results of Section 7.

As shown above, bank conflicts have a dramatic impact on performance for both the AMD and the NVIDIA architectures. This encourages us to propose hardware and software improvements that free programmers from spending their time and effort in data rearrangements or fancy addressing schemes that reduce the number of bank conflicts.

3 Access patterns to scratchpad memory

In this section, we describe typical memory access patterns to scratchpad memory that can be found in real-world applications. We focus on the basic access pattern that generates (at least) one memory transaction, that is, a collection of addresses whose size is equal to the number of memory banks. In current architectures, this size is equal to the number of threads in a warp (NVIDIA) or in a half-wavefront (AMD).



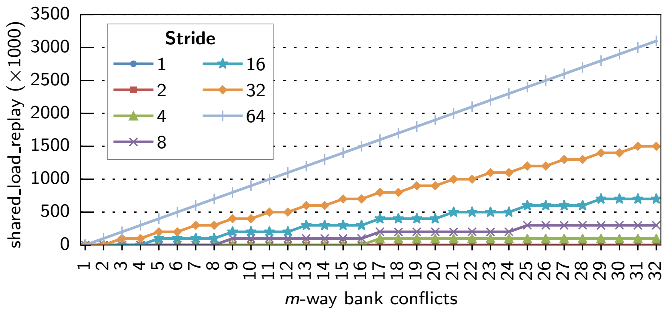


Fig. 2. Execution results on NVIDIA K20 for access patterns to scratchpad memory with different bank conflict degrees (m-way) and strides. (Left) Normalized execution time; (Right) Profiling results.

We will be specially careful with non-unit stride memory accesses, as they are a typical source of bank conflicts. 1D strided accesses are defined in [8] as:

$$shared\_memory[stride \cdot t_x + offset] \tag{1}$$

where *stride* is the distance between threads with consecutive threadID t_x. According to [8], no bank conflicts will occur if *stride* is relative prime to the number of banks.

In [4], a memory access vector $\vec{s}$ is expressed as a combination of a memory access matrix M, an iteration vector $\vec{i}$, and an offset vector $\vec{o}$:

$$\vec{s} = M\vec{i} + \vec{o} \tag{2}$$

The authors apply this notation to loop nests of arbitrary depth. This notation is adapted in [5] to separate inter-thread ($\overrightarrow{eMAP}$) and intra-thread ($\overrightarrow{iMAP}$) components as follows:

$$\begin{aligned}\vec{s} &= \overrightarrow{eMAP} + \overrightarrow{iMAP} = M \cdot \overrightarrow{tid} + \overrightarrow{iMAP} \\ &= \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} iMAP_0 \\ iMAP_1 \end{bmatrix}\end{aligned} \tag{3}$$

As it can be seen, M is a 2×2 matrix, and $\overrightarrow{tid}$ and $\overrightarrow{iMAP}$ are vectors. $\overrightarrow{tid}$ identifies threads in a 2D block (or a workgroup in OpenCL terminology).

In [5] it is assumed that M_{00}, M_{01}, M_{10}, $M_{11} \in \{0, 1\}$. Thus, they only consider 16 cases of $\overrightarrow{eMAP}$, where there are no non-unit strides. In [4] non-unit stride accesses are defined with a matrix M where M_{11} is a constant $C \notin \{0, 1\}$.

In this work, we define our own adaptation of the above notations. We linearize the notation, since we need to detect the collection of addresses that are accessed by a warp (or half-wavefront). For instance, if thread blocks are of size 16×16, threads with $t_y = 0$ and $t_y = 1$ are mapped to the same warp. This is not evident if we use a 2D notation.

Let us assume that a 2D shared memory space of size $ROWS \times COLS$ is accessed. Our linearized notation can be derived from Equation 3 as follows:

$$\begin{aligned}\vec{s} &= M \cdot \overrightarrow{tid} + \vec{o} = \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \\ s &= (M_{00}t_y + M_{01}t_x + o_0)\,COLS + M_{10}t_y + M_{11}t_x + o_1\end{aligned} \tag{4}$$

where s is now the memory position accessed by the thread with $\overrightarrow{tid} = (t_x, t_y)$

Comparing to Equation 1, we identify a stride $M_{01}\,COLS + M_{11}$. Moreover, if the size of the thread block in the x dimension (`blockDim.x` in CUDA) is smaller than the warp size, we should also consider the stride $M_{00}\,COLS + M_{10}$.

3.1 Memory access pattern classification

Using the notation given above, a classification of the access patterns can be carried out. Thus four classes of memory access patterns can be distinguished: *linear*, *stride*, *block* and *random*. A description of these classes is given below.

Linear is the most simple class where all memory accesses in a warp are consecutive.

$$\vec{s} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{5}$$

Stride is similar to *linear*, only the accesses are separated with a stride factor S. Note that *linear* is a special case of *stride* where $S = 1$.

$$\vec{s} = \begin{bmatrix} 0 & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{6}$$

Block is a class including 2D access pattern where the threads in a warp have different values for t_x and t_y. In some combinations of S_1, S_2, S_3 and S_4 multiple accesses map to the same address (e.g., $S_1 = S_2 = S_3 = S_4 = 0$).

$$\vec{s} = \begin{bmatrix} S_1 & S_2 \\ S_3 & S_4 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{7}$$

Random is the last class and contains all cases which cannot be captured by the other classifications, similar to [4]. In the access pattern below Z_x and Z_y are random numbers.

$$\vec{s} = \begin{bmatrix} Z_y & 0 \\ 0 & Z_x \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} \tag{8}$$

3.2 Examples of access pattern classifications

The memory access classification will help us to understand the access patterns that can be found in real-world benchmarks. Once we know the strides involved, we will be able to propose bit-vector hash functions as explained in Section 5.1. To illustrate how different memory access patterns can be expressed, let us consider three widely-known applications included in the CUDA SDK, matrix transpose, reduction and Fast Walsh Transform. Matrix transpose is an

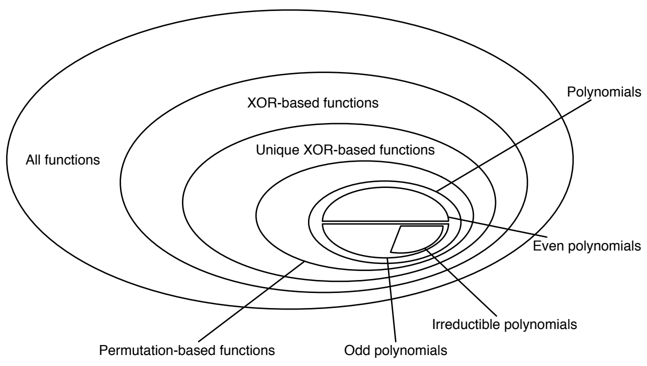


Fig. 3. Classification of different types of hash functions as given by Vandierendonck et al. [2]

out-of-place matrix transposition, and essentially consists of loading data from global memory into shared memory, and then storing the transposed elements from shared memory to global memory.

3.2.1 Matrix transpose - loading

In the loading stage data is written into the scratchpad memory. `tile` is a 2D shared memory space of size $\mathit{TILE\_DIM} \times \mathit{TILE\_DIM}$. It is written by one thread block of the same size. `idata` is the input matrix in global memory, and i is the index of the loop that goes through the matrix.

```
tile[threadIdx.y+i][threadIdx.x] =
 idata[index_in+i*width];
```

We linearize the access:

$$\vec{s} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} i \\ 0 \end{bmatrix}$$

resulting in:

$$\begin{aligned} s &= (t_y + i) \cdot \mathit{TILE\_DIM} + t_x \\ &= t_y \cdot \mathit{TILE\_DIM} + t_x + i \cdot \mathit{TILE\_DIM} \end{aligned}$$

We observe that threads with equal t_y and consecutive t_x perform a linear access (unit stride). However, if $\texttt{blockDim.x} < \mathit{warp\_size}$, threads of consecutive t_y and equal t_x will have a stride $\mathit{TILE\_DIM}$ between them. If $\mathit{TILE\_DIM} = 16$ and $\mathit{warp\_size} = 32$, the memory access pattern for warp 0 and $i = 0$ is:

0, 1, 2, 3, . . . , 15, 16, 17, 18, 19, . . . , 31. No bank conflicts (with 32 banks).

3.2.2 Matrix transpose - storing

In the storing stage data is read from scratchpad memory. `odata` is the output matrix in global memory.

```
odata[index_out+i*height] =
 tile[threadIdx.x][threadIdx.y+i];
```

We linearize the access:

$$\vec{s} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} 0 \\ i \end{bmatrix}$$

resulting in:

$$s = t_x \cdot \mathit{TILE\_DIM} + t_y + i$$

In this case, the source of conflict is the stride $\mathit{TILE\_DIM}$ between threads of consecutive t_x and equal t_y. The memory access pattern for warp 0 and $i = 0$ is:

0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 1, 17, 33, 49, 65, 81, 97, 113, 129, 145, 161, 177, 193, 209, 225, 241. 8-way bank conflict (with 32 banks).

3.2.3 Reduction

If an application does not use 2D memory spaces, the linearization is trivial. For instance, in the reduction kernel (CUDA SDK):

```
sdata[2*S*tx] += sdata[2*S*tx + s];
```

The access on the left is (`S` is a power-of-two, $1 \leq \texttt{S} < \texttt{blockDim.x}$):

$$\vec{s} = \begin{bmatrix} 0 & 0 \\ 0 & 2 \cdot \texttt{S} \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and consequently:

$$s = 2 \cdot \texttt{S} \cdot t_x$$

The stride, i.e. the distance between consecutive threads, is $2 \cdot \texttt{S}$. As `S` is a power of two, bank conflicts appear (with 32 banks).

3.2.4 Fast Walsh Transform

One interesting case that uses 1D blocks is the Fast Walsh Transform (CUDA SDK). In this kernel, memory access patterns in shared memory are generated by the following code:

```
int lo = pos & (stride-1); // or: pos % stride;
int i0 = ((pos - lo) << 2) + lo;
float D0 = s_data[i0];
```

In this kernel, the variable called `stride` takes values of 512, 128, 32, 8 and 2 for the default data used by the code ($\texttt{pos} = t_x$). When the variable `stride` takes values from 512 to 32 no conflicts appear (with 32 banks) and a regular access pattern with stride 1 is generated. However, non regular access patterns are generated for `stride` values of 8 and 2. For instance, the addresses generated in a warp (in this example warp 0 from block 0,0) for `stride` = 8 are:

0, 1, 2 ,3, 4, 5, 6, 7, 32, 33, 34, 35, 36, 37, 38, 39, 64, 65, 66, 67, 68, 69, 70, 71, 96,97, 98, 99, 100, 101, 102, 103 (4-way bank conflict)

In order to adapt this addressing to our notation, we notice that the first of the above instructions can be seen as the calculation of the thread index in a set of threads of size `stride`. Thus, a warp would be divided into several sub-warps of size `stride`. Let us re-write the instructions:

```
lo = pos - Integer_part_of(pos/stride) * stride;
i0 = Integer_part_of(pos/stride) * stride*4 +lo;
```

Notice that `Integer_part_of(pos / stride)` is the index of a sub-warp of size `stride` (we call it `sw_index`): `i0 = sw_index * stride * 4 + lo;`.

A warp divided into sub-warps can be seen as a 2D collection of threads with $\texttt{sw\_index} = t_y$ and $\texttt{lo} = t_x$:

$$\vec{s} = \begin{bmatrix} 4 \cdot \texttt{stride} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$s = 4 \cdot \texttt{stride} \cdot t_y + t_x$$

TABLE 1
Number of hash functions per class, and an example for mapping $n = 14$ address bits to $m = 5$ bank bits.

Class	Size	Example
All functions	$(2^m)^{(2^n)}$	$2.3e24660$
XOR-based functions	2^{nm}	$1.2e21$
Unique XOR-based functions	$N(n,m)$ — Eq. 9	$1.2e14$
Bit-vector functions	$n - m + 1$	10
Bit-vector XOR functions	$(n - m + 1) \cdot n \cdot 2^m$	4480
Bitwise permutation functions	$\binom{n}{m}$	2002
Bitwise XOR functions	$\binom{n(n+1)/2}{m}$	$9.7e7$

Thus, the distance between threads of equal t_y and consecutive t_x is 1, and the distance between threads of equal t_x and consecutive t_y is $4 \cdot$ `stride`.

4 Hash functions

The goal of a hash function is to distribute the memory accesses over the memory banks as evenly as possible. The memory consists of 2^n words, divided over 2^m banks. The simplest hash function selects the least significant bits to select the memory banks. Although this works well for many access patterns, it can cause bank conflicts for other access patterns. In those cases other bits should be chosen.

Hash functions can be divided into several classes, as illustrated in Fig. 3 by Vandierendonck et al. [2]. In total there are $(2^m)^{(2^n)}$ functions to map n to m bits [2]. Many of these functions are not interesting as they do not use all available banks or have high computational requirements.

XOR-based functions are often used as hash functions because of their relative good hashing properties, especially for strided memory accesses, and their low computational cost. According to [2], there are 2^{nm} XOR-based hash functions and $N(n,m)$ unique XOR hash functions, where $N(n,m)$ is calculated using the following expression:

$$N(n,m) = \prod_{i=1}^{m} \frac{2^{n-i+1} - 1}{2^{m-i+1} - 1} \tag{9}$$

The first three entries of Table 1 show the number of hash functions contained in the previous classes and calculate these number for a specific case: NVIDIA's Fermi architecture with 48 kB shared memory divided over 32 banks

Testing all (unique) XOR hash functions for any given access pattern is unfeasible due to the large number of possible functions. Often a suitable hash function can be determined based on the access pattern classification. In the following sections, two approaches to select m out of n bits performing the bank addressing are presented. The first one, called bit-vector approach, looks for m consecutive bits and drastically reduces the search space. The second one, named bit-wise approach, is more flexible and selects m individual bits out of n arbitrary positions, generating a much larger search space.

In addition, both approaches are also combined with an XOR operator yielding to the four types of hash functions described below. The hardware costs of the hash functions is evaluated in in terms of chip-area, power consumption and increased memory access latency in Section 4.5.

TABLE 2
A description of the hash functions used in this work along with the technique employed for searching in the configuration space.

Bits selection	Hash functions	Search method
bit-vector	bit-vector perm.	Exhaustive
	bit-vector XOR	Exhaustive
bitwise	bitwise perm.	Heuristic: Givargis or MIH
	bitwise XOR	Heuristic: Givargis or MIH

All four types of hash functions are configurable. The best configuration can be found using either heuristics or an exhaustive search algorithm, as will be described in Section 5. An overview of the hash functions and the configuration method is shown in Table 2.

4.1 Bit-vector permutation hash function

A common access pattern for a GPU's scratchpad memory happens when the accesses of a warp belong to the classes *linear* or *stride*. Then, the stride S can be written as $S = S_0 \cdot 2^k$. The number k indicates which m bits to select out of the n address bits. In case of the *linear* access pattern and $k = 0$, the selection of the least significant bits $[0 \dots m)$ as the hash function is the best possible choice. In case $k > 0$ the access pattern is classified as *stride*. The best possible hash function for a pure strided memory access uses the address bits $[k \dots k + m)$. As it can be easily calculated, the total number of possible bit-vector hash functions is only $n - m + 1$ (see Table 1).

4.2 Bit-vector XOR hash function

The bit-vector XOR hash functions extend the bit-vector permutation hash functions by combining two vectors that consist of m consecutive bits from the word address with an offset of k_1 and k_2, respectively. Moreover, the second vector has a mask such that a selection of bits in this vector can be made. The bank index is calculated from the word-address using the formula $bank = addr[k_1 \dots k_1 + m] \oplus (addr[k_2 \dots k_2 + m] \& mask)$, which is implemented as:

```
bank = (addr>>k1) ^ ((addr>>k2) & mask);
```

This hash function is particularly useful when multiple strided memory accesses with different strides S occur in one application, or, more precisely, have different values of k in $S = S_0 \cdot 2^k$. It can be also very appropriate when a *block* access pattern is used. For instance, let us consider the memory access pattern from the Fast Walsh transform in the CUDA SDK shown below (see also Section 3).

$$s = \begin{bmatrix} 32 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_y \\ t_x \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 32 \cdot t_y + t_x$$
$$\text{with } 0 \leq t_x < 8 \text{ and } 0 \leq t_y < 4 \tag{10}$$

Thread index t_x uses bits 0-2 of the address, and thread index t_y uses bits 5-6. These bits cannot be captured in a single bit-vector, but the bit-vector XOR hash can combine two vectors to create a better distribution of memory accesses over the memory banks.

Because bit-vectors are combined, instead of individual bits, the total number of possible hash functions is limited

TABLE 3
All $n(n+1)/2$ possible XOR combinations of 4-element vector $(a\ b\ c\ d)$.

	a	**b**	**c**	**d**
a	a	$a \oplus b$	$a \oplus c$	$a \oplus d$
b		b	$b \oplus c$	$b \oplus d$
c			c	$c \oplus d$
d				d

to $(n - m + 1) \cdot n \cdot 2^m$, or 4480 in case of a Fermi GPU's scratchpad memory (see Table 1). Although finding the best possible hash function requires testing all bit-vector XOR hash functions, often only a limited set needs to be evaluated, as will be described in Section 5.1.

4.3 Bitwise permutation hash function

In cases where selecting one bit-vector or combining two bit-vectors is not flexible enough to create a good hash function, it is also possible to select m bits individually. The number of possible choices of m bank addressing bits out of n address bits is $\frac{n!}{(n-m)!}$. The order of the selected bits only influences in which bank the conflicts will occur, not the amount the conflicts. Therefore the actual number of choices is given by the binomial coefficient $\binom{n}{m}$ or $\frac{n!}{m!(n-m)!}$ (see Table 1).

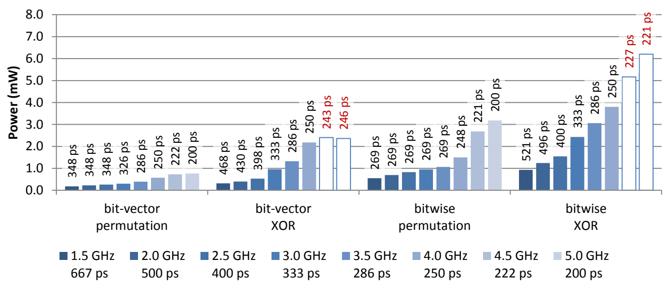
4.4 Bitwise XOR hash function

Instead of choosing individual bits as a hash function, it is also possible to select pairs of bits which will be combined using the XOR operation. In the bitwise permutation hash function m bits were selected out of the n address bits. In the bitwise XOR hash function the n address bits are combined into n^2 combinations, and m pairs of bits are selected. Because the XOR operation is commutative, not all combinations have to be evaluated [9]. Instead of creating n^2 options, only $n(n+1)/2$ combinations have to be evaluated, as shown in the example of Table 3. The total number of possible bitwise XOR hash functions is about 97 million for a Fermi GPU's scratchpad memory (see Table 1).

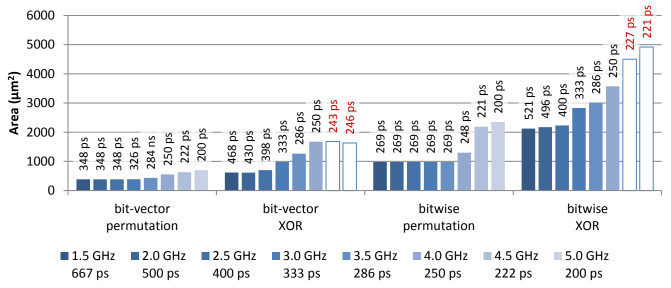
4.5 Hardware design and evaluation

While a fixed hash function has a negligible latency, the proposed configurable hash functions' latency cannot be ignored. To estimate these latencies the proposed hash functions are implemented in Verilog. Latency, area and power numbers are obtained using the Cadence Encounter® RTL Compiler v11.20 and a 40 nm standard cell library. A range of target clock frequencies is tested to find the best trade-off between area, power and latency for each hash function, as shown in Fig. 4. In case the latency obtained is low compared to the GPU's clock period (~700 ps), the configurable hash function can be integrated in an existing clock cycle of the memory access; otherwise each memory access has to be extended with one more clock cycle to facilitate the hash function.

All four configurable hash functions are evaluated: bit-vector permutation, bit-vector XOR, bitwise permutation and bitwise XOR. The power and area costs for a single instantiation for these four hash functions are shown in Fig. 4a and Fig. 4b respectively. These figures show the



(a) power vs. latency



(b) area vs. latency

Fig. 4. Power (a) and area (b) vs. latency results for the four proposed hash functions for a range of target clock frequencies (1.5 GHz – 5.0 GHz)

range of tested clock frequencies (1.5 GHz – 5 GHz) and the target clock period just below the target clock frequency. Furthermore, each bar displays and the achieved latency in each experiment. It can be noticed that the area and power costs increase for each of the four hash functions as the target clock frequency increases. For the bit-vector XOR and the bitwise XOR hash functions the target clock frequencies of 4.5 GHz and 5.0 GHz are not feasible. The minimum latency required for each of the hash functions is 250 ps. This is a significant part of the ~700 ps of a GPU's clock cycle. Therefore we increase the memory access latency by one cycle in the experiments of Section 7 in case a configurable hash function is used.

The hash function hardware has to be instantiated for every bank in the scratchpad memory. An NVIDIA Fermi GPU has a scratchpad memory consisting of 32 banks in each of its 16 streaming multiprocessors. In total the power consumption of the scratchpad memory ranges from 0.1 W for the bit-vector permutation hash function to 0.5 W for the bitwise XOR hash function. This is about 0.2% of the total power consumption of an NVIDIA GTX 580. The corresponding area costs range from 0.2 mm^2 to 1.1 mm^2, as shown in Table 4.

5 HASH FUNCTION CONFIGURATION

As each kernel can employ different patterns to access the scratchpad memory, the hash functions described in Section 4 must be configured per kernel.[1] The configuration parameters for the bit-vector permutation hash functions are determined using an exhaustive search algorithm described in Section 5.1, since the number of options is limited (see Table 1). The options for the parameters of

1. When multiple kernels are executing concurrently, different hash functions can be used for different streaming multiprocessors.

TABLE 4
Power and area costs of the four proposed hash functions compared to an NVIDIA GTX 580 GPU.

Hash function	Power		Area	
bit-vector permutation	0.1 W	(0.04%)	0.2 mm^2	(0.04%)
bit-vector XOR	0.2 W	(0.07%)	0.3 mm^2	(0.06%)
bitwise permutation	0.3 W	(0.1%)	0.5 mm^2	(0.1%)
bitwise XOR	0.5 W	(0.2%)	1.1 mm^2	(0.2%)

the bitwise permutation hash functions are much larger, therefore heuristics are used. Two different heuristics are evaluated: the Givargis heuristic [3] (GH) and the proposed Minimum Imbalance Heuristic (MIH). The extended hash function types with XOR operator are configured same as the corresponding permutation-based types. Although the number of options might also be large for the bit-vector XOR-based hash functions, we show at the end of Section 5.1 that this number can be drastically reduced under certain circumstances, making the exhaustive search much more affordable.

5.1 Bit-vector exhaustive search algorithm

The bit-vector permutation hash function requires only one parameter: k. The number of options for k is very limited (e.g. only 10 options are possible in the example of Table 1). The bit-vector XOR hash function requires three parameters: k_1, k_2 and $mask$. A bit-vector permutation hash function can be emulated by selecting $k_1 = k$ and $mask = 0$. Since every possible bit-vector permutation hash function can easily be tested, and can also be emulated by a bit-vector XOR hash function, we only focus on the latter one.

An example of the bit-vector XOR hash function is shown in Fig. 5, where $k_1 = 2$, $k_2 = 8$ and $mask = 7$, which results in a hash function which selects the following bank bits: $b_0 = a_2 \oplus a_8$, $b_1 = a_3 \oplus a_9$, $b_2 = a_4 \oplus a_{10}$, $b_3 = a_5$ and $b_4 = a_6$.

To select the values for k_1, k_2 and $mask$ every possible combination of k_1, k_2 and $mask$ should be explored. In the example of a Fermi GPU (Table 1) 48 kB of scratchpad memory is divided over 32 banks. Hence 14 bits are required to index every word, and 5 bits for every bank. As a result k_1 ranges from 0 to 9 to make sure always 5 bits are in the result, k_2 ranges from 0 to 13 because it can consist of only one bit due to the mask, and $mask$ ranges from 0 to 31 because there are 5 bits used to index the 32 banks. In total there are $10 \times 14 \times 32 = 4480$ combinations to test.

The number of combinations to evaluate can be reduced by limiting the possible values for k_1, k_2 and $mask$. As every stride S can be written as $S = S_0 \cdot 2^k$, the options for k_1 can be limited to the values of k of all strided memory access patterns encountered. Similarly the most significant bit (MSB) for every strided memory access pattern is calculated as $MSB = \lfloor \log_2 ((t-1) \cdot S) \rfloor$, with t the number of threads in a warp. By taking the minimum value for all k, and the maximum value for all MSB, the range of k_2 can be limited. Furthermore, values $k_2 = k_1$ will not give good hashing functions and do not have to be tested, since $a_x \oplus a_y = 0$ if $x = y$. For instance, if two different strides $S = 4$ and $S = 6$ appear in the scratchpad memory accesses of one kernel, k_1 can be 2 or 1 respectively. The maximum MSB is calculated as 7, so that $k_2 \in [1, 7]$. The options $k_1 = k_2 = 1$ and $k_1 = k_2 = 2$ can be discarded, as they will not give good hashing functions. Taking the various options for $mask$ into account, the total number of combinations to evaluate is 188, only 4% of the 4480 possible combinations. In case k is equal for all strides, the algorithm will select $k_1 = k$ and $mask = 0$ as a simple permutation will already give the best results.

5.2 Bitwise search algorithm based on heuristic

As it was previously indicated, the search space size to find the optimum bitwise hash function configuration can be very high. Thus, brute-force based methods can take a long time. To overcome this problem, two heuristics have been employed. They are presented in this section.

5.2.1 Givargis Heuristic

Givargis introduces in [3] a heuristic to select the best m out of n address bits to index a cache. The goal is to use the available cache as fully as possible over the duration of a program. Therefore all memory accesses of an application are put in one set, and the heuristic has to find the address bits to index the cache such that there are as few as possible collisions in the cache.

In the case of scratchpad memory accesses, we need to find the best address bits to eliminate bank conflicts within one access made by one warp. This makes it possible to apply the heuristic on every warp access pattern separately. First, the GH is briefly described below (for a full description see Section 2.3 in [3]). Then, an extension is presented to combine the results of all the warp access patterns to select the overall best bank addressing bits.

Given a set R of memory references. Such a set could for example be the addresses accessed by a single warp in a single instruction. For each bit A_i in the address space a corresponding quality measure Q_i is calculated. The quality measure is a real number ranging from 0 to 1 and is calculated by taking the ratio of zeros and ones of bit A_i in all memory addresses in the set R as in the following equation:

$$Q_i = \frac{\min(Z_i, O_i)}{\max(Z_i, O_i)} \tag{11}$$

where Z_i and O_i are the number of references having 0 and 1 at bit A_i, respectively.

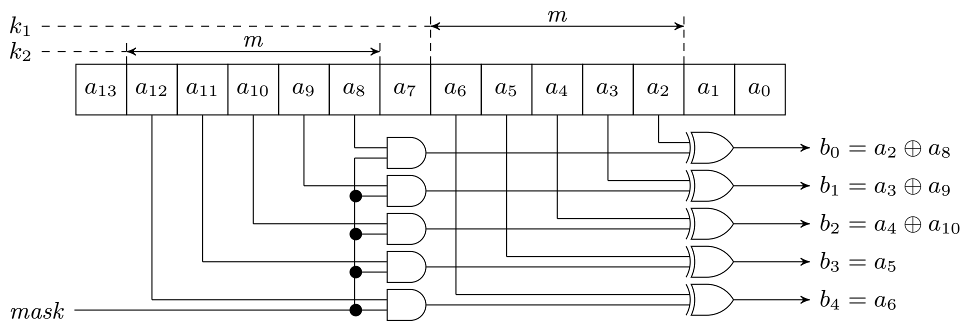
For each pair of bits (A_i, A_j) in the address space a corresponding correlation measure C_{ij} is calculated. This correlation is a real number ranging from 0 to 1 and can be calculated using the following equation:

$$C_{ij} = \frac{\min(E_{ij}, D_{ij})}{\max(E_{ij}, D_{ij})} \tag{12}$$

where E_{ij} and D_{ij} are the number of references having identical and different bits at A_i and A_j respectively.

To order and select the bits, which should be used to index the banks in the memory, the following algorithm is used by Givargis:

```
loop:
  select Ab = max { Q0, Q1, Q2, ... QM }
  for each Qi in { Q0, Q1, Q2, ... QM }
    Qi := Qi x Cbi
  halt when all Ai’s are selected
```



k_1
k_2
m
m
a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
$b_0 = a_2 \oplus a_8$
$b_1 = a_3 \oplus a_9$
$b_2 = a_4 \oplus a_{10}$
$b_3 = a_5$
$b_4 = a_6$
$mask$

Fig. 5. The bit-vector search algorithm selects the best values for k_1, k_2 and $mask$, e.g. $k_1 = 2$, $k_2 = 8$ and $mask = 7$.

This algorithm repeatedly selects an address bit with the highest corresponding quality measure and then updates the quality measures using the correlations. When all bits are selected in the order from highest to lowest quality the algorithm stops.

The goal of Givargis [3] was to evenly distribute all memory accesses of an application over a CPU's cache. Therefore all memory accesses are put in one set. In our application we want to reduce bank conflicts for each memory access pattern, and a balance must be found in optimizing all access patterns together. Therefore a set of memory addresses is created for each warp access pattern, and the heuristic is used to select the best bank addressing bits for the combination of these sets.

Let us take two sets of memory addresses, R^1 and R^2, for example corresponding with warp access patterns from two memory accesses in the same application. Ideally both patterns should access the memory banks with the lowest number of conflicts possible. Therefore the quality and correlation metrics of each set of memory addresses is calculated individually as described above. The respective quality and correlation measures are called Q_i^1, Q_i^2, C_{ij}^1 and C_{ij}^2. The proposed updated algorithm combines the quality metrics of each memory address set using the sum operator (+). It finds the best bits for indexing the banks in the memory as shown below:

```
loop:
  //calculate the combined quality for each address
      bit
  for each Qi in { }
    Qi := Q1i + Q2i
  select Ab = max { Q0, Q1, Q2, ... QM }
  for each Q1i in { Q10, Q11, Q12, ... Q1M }
    Q1i := Q1i x C1bi
  for each Q2i in { Q20, Q21, Q22, ... Q2M }
    Q2i := Q2i x C2bi
  halt when all Ai's are selected
```

This algorithm repeatedly calculates the combined quality of all address bits by taking the sum quality value of an address bits over the different sets of memory addresses. Then it selects the address bit with the highest combined quality value and updates the quality value for all address bits in all sets with their respective correlations.

Example: For a combination of stride=8 and stride=45 this algorithm selects address bits (ordered from highest to lowest quality): $\{A_3, A_4, A_5, A_6, A_7\}$. For a combination of stride=8 and stride=13 it selects $\{A_3, A_4, A_6, A_5, A_7\}$.

To use the GH also for the bitwise XOR hash function, all possible combinations of two address bits are created for each memory reference, as shown in the example of Table 3. These combinations are then used as the input bits for the GH. For each combination a quality measure Q_i and a correlation measure C_{ij} can be calculated as described above.

5.2.2 *Minimum Imbalance Heuristic*

In this section we present a new heuristic, named Minimum Imbalance Heuristic (MIH), that finds the best set of addressing bits minimizing the number of bank conflicts given a set of R memory references.

Similarly to Givargis, our heuristic sequentially computes the best addressing bits but it introduces two important modifications to the previous mentioned heuristic. Firstly, it employs a measure based on the imbalance of memory references to select the best addressing bits. Secondly, a new addressing bit is chosen taking into account the contribution of previous selected addressing bits.

In our heuristic $P_n = (b_{n-1}, b_{n-2}, \ldots, b_0)$ is the ordered sequence of n previously selected addressing bits. Then, the calculation of b_n (the following selected bit) is carried out as follows:

$$b_n = \arg\min_i (\text{imbalance}(A_i)) \qquad \text{for all } A_i \notin P_n \tag{13}$$

where $\text{imbalance}(A_i)$ is given by the expression:

$$\text{imbalance}(A_i) = \frac{\sum_{j=0}^{2^{n+1}} \left| h_i(j) - \frac{\|R\|}{2^{n+1}} \right|}{\|R\|} \tag{14}$$

and $h_i[j]$ is the j-th bin of a histogram h_i that contains the number of references with addressing bits $(A_i, b_{n-1}, \ldots, b_0)$ referencing position j. Notice that a perfect balance of the $\|R\|$ references to a set of 2^{n+1} histogram bins should result in $\frac{\|R\|}{2^{n+1}}$ accesses per bin. This quantity is subtracted from the real number of accesses per bin to calculate the imbalance per memory position. Finally, the calculated imbalances per memory position are added to obtain the total imbalance, which is normalized dividing by the total number of references $\|R\|$.

```
P0={ empty }
for(j=0; j<n; j++)
  b_j = min_i(imbalance(Ai, Pj)) or all Ai not
      belonging to Pj
  Pj+1 = { Ab, Pj } // New selected bit Ab is added
endfor                 to the ordered list Pj
```

Listing 1. Implementation of the Minimum Imbalance Heuristic.

$$\begin{aligned} h_0 &= (4,4) & I_0 &= 0 \\ h_1 &= (3,5) & I_1 &= 0.25 \\ h_2 &= (4,4) & I_2 &= 0 \\ h_3 &= (4,4) & I_3 &= 0 \\ h_4 &= (5,3) & I_4 &= 0.25 \end{aligned}$$

(a) $b_0 = \arg\min_i(I_i) = 0$

$$\begin{aligned} h_1 &= (3,0,1,4) & I_1 &= 0.75 \\ h_2 &= (0,4,4,0) & I_2 &= 1 \\ h_3 &= (2,2,2,2) & I_3 &= 0 \\ h_4 &= (3,2,1,2) & I_4 &= 0.25 \end{aligned}$$

(b) $b_1 = \arg\min_i(I_i) = 3$

$$\begin{aligned} h_1 &= (1,0,2,0,1,2,0,2) & I_1 &= 0.75 \\ h_2 &= (0,2,0,2,2,2,0,0) & I_2 &= 1 \\ \\ h_4 &= (2,1,1,1,0,1,1,1) & I_4 &= 0.25 \end{aligned}$$

(c) $b_2 = \arg\min_i(I_i) = 4$

Fig. 6. Minimum Imbalance Heuristic example. In three steps, (a), (b), (c), the set of addressing bits $P = \{b_0, b_1, b_2\} = \{0, 3, 4\}$ is selected for a set of 8 addresses $\{27, 12, 6, 19, 11, 4, 28, 3\}$ by calculating a histogram h_n and imbalance value I_n for each bit n in each step.

As it can be deduced from the previous expressions, the information employed by our method to select addressing bits for values of $n > 0$ (more than one addressing bit) is much richer than those employed for Givargis as all sets of addresses referenced by P_j at j-th step are considered.

Finally, the heuristic can be written as shown in Listing 1.

In Fig. 6 an example of the proposed heuristic is shown. Eight references ($\|R\| = 8$) to positions 27, 12, 6, 19, 11, 4, 28 and 3 of a memory organized in eight banks are carried out. The figure shows the three iterations needed by our heuristic to select the addressing bits employed to address the memory banks. Consequently, after applying our heuristic the bank addressing bits are reordered as $\{A4, A3, A0\}$.

Like the GH, the MIH calculates the best set of bank addressing bits for one set of memory references R. Since we want to reduce bank conflicts for each memory access pattern in an application, we have to combine the imbalance values for each set of references to find the overall best possible set of bank addressing bits. This is achieved by adding the imbalance values of every memory reference together (for all address bits $A_i \notin P_n$), and selecting the bit with the lowest combined imbalance value. Therefore Eq. 13 is replaced by Eq. 15.

$$b_n = \arg\min_i \left(\sum_R \text{imbalance}\left(A_i^R\right) \right)$$
$$\text{for all } A_i^R \notin P_n \text{ for all memory reference sets } R \quad (15)$$

To use the Minimum Imbalance Heuristic also for the bitwise XOR hash function, all possible combinations of two address bits are created for each memory reference, as shown in the example of Table 3. These combinations are then used as the input bits for the Minimum Imbalance Heuristic.

6 Framework for bank conflict reduction

To test the performance improvements of the proposed hash functions of Section 4 and the quality of the heuristics of Section 5, a framework is developed as shown in Fig. 7. It automatically processes an application's kernel code, analyses the memory access patterns and configures the proposed hash functions using the aforementioned heuristics.

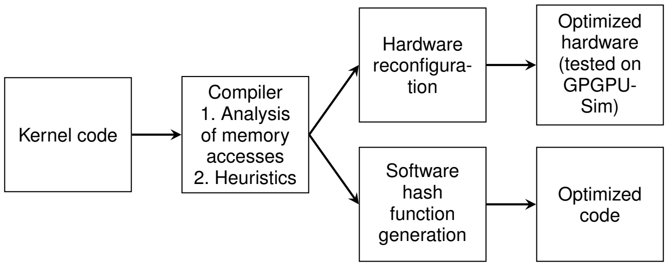

Kernel code
Compiler
1. Analysis of memory accesses
2. Heuristics
Hardware reconfiguration
Optimized hardware (tested on GPGPU-Sim)
Software hash function generation
Optimized code

Fig. 7. Our framework for bank conflict reduction. It encompasses hardware and software approaches.

The first step in the framework is to analyze the kernel code and determine the hash functions' parameters. The memory accesses analysis is done on source code level, based on techniques developed in [10]. On some occasions the analysis produces sub-optimal results, for example because not all memory accesses in a loop are known due to an unknown loop count. In this case a memory access trace can be made which is then analyzed. The results of the analysis are used by the heuristics and search algorithm described in Section 5 to find the best possible parameters for each hash function.

The effects of the hardware hash functions on bank conflict numbers and execution time are tested using a modified version of GPGPU-Sim in which the different hash functions are integrated. The source code of the benchmark applications is modified by inserting a setup function before a kernel is launched. This setup function will configure the hash function being tested with the aforementioned parameters.

The hash functions can also be applied in software in the kernel code itself. In Section 7.2 we do this by hand with the aim of presenting a proof of concept, which demonstrates the benefits of doing it by a compiler. The hash function is inserted in every shared memory access and is configured using the same parameters. Only the bit-vector XOR hash is tested as a software solution, since it gives very good results (see Section 7) and proves to be a good trade-off between added address calculation costs and memory access bank conflict reductions.

7 Experimental results

The effect of the bit-vector XOR, bitwise permutation and bitwise XOR hash functions on the number of bank conflicts and consequently the execution time has been tested on a number of benchmarks. Most of the benchmarks are taken from the CUDA SDK 6.0, Rodinia 2.4 [11] and Parboil 2.5 [12]. We added two more benchmarks that can be burdened by bank conflicts: matrix-scan [13], [14], and FFT [15].

The parameters of the bit-vector XOR hash function are determined by using the search algorithm described in Section 5.1. The parameters of the bitwise permutation and bitwise XOR hash functions are determined using the Givargis heuristic and the proposed Minimum Imbalance

TABLE 5
List of benchmarks. The parameters for each of the hash functions (bit-vector XOR, bitwise permutation and bitwise XOR) are determined by either code analysis or a memory trace. The parameters for the bitwise hash functions are determined using either the Givargis heuristic or the proposed Minimum Imbalance Heuristic.

Name	Method	bit-vector XOR hash	bitwise perm. Givargis	bitwise perm. Min. Imbalance	bitwise XOR hash Givargis	bitwise XOR hash Min. Imbalance
conv-1	Analysis	k1=0 k2=1 mask=16	(0) (1) (2) (3) (5)	(0) (1) (2) (3) (5)	(0) (0^1) (0^2) (0^3) (0^5)	(0) (1) (2) (3) (5)
conv-2	Analysis	k1=0 k2=4 mask=14	(0) (4) (5) (6) (7)	(0) (4) (5) (6) (7)	(0) (0^4) (0^5) (0^6) (0^7)	(0) (4) (5) (6) (7)
dct8x8-1	Analysis	k1=0 k2=5 mask=7	(3) (4) (0) (1) (2)	(3) (4) (0) (1) (2)	(0^3) (0^4) (0^5) (1^6) (2^7)	(3) (4) (0^5) (1^6) (2^7)
dct8x8-2	Analysis	k1=0 k2=5 mask=7	(3) (4) (0) (1) (2)	(3) (4) (0) (1) (2)	(0^3) (0^4) (0^5) (1^6) (2^7)	(3) (4) (0^5) (1^6) (2^7)
dwtHaar1D	Trace	k1=0 k2=5 mask=15	(4) (3) (2) (1) (5)	(4) (3) (2) (1) (5)	(3^8) (2^7) (1^6) (0^5) (0^4)	(3^8) (2^7) (1^6) (0^5) (4)
FFT-1	Trace	k1=1 k2=10 mask=1	(10) (5) (1) (2) (3)	(10) (5) (4) (3) (2)	(1^2) (1^3) (1^4) (1^5) (1^10)	(1^2) (1^3) (1^4) (1^5) (1^10)
FFT-2	Trace	k1=1 k2=0 mask=0	(1) (2) (3) (4) (5)	(1) (2) (3) (4) (5)	(0^1) (0^2) (0^3) (0^4) (0^5)	(1) (2) (3) (4) (5)
FWT	Analysis	k1=0 k2=2 mask=31	(0) (1) (2) (3) (4)	(4) (3) (2) (1) (0)	(0^2) (0^3) (0^4) (0^5) (1^6)	(0^2) (0^3) (0^4) (0^5) (1^6)
hist64	Trace	k1=6 k2=4 mask=1	(6) (7) (8) (9) (10)	(8) (7) (6) (10) (9)	(6) (6^7) (6^8) (7) (7^8)	(8) (7) (6) (10) (9)
hist256	Trace	k1=0 k2=6 mask=28	(8) (9) (10) (11) (12)	(0) (1) (2) (3) (4)	(4^8) (3^9) (2^12) (1^10) (0^7)	(4^8) (3^9) (2^11) (0^10) (1^2)
lavaMD	Analysis	k1=1 k2=6 mask=3	(3) (4) (5) (6) (7)	(3) (4) (5) (6) (7)	(0^3) (0^4) (0^5) (1^6) (2^7)	(3) (4) (5) (1^6) (2^7)
LUD-1	Analysis	k1=0 k2=5 mask=7	(0) (1) (2) (3) (4)	(0) (1) (2) (3) (7)	(0^4) (1^5) (2^6) (3^7) (0^1)	(0^4) (1^5) (2^6) (3^7) (13)
LUD-2	Analysis	k1=0 k2=5 mask=15	(4) (0) (1) (2) (3)	(4) (0) (1) (2) (3)	(0^4) (1^5) (2^6) (3^7) (0^8)	(4) (0^5) (1^6) (2^7) (3^8)
matrix scan	Analysis	k1=0 k2=5 mask=7	(3) (4) (5) (6) (7)	(3) (4) (7) (6) (5)	(0^3) (0^4) (0^5) (1^6) (2^7)	(3) (4) (0^5) (1^6) (2^7)
MRI-grid-1	Trace	k1=0 k2=5 mask=31	(4) (3) (5) (2) (1)	(4) (3) (2) (5) (1)	(4^9) (3^8) (2^7) (1^6) (0^5)	(0^5) (1^6) (2^7) (3^8) (4^9)
MRI-grid-2	Trace	k1=1 k2=6 mask=1	(5) (4) (3) (2) (6)	(2) (3) (4) (5) (1)	(1^6) (0^5) (0^4) (4^5) (0^3)	(2) (3) (4) (5) (1^6)
MRI-grid-3	Trace	k1=1 k2=6 mask=1	(5) (4) (3) (6) (2)	(2) (3) (4) (5) (1)	(1^6) (0^5) (0^4) (4^5) (0^3)	(2) (3) (4) (5) (1^6)
MRI-grid-4	Trace	k1=0 k2=5 mask=3	(2) (3) (4) (6) (5)	(6) (3) (2) (5) (4)	(0^2) (3^6) (2^3) (2^4) (1^4)	(6) (3) (2) (0^4) (1^5)
NW-1	Trace	k1=0 k2=5 mask=7	(4) (5) (6) (1) (0)	(4) (5) (6) (7) (0)	(1^4) (2^5) (0^6) (4^5) (3^7)	(0^5) (2^4) (3^6) (1^7) (4)
NW-2	Trace	k1=0 k2=5 mask=15	(4) (5) (6) (1) (0)	(4) (5) (6) (7) (0)	(1^4) (2^5) (0^6) (4^5) (3^7)	(1^4) (2^5) (3^6) (0^7) (4)
reduction	Trace	k1=0 k2=5 mask=7	(4) (3) (5) (2) (1)	(4) (3) (2) (1) (5)	(2^7) (1^6) (0^5) (0^4) (4^5)	(2^7) (1^6) (0^5) (4) (3)
transpose	Analysis	k1=0 k2=4 mask=14	(0) (4) (1) (2) (3)	(0) (4) (1) (2) (3)	(0) (0^4) (1^4) (1^5) (2^6)	(0) (4) (1^5) (2^6) (3^7)

Heuristic described in Section 5.2.1 and Section 5.2.2 respectively. The memory access patterns used by the search algorithms and the heuristics are extracted from the benchmarks using either source code analysis or a memory access trace, as describe in Section 6. The resulting parameters for each benchmark and hash function are shown in Table 5.

The proposed hash functions can be implemented in hardware, but also in software. To evaluate the impact of the hash functions in hardware, all hash functions are implemented in GPGPU-Sim version 3.2.0 [16], which is configured as an NVIDIA GTX 480 (Fermi) GPU. The effect of the hash functions on the number of bank conflicts and the execution time for the benchmarks is evaluated in Section 7.1. The hardware cost in terms of chip-area, power consumption and added memory access latency has been evaluated in Section 4.5. The use of hash functions as a software solution is evaluated in Section 7.2, which shows the benefits of adding hash functions to memory accesses either manually by a programmer or automatically by a compiler.

7.1 Hardware hash function results

The bank conflict reduction of the various hash functions is compared against the regular GPU which does not use a hash function in the addressing of the banks of the shared memory. An additional fixed bit-vector hash function used in [17] has been also included in this study for comparison purposes. This fixed hash function calculates the bank index as: $bank = addr[0 \dots 4] \oplus addr[5 \dots 9]$.

The relative number of bank conflicts removed by each hash function for the benchmarks listed in Table 5 is shown in Fig. 8. For 14 of the 22 benchmarks the fixed bit-vector hash function from [17] removes all bank conflicts, and all configurable hash functions do so as well. For the other 8 benchmarks the configurable hash functions also remove all bank conflicts, except for the histogram benchmarks which use indirect memory accesses. Average values are displayed in Table 6. Thus, the fixed bit-vector XOR works well and removes 86% of all bank conflicts on average. The configurable bit-vector XOR hash function (Section 4.2) improves the number of removed bank conflicts to 96%. The bitwise permutation hash function performs worse, regardless if the Givargis or Minimum Imbalance Heuristic is used. It only removes 49% and 47% respectively of the bank conflicts. The bitwise XOR hash function removes 88% of all bank conflicts if the parameters are determined using the Givargis heuristic. In case the Minimum Imbalance Heuristic is used to determine the parameters, 97% of all bank conflicts are removed and only the histogram (hist64 and hist256) benchmarks have bank conflicts remaining.

The histogram algorithm is a special kind of algorithm in which the location of the memory accesses is dependent on the input data itself, and not (just) the input data dimensions. Therefore one input image can result in more bank conflicts than another. To take this into account in the experiments, the parameters of the hash functions are

TABLE 6
Bank conflicts reduction percentage achieved by the different hash functions and heuristics.

	Heuristic			
Hash function	None	Exhaustive search	GH	MIH
fixed bit-vector XOR	86%	-	-	-
bit-vector XOR	-	96%	-	-
bit-wise permutation	-	-	49%	47%
bit-wise XOR	-	-	88%	97%

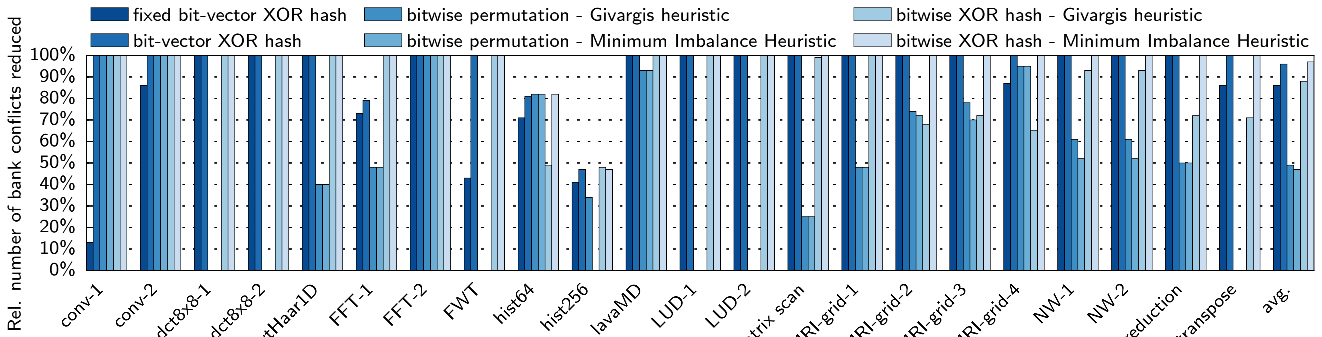


Fig. 8. Relative number of bank conflicts removed by various hash function compared to a baseline GPU (no hash function) for a set of benchmarks.

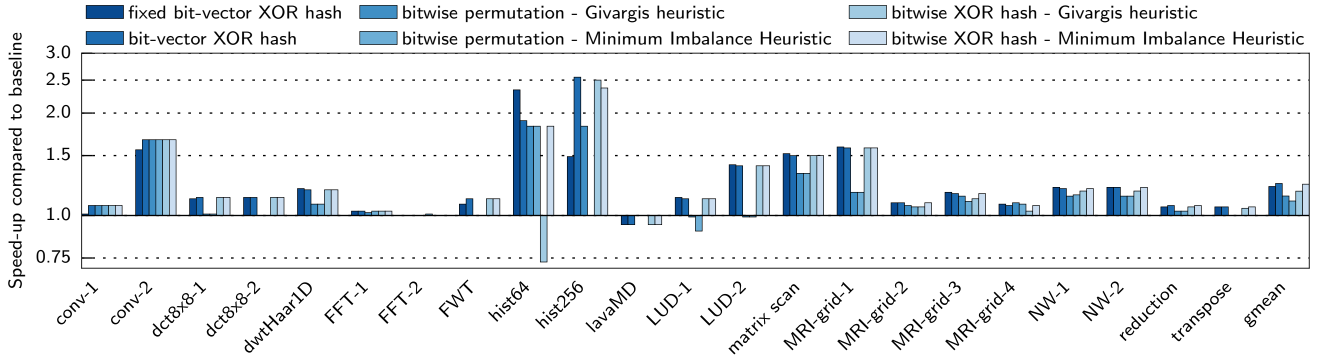


Fig. 9. Overall speed-up obtained by the various hash functions compared to a baseline GPU (no hash function) for a set of benchmarks.

determined using a (randomly selected) image, and the results of Fig. 8 and Fig. 9 are obtained by averaging the results of ten other images.

An application does not consist solely of memory accesses, therefore the performance gains are less than the bank conflicts reduction numbers. The speed-up obtained by the various hash functions over the baseline GPU is shown in Fig. 9 for a set of benchmarks. The configurable hash functions perform similar to the fixed hash function for the 14 benchmarks in which all conflicts are removed by any hash function. For the other 8 benchmarks the configurable hash functions show a small performance improvement over the fixed hash function, except for the hist64 benchmark. The geometric mean of the speed-up for the fixed bit-vector XOR [17] is $1.21\times$ compared to a baseline GPU. The configurable bit-vector XOR hash function performs a little bit better with a speed-up of $1.24\times$. The bitwise permutation hash function removes fewer bank conflicts, and consequently also shows a smaller speed-up of $1.14\times$ and $1.10\times$ for the Givargis and Minimum Imbalance Heuristic respectively. The bitwise XOR hash functions score best, with a speed-up of $1.18\times$ and $1.24\times$ for the Givargis and Minimum Imbalance Heuristic respectively. Because memory accesses using the flexible hash functions require one extra clock cycle (see Section 4.5), some applications experience a slowdown due to the configurable hash functions, see for example the lavaMD benchmark in Fig. 9. The hist256 benchmark benefits the most from the configurable hash functions with a speed-up of $2.5\times$. It consists mainly of load and store operations to the scratchpad memory, and is therefore very sensitive to bank conflicts.

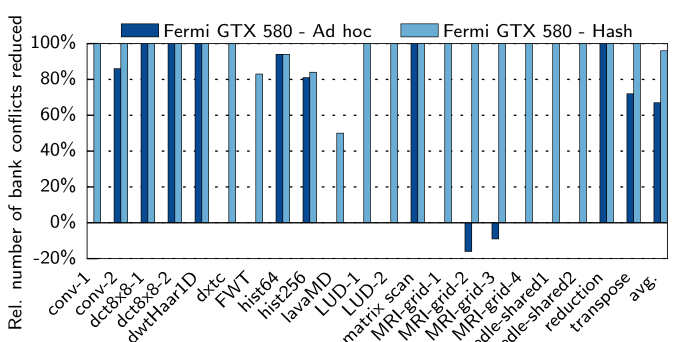
Some applications use the scratchpad memory but do not have bank conflicts. The performance impact of the extra cycle of latency for every memory access (see Section 4.5) on these kind of applications has been evaluated by testing five benchmarks: back propagation, srad and hotspot from Rodinia [11], scalar product from the CUDA SDK and matrix-matrix multiply from Parboil [12]. The average loss in execution time is only 1%, and the maximum performance loss is 4.5% for the srad benchmark.

7.2 Software hash function results

As indicated in Section 6, our framework can be integrated in a compiler, which would generate optimized code using hash functions. That way, such optimization would be transparent for the programmer. In this section, we carry out a proof of concept applying software optimization manually. With this aim, we use the bit-vector XOR hash functions shown in Table 5 for a number of the benchmarks. The code in Listing 2 illustrates how the software optimization can be applied in a kernel. This sample CUDA code corresponds to the lavaMD benchmark, where `rA_shared`, `rB_shared`, and `qB_shared` are three arrays in scratchpad memory.

Experiments have been run on real hardware: GTX 580 with Fermi architecture, and K20 with Kepler architecture. The benchmark dxtc has only been run on a GTX 280 with Tesla architecture. The shared memory of this GPU has 16 banks. More recent NVIDIA GPUs have 32-banked shared memories, and dxtc does not present bank conflicts on them.

Fig. 10 presents the relative number of bank conflicts reduced on the GPUs by either applying ad-hoc optimizations or hash functions. Results are compared to a baseline



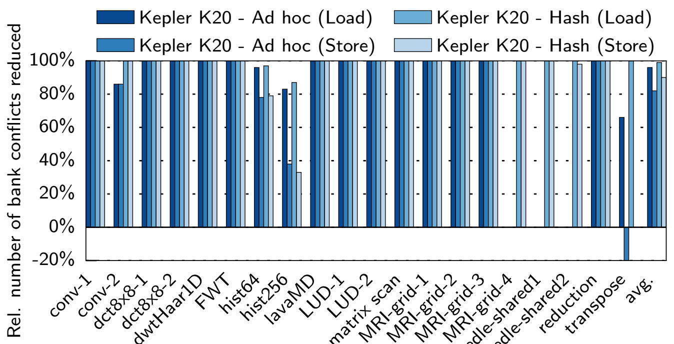
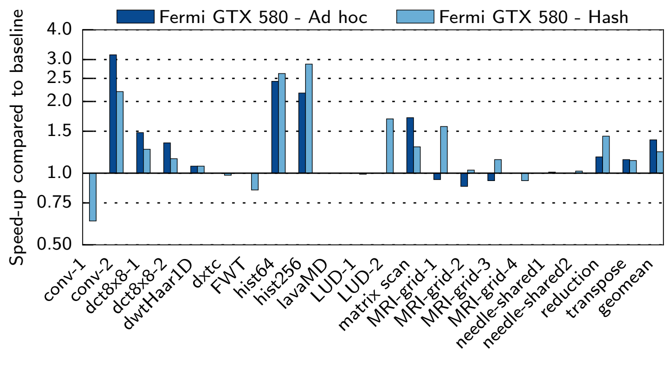


Fig. 10. Relative number of bank conflicts removed by an ad-hoc optimization technique (typically, padding), and a bit-vector XOR hash function compared to a baseline implementation (neither hash function, nor ad-hoc technique) for a set of benchmarks, on GTX 580 (Fermi) and K20 (Kepler). dxtc has been tested on GTX 280 (Tesla architecture).



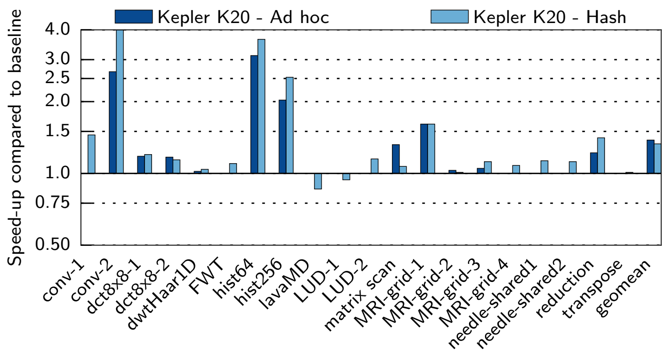


Fig. 11. Overall speed-up obtained by an ad-hoc optimization technique (typically, padding), and a bit-vector XOR hash function compared to a baseline implementation (neither hash function, nor ad-hoc technique) for a set of benchmarks, on GTX 580 (Fermi) and K20 (Kepler). dxtc has been tested on GTX 280 (Tesla architecture).

implementation in which no specific software technique has been used to reduce bank conflicts. This figures have been obtained with the CUDA command-line profiler. For Fermi and Tesla, the profiler returns a single number as the bank conflict count. For Kepler, it differentiates between shared memory loads and stores.

For each benchmark, two columns may appear. The one on the left (darker color) stands for the results for an ad-hoc technique, such as padding, to reduce bank conflicts. This is the technique (if any) that can be found in the original code. The right column (lighter color) represents the results for a hash function. As it can be seen, hash functions always obtain at least the same reduction of the number of bank conflicts.

The speed-up obtained by the ad-hoc techniques and the hash functions is shown in Fig. 11. In general, the hash functions achieve a speed-up relative to the baseline that is comparable to the ad-hoc techniques. The geometric mean of the speed-up of the hash functions to the baseline implementations is $1.23\times$ on Fermi and $1.33\times$ on Kepler. Moreover, it is remarkable that the ad-hoc techniques are only applied to 12 out of 21 benchmarks.

In those cases where there is a small performance loss (e.g. conv-1 on GTX 580 and lavaMD on K20), the reduction in the number of bank conflicts does not compensate for the cost of the hash function (shift and logic operations). It is worth noting that in this cases no ad-hoc techniques were used in the original code. The number of bank conflicts is so little that no improvement is obtained from them.

The benchmarks hist64 and hist256 are only a sample of the benefits that hash functions can have on histogramming. In these tests, histograms of 64 and 256 bins have been calculated for 10 real images using a replication factor of 32, which is the number of sub-histograms in shared memory per thread block. More details of the use of hash functions on software-optimized implementations of histogramming, such as [18] and [19], can be found in [17].

In summary, a programmer could benefit from our framework, since this can generate a hash function that reduces the bank conflicts at least as effectively as manually-applied ad-hoc techniques. Actually, the optimization could be transparent for the programmer, if the proposed framework of Section 6 is integrated into a compiler. Hash functions also save memory space compared to the padding approach, so that occupancy might be increased in some cases.

```
// k1 = 1, k2 = 6, mask = 3
__device__ int hash(int address){
  int addr_xor = (address >> 6) & 3;
  addr_xor = addr_xor ^ (address >> 1);
  return addr_xor;
}
...
d.x = rA_shared[hash(4*wtx+1)]
    - rB_shared[hash(4*j+1)];
...
fA[wtx].v += qB_shared[hash(j)] * vij;
```

Listing 2. Software implementation of the bit-vector XOR hash function (top) and the CUDA code taken from the lavaMD benchmark (bottom).

8 Related work

GPU memory access pattern classification have been introduced in [4], [5]. Jang et al. describe in [4] six different memory access patterns which are used for loop vectorization for AMD GPUs and memory selection (e.g. global, shared, texture, constant) on NVIDIA GPUs. Fang et al. [5] specify 33 memory access patterns (MAPs). Each MAP consists of an inter- and intra-thread component. The MAPs are used to predict performance for various platforms (e.g. CPU or GPU) by querying a database of MAP performance of a particular platform. In this work we reduce the number of patterns found to only four: *linear*, *stride*, *block* and *random*, and use the classification in the search algorithm and heuristics to configure the proposed hash functions.

Previous work on GPU scratchpad hash functions proposed a fixed hash function for all applications [17]. These hash functions can also be used to avoid atomic conflicts in some implementations of atomic operations, such as NVIDIA Tesla, Fermi and Kepler architectures [17], [20], [21]. Other works propose configurable hash functions per application for CPU caches [3], [9], [22], [23], [24] and interleaved memories [25], [26]. Patel et al. [22], [23] find the best possible hash function for a single set of memory references. The proposed methods in this work find a hash function for all sets of memory references. This work extends the heuristics from previous work [3] to configure the proposed hash functions which are an improvement performance-wise compared to the fixed hash functions [17].

Instead of configuring the indexing of the banked memory to reduce bank conflicts, it is also possible to change the memory itself, as shown in [27]. Diamond et al. show that it is possible to efficiently address a banked memory with an arbitrary modulus (instead of 2^N). When implemented on a GPU's L1 cache and scratchpad memory 98% of all bank and set conflicts can be removed, resulting in an average speed-up of 24%. When the arbitrary modulus indexing is only applied to the scratchpad memory, they get a geometric mean 11% speed-up for 5 benchmarks. In our work, we have proposed the use of configurable hash functions to achieve a geometric mean 24% speed-up on 22 benchmarks.

9 Conclusions

In this work four configurable hash functions for banked memories are evaluated: bit-vector permutation, bit-vector XOR, bitwise permutation and bitwise XOR. The impact on the number of bank conflicts and the resulting performance gains of hardware implementations are assessed on the NVIDIA Fermi architecture using GPGPU-Sim. In total 22 benchmarks from the NVIDIA CUDA SDK, Rodinia and Parboil benchmark suites are tested. Bank conflicts are removed completely for 20 benchmarks, while a fixed hash function from previous work [17] only managed to remove all bank conflicts for 14 benchmarks. Bank conflict are reduced on average by 86% for this fixed hash function, by 96% for the configurable bit-vector XOR, and by 97% for the bitwise XOR hash function using the proposed heuristic. Only the two histogram benchmarks with their indirect, data dependent memory references have bank conflicts remaining. In terms of performance, a geometric mean 24% speed-up over all benchmarks is attained for the configurable hash functions. Also the hardware costs in terms of latency, power and area are evaluated. These are estimated to be no more than 0.2% of the power and area budget of a contemporary GPU for the most complex configurable hash function.

Next to this hardware solution a software approach is proposed, which does not require any changes to the hardware. This software approach can reduce the average number of bank conflicts by 99% in load accesses and 90% in store accesses, and leads to a 1.33× speed-up on the NVIDIA Kepler architecture.

To configure the hash functions, the Givargis heuristic [3] is extended to select the overall best bank addressing bits for multiple sets of memory references, not just for a single set. Also the Minimum Imbalance Heuristic is introduced, which removes 97% of all bank conflicts for the bitwise XOR hash functions, outperforming the Givargis heuristic.

Future work

The proposed bitwise / bit-vector permutation and XOR hash functions are initialized at launch time for the complete duration of a kernel's execution. An alternative would be to incorporate the hash functions' parameters in the load- and store instructions, which makes it possible to use different hash functions for different memory accesses. One problem is that different hash functions can map different memory addresses to the same memory location. This can be prevented by dividing the memory in regions, for example by using the most significant bits of the address.

Acknowledgments

This work was partly funded by the Dutch government in their Point-One research program within the Morpheus project PNE101003. We also thank NVIDIA (for hardware donation to the University of Málaga under CUDA Research Center Awards), and the Ministry of Education of Spain (TIN2010-16144) and the Junta de Andalucía of Spain (TIC-1692) for financial support.

References

[1] J. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-M. Hwu, and N. Obeid, "Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems," *Computer*, vol. 45, no. 8, pp. 26–32, August 2012.

[2] H. Vandierendonck and K. De Bosschere, "XOR-based hash functions," *Computers, IEEE Transactions on*, vol. 54, no. 7, pp. 800–812, July 2005.

[3] T. Givargis, "Improved indexing for cache miss reduction in embedded systems," in *Design Automation Conference, 2003. Proceedings*, June 2003, pp. 875–880.

[4] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 105–118, Jan 2011.

[5] A. L. V. Jianbin Fang, Henk Sips, "Aristotle: A Performance Impact Indicator for the OpenCL Kernels Using Local Memory," *Scientific Programming*, vol. 22, pp. 239–257, 2014.

[6] AMD, "CodeXL profiler."

[7] NVIDIA, "CUDA profiler."

[8] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide 6.0," 2014.

[9] H. Vandierendonck, P. Manet, and J.-D. Legat, "Application-specific reconfigurable XOR-indexing to eliminate cache conflict misses," in *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, ser. DATE '06, 2006, pp. 357–362.

[10] G.-J. van den Braak, B. Mesman, and H. Corporaal, “Compile-time GPU memory access optimizations,” in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, July 2010, pp. 200–207.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[12] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, Mar 2012.

[13] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, “Fast Scan Algorithms on Graphics Processors,” in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS ’08. New York, NY, USA: ACM, 2008, pp. 205–213.

[14] S. Yan, G. Long, and Y. Zhang, “StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’13, 2013, pp. 229–238.

[15] E. Gutierrez, S. Romero, M. A. Trenas, and O. Plata, “Experiences with Mapping Non-linear Memory Access Patterns into GPUs,” in *Proceedings of the 9th International Conference on Computational Science: Part I*, ser. ICCS ’09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 924–933.

[16] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, April 2009, pp. 163–174.

[17] G.-J. van den Braak, J. Gómez-Luna, H. Corporaal, J. González-Linares, and N. Guil, “Simulation and architecture improvements of atomic operations on GPU scratchpad memory,” in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, Oct 2013, pp. 357–362.

[18] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, “An Optimized Approach to Histogram Computation on GPU,” *Mach. Vision Appl.*, vol. 24, no. 5, pp. 899–908, Jul. 2013.

[19] G.-J. van den Braak, C. Nugteren, B. Mesman, and H. Corporaal, “GPU-vote: A Framework for Accelerating Voting Algorithms on GPU,” in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par’12, 2012, pp. 945–956.

[20] B. Coon, P. Mills, J. Nickolls, and L. Nyland, “Lock mechanism to enable atomic updates to shared memory,” *US Patent 8055856*, 2011.

[21] J. Gomez-Luna, J. Gonzalez-Linares, J. Benavides Benitez, and N. Guil Mata, “Performance Modeling of Atomic Additions on GPU Scratchpad Memory,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 11, pp. 2273–2282, Nov 2013.

[22] K. Patel, E. Macii, L. Benini, and M. Poncino, “Reducing cache misses by application-specific re-configurable indexing,” in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, Nov 2004, pp. 125–130.

[23] K. Patel, L. Benini, E. Macii, and M. Poncino, “Reducing Conflict Misses by Application-Specific Reconfigurable Indexing,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 12, pp. 2626–2637, Dec 2006.

[24] A. González, M. Valero, N. Topham, and J. M. Parcerisa, “Eliminating cache conflict misses through XOR-based placement functions,” in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS ’97. New York, NY, USA: ACM, 1997, pp. 76–83.

[25] J. Frailong, W. Jalby, and J. Lenfant, “Xor-schemes: A flexible data organization in parallel memories,” in *International Conference on Parallel Processing (ICPP)*, 1985, pp. 276–283.

[26] B. R. Rau, “Pseudo-randomly interleaved memory,” in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ser. ISCA ’91. New York, NY, USA: ACM, 1991, pp. 74–83.

[27] J. Diamond, D. Fussell, and S. Keckler, “Arbitrary Modulus Indexing,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, Dec 2014, pp. 140–152.

Gert-Jan van den Braak received the MSc degree in Electrical engineering from the Eindhoven University of technology, the Netherlands in 2009. Currently he is a PhD candidate at the Eindhoven University of Technology. His main research topics are GPU architectures and the mapping of irregular voting algorithms on GPUs.

Juan Gómez-Luna received the B.S. degree in Telecommunication Engineering from the University of Sevilla, Spain, in 2001. He obtained the Ph.D. degree in Computer Science from the University of Córdoba, Spain, in 2012. Since 2005, he is a lecturer at the University of Córdoba. His research interests focus on GPU and heterogenous computing.

José María González-Linares received the B.S. and Ph.D. degrees in Telecommunication Engineering from the University of Málaga, Spain, in 1995 and 2000, respectively. Since 2002, he has been an associate professor with the Department of Computer Architecture, at the University of Málaga. He has published more than 30 papers in international journals and conferences. His research interests are in the areas of heterogeneous computing and video and image processing.

Henk Corporaal is Professor in Embedded System Architectures at the Einhoven University of Technology (TU/e) in The Netherlands. He has gained a MSc in Theoretical Physics from the University of Groningen, and a PhD in Electrical Engineering, in the area of Computer Architecture, from Delft University of Technology.

Corporaal has co-authored over 300 journal and conference papers. Furthermore he invented a new class of VLIW architectures, the Transport Triggered Architectures, which is used in several commercial products, and by many research groups.

His research is on low power single and multi-processor architectures, their programmability, and the predictable design of soft- and hard real-time systems. This includes research and design of embedded system architectures, accelerators, the exploitation of all kinds of parallelism, and the (semi-)automated mapping of applications to these architectures. For further details see http://corporaal.org.

Nicolás Guil received the B.S. degree in Physics from the University of Sevilla, Spain, in 1986 and the Ph.D. degree in Computer Science from the University of Málaga in 1995. Currently, he is full professor with the Department of Computer Architecture in the University of Málaga. He has published more than 60 papers in international journals and conferences. His research interests are parallel computing, and video and image processing.