Future Generation Computer Systems 30 (2014) 14–26

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Optimizing convolution operations on GPUs using adaptive tiling

Ben van Werkhoven [a,*], Jason Maassen [a,b], Henri E. Bal [a], Frank J. Seinstra [a,b]

[a] Department of Computer Science, VU University Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
[b] Netherlands eScience Center, Science Park 140, 1098 XG Amsterdam, The Netherlands

HIGHLIGHTS

- We present an extensive study of the optimization process of convolutions on GPUs.
- Existing optimization techniques are too limited in performance and flexibility.
- We present a new optimization for convolutions on GPUs called adaptive tiling.
- Our implementation is the best performing one in the spatial domain available to date.

ARTICLE INFO

Article history:
Received 20 November 2012
Received in revised form
6 August 2013
Accepted 5 September 2013
Available online 16 September 2013

Keywords:
High-performance computing
GPU computing
Parallel applications
GPU clusters
High-level programming models

ABSTRACT

The research domain of Multimedia Content Analysis (MMCA) considers all aspects of the automated extraction of knowledge from multimedia data. High-performance computing techniques are necessary to satisfy the ever increasing computational demands of MMCA applications. The introduction of Graphics Processing Units (GPUs) in modern cluster systems presents application developers with a challenge. While GPUs are well known to be capable of providing significant performance improvements, the programming complexity vastly increases. To this end, we have extended a user transparent parallel programming model for MMCA, named Parallel-Horus, to allow the execution of compute intensive operations on the GPUs present in the cluster. The most important class of operations in the MMCA domain are convolutions, which are typically responsible for a large fraction of the execution time. Existing optimization approaches for CUDA kernels in general as well as those specific to convolution operations are too limited in both performance and flexibility. In this paper, we present a new optimization approach, called *adaptive tiling*, to implement a highly efficient, yet flexible, library-based convolution operation for modern GPUs. To the best of our knowledge, our implementation is the most optimized and best performing implementation of 2D convolution in the spatial domain available to date.

1. Introduction

Multimedia Content Analysis (MMCA) investigates methods of automated knowledge extraction from image, video, and multimedia data. Research in the domain is driven by emerging applications, ranging from real-time analysis of video data from surveillance cameras, to searching digital television archives [1]. The massive amounts of data in such applications makes storing, cataloging, processing, and retrieving of information a very challenging task. As a result, high-performance computing is indispensable in the MMCA domain.

It is unrealistic to expect MMCA researchers to also become experts in high-performance computing. Therefore, it is essential to develop efficient programming models that hide the intrinsic complexities of the underlying computing hardware. In the literature, a number of such *user transparent* parallel programming models have been described (e.g. see [2,3]). These programming models are based on a software library of pre-parallelized compute kernels that cover the bulk of all commonly applied MMCA functionality. Generally, these kernels are designed for data parallel execution on *traditional* compute clusters.

Today, many emerging cluster systems are equipped with Graphics Processing Units (GPUs). Although GPUs are capable of providing significant performance improvements, programming complexity vastly increases. As current MMCA programming models for cluster systems do not incorporate GPUs, only a fraction of the compute power of modern clusters is exploited. Clearly, there is a need for easy-to-use and efficient programming models for high-performance multimedia computing on GPU-equipped cluster systems.

In this paper, we present an extensively optimized library-based implementation for convolution operations. Convolutions

* Corresponding author. Tel.: +31205985849.
E-mail addresses: ben@cs.vu.nl (B. van Werkhoven), j.maassen@esciencecenter.nl (J. Maassen), bal@cs.vu.nl (H.E. Bal), f.seinstra@esciencecenter.nl (F.J. Seinstra).

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.09.003

are essential to signal and image processing applications, and are typically responsible for a large fraction of the application's execution time.

This work is part of a larger effort to obtain an implementation of the Parallel-Horus [4] programming model that allows sequentially written MMCA programs to execute as highly optimized applications for GPU-clusters without requiring *any* parallelization effort from the application programmer. Because 2D convolution operations can be parallelized over multiple compute nodes simply by splitting and merging the input and output images across the nodes, this paper only discusses optimizations within a single GPU compute node.

This paper provides the following contributions:

- We present an extensive study of the optimization process of 2D convolution and separable convolution operations on modern graphics cards.
- We demonstrate that once all the well-known optimization techniques have been applied, there are many optimizations still possible.
- We introduce a new optimization approach for implementing efficient GPU-enabled library-based convolution operations, called *adaptive tiling*, which we also combine with loop unrolling.
- To the best of our knowledge, our implementation is the most optimized and best performing implementation of 2D Convolution in the spatial domain available to date.

We have made the source code of our kernels available from the first author's homepage as part of the data-parallel Parallel-Horus programming model.

The remainder of this paper is organized as follows. Section 2 discusses well-known optimization techniques that have to be applied to our CUDA kernels before we can apply our own optimization approach. Section 3 presents our approach for avoiding shared memory bank conflicts. Section 4 presents our new optimization approach called *adaptive tiling* and discusses the performance improvements. Section 5 combines adaptive tiling with loop unrolling to create our most efficient implementation. Section 6 evaluates the performance improvements of each optimization step on various graphics hardware. Section 7 discusses the limitations that are inherent to spatial solutions to the 2D convolution problem. Section 8 discusses related work and Section 9 and concludes.

2. Naive implementation and well-known optimizations

This section presents a naive CUDA implementation and discusses existing optimization techniques that form a starting point for our own optimizations. The discussion of these techniques is included to present the reader with a complete overview of the optimization process. Readers with much experience in GPU programming and optimization may choose to skip this section. As detailed in Section 8, the implementation approach presented also improves upon existing work.

In this paper, we continuously report performance results obtained on the Nvidia GTX680 Kepler GPU [5]. Whenever necessary, we also report results obtained on the GTX480 Fermi GPU [6] and the Tesla K20 [7], also of the Kepler architecture. The Kepler cards have significantly more compute cores than the Fermi cards, for example, 8 SMs of 192 cores (i.e. 1536 cores) for the GTX680 versus 15 SMs of 32 cores (i.e. 480 cores) for the GTX480. The Kepler cores run at a lower clock frequency to improve energy efficiency. The respective theoretical peak performance, computed as $cores \times frequency \times 2$, of the GTX480, GTX680, and K20 is 1344.96, 3090.43, and 3519.36 GFLOP/s. The theoretical peak global memory bandwidth, however, has not scaled up proportionally with the increased compute performance of the newer cards. The respective theoretical peak global memory bandwidth, computed as $(buswidth \times memoryclock)/8$, of the GTX480, GTX680, and K20 is 177, 192, and 208 GB/s. On the Kepler architecture global memory loads and stores are only cached in L2 and not in L1. The L1 cache is reserved for accesses to local memory and register spilling. On the Fermi architecture, however, global memory loads and stores are cached in L2 and L1. The caches give an important, yet very hard to predict, performance boost to the 2D convolution kernels. The Kepler SMs also have increased space for registers and can support a higher number of threads executing concurrently per SM. However, the amount of shared memory per SM on Kepler is exactly the same as on Fermi, 48KB per SM. While the GTX680 only has 8 SMs, the K20 has 13, and the GTX480 has 15, therefore, in total the older GTX480 has even more shared memory than either Kepler card.

In each of our measurements, the kernel performs a 2D convolution of an image of 4096×4096 floating point pixels and uses filter sizes ranging from 3 up to 43 in both dimensions. Using larger or smaller images influences the total execution time of the operation, but only has a very limited effect on the performance behavior of the kernel in terms of GFLOP/s. 3D graphs are used as the performance of our 2D convolution implementations often varies in both dimensions. Some configurations cause performance cliffs, that is a significant drop in performance occurs when the filter size is increased beyond a certain point.

In image processing, a convolution operation computes a new value for every pixel based on a weighted average of the original pixel and the pixels in its *neighborhood*. These weights are stored in a *convolution filter*, which also determines the size of the neighborhood. To ensure that every pixel can be evaluated (even at the edge of the image) we assume that the input image includes a border and is thus larger than the output image.

An implementation in C for the 2D convolution kernel, shown in Fig. 1(a), uses two loops to iterate over all pixels in the image. The inner two loops iterate over each pixel in the neighborhood of the current pixel and compute a weighted average using the weights stored in the convolution filter. The algorithm takes an image I of size $(I_w \times I_h)$ and a filter F of size $(F_w \times F_h)$ as arguments. A naive CUDA implementation, shown in Fig. 1(b), is obtained by creating one CUDA thread for each output pixel. This way, every CUDA thread computes the weighted average of a single pixel's neighborhood and writes a single pixel to the output image. The input and output images can be padded to multiples of the thread block width and height, to allow images of any size to be processed by the kernel.

The first step in the process of optimizing CUDA kernels is ensuring that the kernel is not global memory bandwidth bound. This can be easily checked using the Roofline Model [8]. The key idea behind the roofline model is to calculate the arithmetic intensity (FLOP/byte ratio) of a kernel and multiply this by the theoretical peak bandwidth of the device. The result is an estimate of the peak performance that can be achieved by the kernel. If this exceeds the theoretical peak performance of the device the kernel is clearly compute bound, otherwise it is memory bandwidth bound.

The arithmetic intensity of the 2D convolution kernel is calculated as follows. For every weight in the convolution filter, each thread loads 2 floating point values, the pixel and the filter weight making up a total of 8 bytes. These two inputs are multiplied and added to a local sum, giving an arithmetic intensity of 0.25 FLOP/byte. On a device with no hardware managed caches, the maximum compute performance of the kernel is computed by multiplying the memory bandwidth of the device with the arithmetic intensity of the kernel. However, on devices with hardware managed caches, many pixel values can be loaded from the cache as neighboring threads will require the overlapping pixel data.

Rather than relying on the hardware caches to cope with the high memory bandwidth requirements, parts of the data can be stored in different device memories. First of all, half of the loads

a

```
 1 for (y=0; y<I_h; y++) {
 2   for (x=0; x<I_w; x++) {
 3     sum = 0;
 4     for (j=0; j<F_h; j++) {
 5       for (i=0; i<F_w; i++) {
 6         sum += S[y+j][x+i] * F[j][i];
 7       }
 8     }
 9     I[y][x] = sum / (F_w * F_h);
10   }
11 }
```

b

```
 1 x = threadIdx.x+blockIdx.x*B_w;
 2 y = threadIdx.y+blockIdx.y*B_h;
 3 sum = 0;
 4 for (j=0; j<F_h; j++) {
 5   for (i=0; i<F_w; i++) {
 6     sum += S[y+j][x+i] * F[j][i];
 7   }
 8 }
 9 I[y][x] = sum / (F_w * F_h);
10
11
```

Fig. 1. (a) Pseudo code for a simple 2D convolution. (b) Pseudo code for the same algorithm as a CUDA kernel.

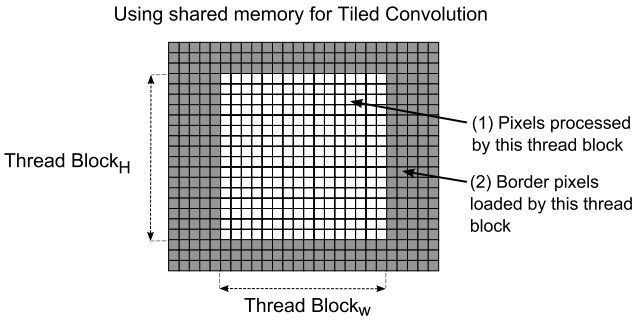

Fig. 2. Layout of the data stored in shared memory. Example shows a 16×16 thread block processing an 11×7 filter.

in the kernel are values from the convolution filter, of which all threads in a warp load the same value simultaneously. This access pattern is ideally suited for constant memory. Secondly, threads within a block can cooperate by sharing data through shared memory. As neighboring threads will require largely overlapping regions from the input image, the threads within a block may cooperatively load the entire area needed by all threads in the block, this approach is occasionally referred to as tiled convolution [9] and is illustrated in Fig. 2. This way, the threads load the area required by this thread block exactly once, instead of many times. For 2D convolution it is crucial to assign 2-dimensional workloads to thread blocks, in order to maximize data reuse. This is because threads that require mostly overlapping regions from the input image neighbor each other in 2 dimensions.

The exact bandwidth requirements of the kernel now depend on the convolution filter and thread block dimensions, and is given by $(F_w - 1 + B_w) \times (F_h - 1 + B_h) \times 4$ bytes per thread block, where F_w and F_h are the width and the height of the convolution filter and B_w and B_h are the width and the height of the thread block. The arithmetic intensity then becomes,

$$AI = \frac{2 \times F_w \times F_h \times B_w \times B_h}{(F_w - 1 + B_w) \times (F_h - 1 + B_h) \times 4}. \quad (1)$$

Given a thread block of 32×32 and the smallest possible 2D filter 3×3, the arithmetic intensity is 3.99 FLOP/byte. Given that the GTX680 has 192.26 GB/s of global memory bandwidth, a 2D convolution kernel that uses shared memory has a peak performance of 766.36 GFLOP/s, which is still lower than the theoretical peak of the device (3090.43 GFLOP/s). However, arithmetic intensity increases as the filter size increases. For most filter sizes in our test range, except for the 20 smallest, the theoretical peak of the kernel exceeds the theoretical peak of the device and therefore the kernel is compute bound rather than memory bandwidth bound for most filters. In Section 4, we will discuss how the arithmetic intensity, especially for the smallest filters, can be further increased using our adaptive tiling approach.

Fig. 3 shows the performance for the compute bound kernel that uses constant and shared memory. The performance first increases as the filter size gets larger. However, as the filter size increases the amount of shared memory used by the kernel also increases, reducing the performance as fewer thread blocks can run concurrently on each SM. In Fig. 3(a), the number of thread blocks that execute concurrently on each SM decreases from 8 to 3 as the filter size increases. This is visible in the graph as five distinct edges where performance decreases slightly. Between these edges performance slightly increases again, because larger filter sizes have a higher arithmetic intensity.

Fig. 3(a) also shows a few performance peaks, the largest at the high edge when the filter width is 33. This high edge occurs because the other filter sizes suffer from shared memory bank conflicts that occur when multiple threads try to access different values in the same bank. Our techniques for avoiding shared memory bank conflicts in 2D convolution kernels are discussed in Section 3. The smaller peaks occur at filter widths 5, 9, and 17, which execute with fewer bank conflicts than other filter widths, except for width 33. This is because multiple rows of border and pixel data add up to multiples of 32 banks, for these filter widths, and therefore some of the memory accesses will run conflict free. For more information on shared memory bank conflicts see the CUDA Programming Guide [10].

In the convolution kernels, discussed so far, each thread block processes a single tile of pixels from the input image. However, the number of tiles each thread block computes can be increased. The approach of processing multiple tiles in the horizontal dimension is often referred to as $1 \times N$ *tiling* [11] or thread-block merging [12]. Increasing the amount of work per thread eliminates redundant instructions such as array index calculations, loop accounting, and loading values from the convolution filter that were previously distributed across different threads, as shown in Fig. 4. Additionally, a thread block that computes two neighboring tiles from the input image no longer needs to load the overlapping borders between the two tiles, further increasing arithmetic intensity. The total amount of shared memory allocated to each thread block increases to $(F_w - 1 + B_w \times N) \times (F_h - 1 + B_h)$, when $1 \times N$ tiling is used. When the amount of work per thread is doubled, only half the number of thread blocks are created to execute the same kernel.

Fig. 4 shows how 1×2 tiling may be applied to the 2D convolution kernel. S is the dynamically allocated shared memory. B_w is the width of the thread block. The code on Line 5 is inserted to allow threads to compute two intermediate sums, while reusing the filter weight stored in F. Register usage increases by one register per thread to store the additional intermediate result. Tiling increases the amount of work per thread, which requires more registers per thread and more shared memory per thread block. Therefore, the tiling factor can only be increased within the resource limits of the device.

Fig. 5 shows the performance of the 1×2 and 1×4 tiled kernels on a GTX680 using a 32×32 thread block. Using large thread blocks reduces memory bandwidth consumption of the kernel, because with fewer thread blocks there are fewer overlapping neighborhoods between the tiles processed by each thread block. Because of the large number of threads in each thread block at most two thread blocks can execute on each SM at any given

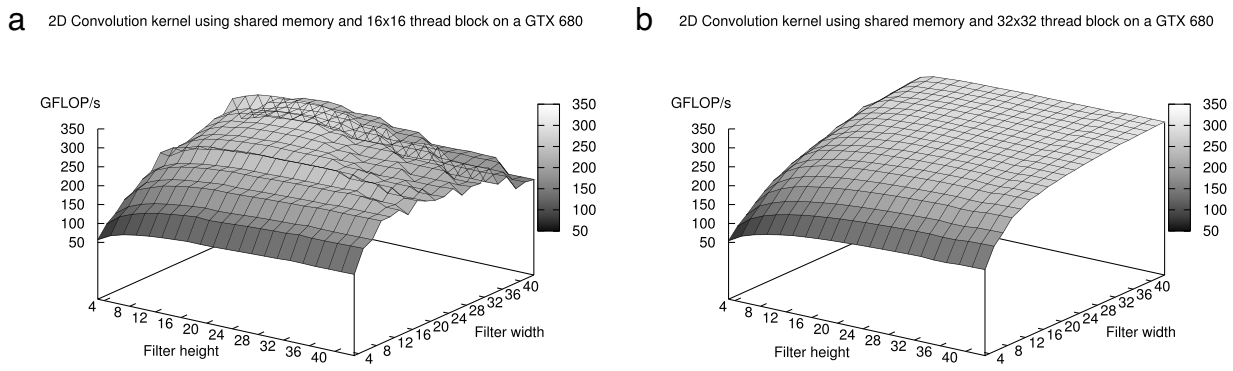

Fig. 3. Performance of the kernel using shared memory and (a) a 16 × 16 thread block size and (b) a 32 × 32 thread block size.

```
1 sum0 = 0, sum1 = 0;
2 for (j=0; j<F_h; j++) {
3   for (i=0; i<F_w; i++) {
4     sum0 += S[j*S_w+i] * F[j][i];
5     sum1 += S[j*S_w i+B_w] * F[j][i];
6   }
7 }
```

Fig. 4. Pseudo code for the main loop of the 1 × 2 tiled 2D convolution kernel.

time. However, as the filter size increases shared memory usage also increases, which limits the number of blocks that can execute concurrently on each SM. The edge in both performance graphs is caused by this exact drop in occupancy, from 64 warps (i.e. two 32 × 32 thread blocks) to 32 warps (i.e. a single 32 × 32 thread block). In fact, for the 1 × 4 tiled kernel, filter sizes (41 × 43) and (43 × 43) require more shared memory than available per SM and therefore cannot be executed.

Fig. 6 shows the performance of the 1 × 2 and 1 × 4 tiled kernels on a GTX680 using a 16 × 16 thread block. The performance in both figures decreases as the filter size increases, because increased shared memory usage decreases the number of thread blocks that can execute concurrently. Overall, the performance for both kernels is dominated by shared memory bank conflicts, except when the filter width is exactly 17. The next section explains why shared memory bank conflicts occur and how they can be avoided.

3. Avoiding shared memory bank conflicts

The previous section discussed an implementation of a 2D convolution kernel based on existing optimization techniques, such as 1 × *N* tiling. We showed that kernels with higher tiling factors often execute more efficiently, but may require more shared memory than available on the device when combined with a large thread block size (32 × 32). The shared memory

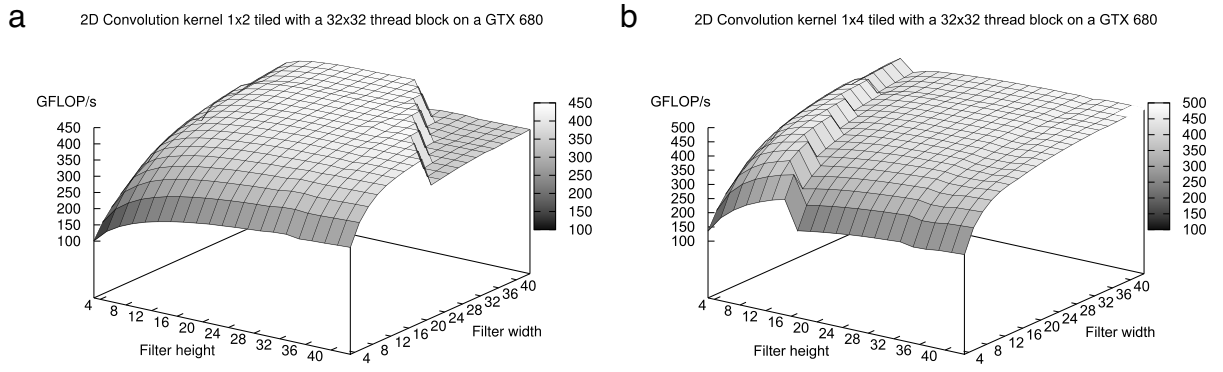

Fig. 5. Performance of (a) 1 × 2 and (b) 1 × 4 tiled kernels executed with 32 × 32 thread block.

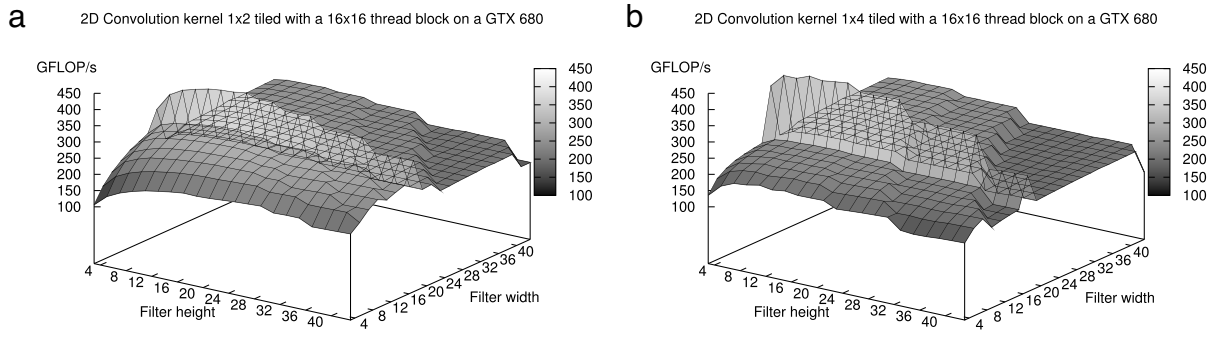

Fig. 6. Kernels executed with a 16 × 16 thread block, most filter widths cause bank conflicts, (a) using a 1 × 2 tiled kernel (b) using a 1 × 4 tiled kernel.

```
1   int ty = threadIdx.y;
2   int tx = threadIdx.x;
3
4   iPtr += blockIdx.y*B_h*I_w + (blockIdx.x*B_w);
5
6   int jEnd = F_h-1 + B_h;
7   int iEnd = F_w-1 + B_w;
8   for (j=ty; j<jEnd; j+= B_h) {
9       for (i=tx; i<iEnd; i+= B_w) {
10          shared_mem[j*S_w + i] = iPtr[j*I_w + i];
11      }
12  }
13  __syncthreads();
```

Fig. 7. Pseudo code that allows threads within a thread block to cooperatively load a rectangular area of size $(F_h - 1 + B_h) \times (F_w - 1 + B_w)$ and store it in shared memory.

requirements of the kernel can be reduced by reducing the thread block size to, for example 16×16. However, using smaller thread blocks may cause shared memory bank conflicts. Therefore, this section explains in detail why shared memory bank conflicts occur and presents a novel approach to avoid bank conflicts in convolution kernels.

To understand why bank conflicts can occur we need to understand how the threads access the shared memory data structure. The code for loading the data into shared memory is shown in Fig. 7. In 2D convolution, it is crucial to assign 2-dimensional workloads to thread blocks to maximize data reuse. In our approach, the threads within a thread block cooperatively load the area needed from the input image and store it in shared memory as shown in Fig. 7. The statement at Line 4 sets the input image pointer to the start of the area loaded by the thread block. Note, that this does not include individual thread indices yet. The for-loops on lines 9 and 10 ensure that each thread loads a different item and that the threads cooperatively load the required 2D area of pixels into shared memory.

In modern graphics hardware, any shared memory read or write of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module [10]. In CUDA, multi-dimensional arrays are wrapped row-wise across different shared memory banks. Whenever two threads from the same warp access different values in the same memory bank, a shared memory bank conflict occurs, causing the accesses to be serialized. Given the access pattern in Fig. 7, it depends on the thread block width and the convolution filter width whether or not a shared memory bank conflict will occur. Note that when the thread block width is equal to the number of memory banks, no shared memory bank conflicts occur as all threads in each warp access values in different banks, independent of the filter dimensions.

Fig. 8 illustrates how bank conflicts occur when the thread block width is 16 and the convolution filter width is 23. The light gray colored values represent the values that belong to the first row of pixels required by this thread block. The dark gray colored values belong to the second row; note that both rows include border pixels. In the top figure, bank conflicts occur as threads 6–15 and 16–25 access different values in the same memory banks. In the bottom figure, bank conflicts are avoided as threads 16–32 are directed towards different memory banks. This can be achieved by introducing a number of padding columns to the array stored in shared memory. Padding was introduced as a technique to solve shared memory bank conflicts in [13].

Our approach to avoiding bank conflicts is to extend the 2-dimensional array stored in shared memory with zero or more padding columns. The width of the array without padding is defined as $S_w = F_w - 1 + B_w$. The width of the padded array is obtained by increasing $F_w - 1$ to the first multiple of the number of memory banks, $S_w = \lceil \frac{F_w - 1}{M} \rceil \times M + B_w$, where M is the number of memory banks. The new width of the array is chosen such that each consecutive group of threads from the same row (i.e. with same threadIdx.y) within a single warp starts at the first unused memory bank. This guarantees that all threads within that warp access values in different memory banks, and thus guarantees conflict free access to shared memory. Note that when B_w is a multiple of the number of memory banks no padding is required, and when B_w is less than half the number of memory banks this approach may use more padding than strictly necessary. However, the advantage of this approach is that the same kernel code as listed in Fig. 7 may be used to obtain a bank conflict free implementation.

An alternative approach to avoiding bank conflicts may consider rearranging the order in which threads compute their local sum. The idea behind this approach is that threads may compute their local sum in any order, and as such, may direct themselves to different memory banks in order to avoid bank conflicts. There are two important drawbacks to this approach. First of all, when threads within a warp compute their local sum in different orderings, weights from the convolution filter can no longer be broadcast, increasing memory bandwidth consumption. Secondly, reordering the computations for threads within the same warp introduces a significant number of instructions for index calculations, increasing the execution time.

The performance of both kernels with and without padding is shown in Fig. 9, which clearly shows that the kernel with padding outperforms the one without padding in nearly all cases. For filter size 17×17 in Fig. 9(a) and (b) the performance of both kernels is exactly the same. This is because for this particular filter size all kernels run conflict-free and no padding is required. For filter size 41×41, the additional shared memory required for padding limits the occupancy and causes a drop in performance larger than the performance hit caused by bank conflicts. Fig. 9(b) shows that our approach for avoiding shared memory bank conflicts also applies to the GTX480, because cards of the Fermi architecture also have 32 memory banks. The performance across different filter sizes is less stable on the GTX680, compared to the GTX480, because drops in occupancy have a much larger effect on performance for the GTX680 [14]. This is because the Kepler cards have fewer, but larger SMs. Therefore, the SMs on Kepler are capable of executing more thread blocks concurrently. However, as the amount of shared

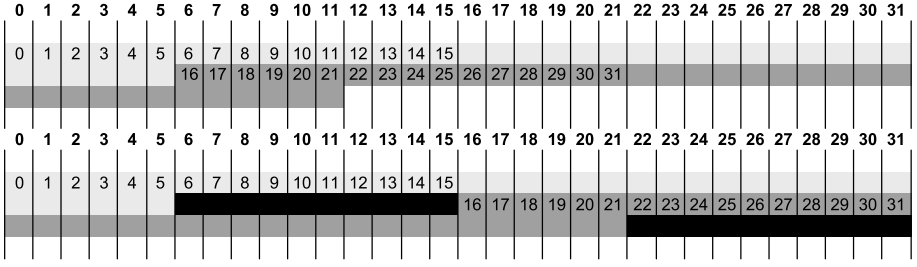

Fig. 8. Mappings between data and memory banks in shared memory, both without padding (top figure) and with padding (bottom figure).

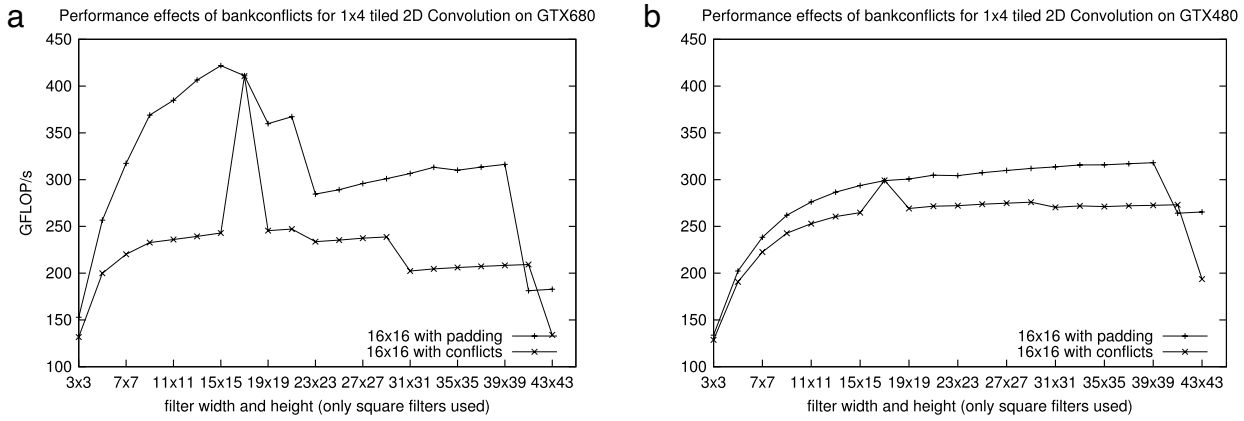

Fig. 9. Kernels executed with a 16 × 16 thread block. Padding is used to prevent bank conflicts. (a) shows execution on the GTX680 and (b) on the GTX480.

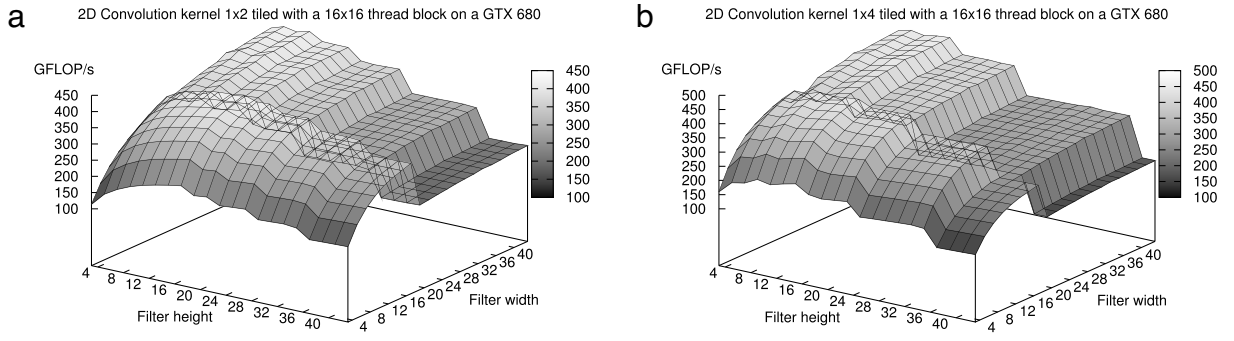

Fig. 10. Kernels executed with a 16 × 16 thread block. Padding is used to prevent bank conflicts. (a) using a 1 × 2 tiled kernel (b) using a 1 × 4 tiled kernel.

memory per SM is the same for both Kepler and Fermi SMs, only a smaller filter can be executed at higher occupancy. A more detailed analysis of the implications of both architectures on 2D convolution is given in Section 6.

Fig. 10 shows the performance for the full range of tested filter sizes using padding to prevent bank conflicts. Fig. 10 may be compared to Fig. 6 to see that padding improves the performance of the kernel for nearly all filter sizes. The performance drop that occurs right after the filter width is increased beyond 17 occurs because of the increase in shared memory use by the padding columns that avoid bank conflicts. Performance drops in the other direction, when the filter height is increased, also occur because the increase in shared memory usage causes a reduction in the occupancy.

Convolution operations with large filters have large overlapping borders between tiles, and thus require more shared memory per thread block. However, tiling further increases the use of shared memory. In fact, the 1 × 4 tiled kernel, shown in Fig. 5(b), requires more shared memory than available on the device when executed with filter sizes (41 × 43) and (43 × 43). This shows the very limited flexibility of the 1 × N tiling approach. Therefore, tiling must be kept to a minimum when the kernel is required to execute convolution operations with large filters. However, when only small filters are used, which is typical in many applications, a higher tiling factor may be applied to achieve more efficient execution on the GPU. The next section presents our *adaptive tiling* approach that addresses exactly this issue.

4. Adaptive tiling

For many CUDA kernels, such as the matrix multiplication kernel described in [11], there exists a one-size-fits-all best tiling factor that can be set at compile time and will create the most efficient kernel for any input size used at runtime. For convolution operations this approach is too restrictive as the efficiency of the kernel is dictated by the size and shape of the convolution filter. In this section, we present *Adaptive Tiling* as a new optimization approach for convolution operations on GPUs. With adaptive tiling the tiling factor is selected at runtime depending on the input data and the resource limitations of the device.

Adaptive tiling allows our convolution operations with relatively small filters to be executed with higher tiling factors and operations with relatively larger filters by kernels with lower tiling factors. This approach further increases the arithmetic intensity, especially for small convolution filters, which is given by

$$AI = \frac{2 \times F_w \times F_h \times B_w \times T \times B_h}{(F_w - 1 + B_w \times T) \times (F_h - 1 + B_h) \times 4} \quad (2)$$

where T is the tiling factor.

In fact, to select a tiling factor at run time means to select a particular implementation of a kernel to run. A small script is used to generate the variations of the kernel for a set of tiling factors before compilation. These kernels are all included in the library and can be selected by the run-time system. Instead of using a script, one might also use for example, C++ templates. Our adaptive tiling approach works independently of how the kernels are generated.

There are two main resource constraints that need to be considered when increasing the tiling factor: shared memory and the register file. First, before the tiling factor is increased, shared memory only contains the pixels processed by this thread block and their overlapping borders, using relatively little shared memory. More efficient execution can be achieved when the tiling factor is increased, at the cost of increasing the amount of shared

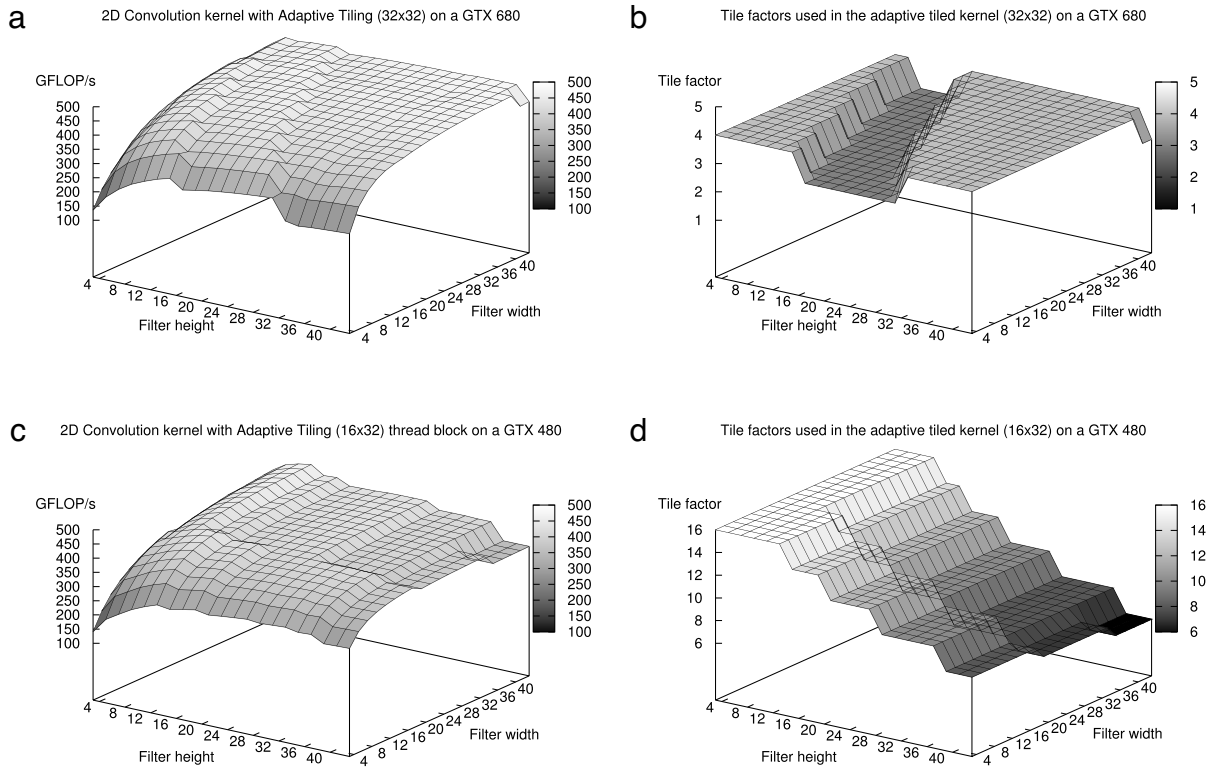

Fig. 11. (a) and (b) performance using the adaptive tiling optimization on GTX680 and GTX480. (c) and (d) the tiling factors selected by the run time system.

memory the kernel uses. Therefore, the tiling factor can only be increased as long as the tiles and the border around them still fit in shared memory.

Second, as the tiling factor increases, more registers are required per thread. Therefore, the tiling factor can only be increased up to the point where one thread block consumes the entire register file on an SM. However, the total number of registers used by a thread block may be reduced, by reducing the number of threads per thread block. This slightly increases the use of global memory bandwidth as more thread blocks will be required to complete the computation. However, if the decrease in thread block size allows a further increase in the tile factor, the total number of thread blocks does not necessarily increase. This suggests that it might be worthwhile to lower the number of threads per thread block to allow for higher tiling factors.

Using a compile-time fixed thread block size, the highest possible tiling factor is selected at run time based on the largest possible fit within the shared memory available on the device and the convolution filter size of the operation at hand. Note that this means each SM can execute no more than one thread block simultaneously, because this single thread block will consume almost all of the available shared memory. While this approach yields a very flexible and efficient implementation on the Fermi, the selection scheme needs some fine tuning to be efficient for the Kepler cards.

The SMs of the Kepler architecture need roughly twice as much parallelism per SM, compared to Fermi SMs [14]. For 2D convolution, increased parallelism per SM can only be achieved by increasing occupancy, which means either creating larger thread blocks or allowing multiple thread blocks to execute concurrently on each SM. Both approaches for increasing occupancy imply a significant increase in the required amount of shared memory, while the Kepler SMs do not have more shared memory than Fermi SMs. That is why the run-time selection scheme on the Kepler chooses the highest possible tiling factor such that two thread blocks can still run concurrently on each SM for small filter sizes. For larger filter sizes it is still efficient to simply select the highest possible tiling factor.

Fig. 11 shows the achieved performance for the 2D convolution kernel that uses the adaptive tiling optimization for the best performing thread block sizes. The filter dimensions are used to determine the resource requirements of the kernel, which are then used to select the best tiling factor within the resource limitations of the device present at runtime. The edges in the performance graphs (a) and (b) are caused by changes in the tiling factor. Figures (c) and (d) show the actual tile factors that were selected by the run-time system to create the corresponding performance graphs shown in (a) and (b).

The adaptive tiling approach for the GTX680 uses a 32×32 thread block size, similar to Fig. 5. Compared to Fig. 5(b) adaptive tiling improves the efficiency of 121 out of the 441 tested filter sizes, for the other filter sizes the same tiling factor is selected and therefore the performance is the same. Additionally, the adaptive tiling implementation is able to execute filter sizes 41×43 and 43×43, because the run-time system decreases the tiling factor if the resource requirements exceed that of the device. The relatively large thread block size used on the GTX680 limits the range of possible tiling factors, but is necessary to supply sufficient parallelism to the Kepler SMs. However, the fixed thread block size of 32×32 is not the best performing for each filter size within our test range. Therefore, our parameter sweep discussed in the next section, also uses different thread block sizes for operations with different filter sizes.

5. Adaptive tiling combined with loop unrolling

One important optimization that we have not considered until now is loop unrolling. Loop unrolling introduces trade off between flexibility and efficiency, as the number of iterations for all loops of a given kernel have to be known at compile time. This means that for each possible filter size we create a unique CUDA kernel and optimize it individually. Instead of selecting the highest possible tiling factor that still fits in shared memory, the run time system now selects the CUDA kernel with the best performing

combination of block size and tiling factor given a particular filter size. We have implemented this using a lookup table which holds this information for each possible filter size in the expected range of filter sizes.

This approach has a number of drawbacks: First of all, the number of CUDA kernels becomes quite large, which also increases compilation times. Secondly, the range of filter sizes that may be used by the applications have to be known at the time the library is compiled. However, it is important to know exactly how much performance can be gained from unrolling each kernel. Although creating an optimized library using this optimization is considerably more work, it is still worthwhile considering that many researchers from the multimedia domain can benefit from this effort.

We have generated the unrolled versions of our CUDA kernels for each possible filter size ranging from 3 to 43 in both dimensions. At the time the kernel is generated, the thread block width and height as well as the tiling factor have to be known, as these determine the number of iterations of each unrolled loop. For each filter size, a search is performed through a range of selected thread block dimensions and tiling factors. The selected ranges for thread block dimensions are powers of two from 16 to 64 in the x-dimension and 4 to 64 in the y-dimension. The tiling factors range from 1 to 10. In this search, we have tested the performance of 110 (11 block sizes $\times$ 10 tiling factors) unrolled kernels generated specifically for each of the 441 evaluated filter sizes. The results are stored in a lookup table, which for each filter size stores a reference to the best performing unrolled kernel as well as the execution parameters required at launch time, such as the tiling factor and thread block dimensions.

Whenever the host code of the library is required to execute a 2D convolution operation on the GPU at run time, it uses the lookup table to select the best performing implementation that should be used based on the filter dimensions of the operation at hand. The lookup table also specifies the tiling factor and thread block dimensions, which are used to compute several execution parameters, such as the amount of dynamically allocated shared memory, possibly including padding, and the total number of thread blocks required to complete execution of the kernel.

The lookup table, which is unfortunately too large to be included in this text, essentially describes several of the properties of the underlying device and could potentially be used to guide the optimization process of other kernels. For 2D convolution, thread block size 16 $\times$ 32 was the best performing thread block size for 49.7% of the tested filter sizes. In fact, 342 out of the 441 best performing kernels use a thread block width of 16. This means our technique for avoiding shared memory bank conflicts, as described in Section 3, is used in 77.6% of the best performing 2D convolution kernels. While using a thread block width of 16 increases resource usage in terms of shared memory, and possibly reduces the occupancy, the fact that the thread block size is small, allows for higher tiling factors. High tiling factors significantly reduce the number of redundant instructions previously spread across different threads, which has a large impact on the performance of compute-bound kernels. The average tiling factor used in the 441 best performing kernels is 4.16, tiling factor 2 is used for 21.3% of the filter sizes, factor 4 in 33.6%, and tiling factors larger than 4 are used in 34.2% of the kernels. This confirms the limitations of using a compile-time fixed tiling factor and shows why adaptive tiling is so important to convolution operations.

Fig. 12 shows the achieved performance for the 2D convolution kernel that uses a specifically generated CUDA kernel for each filter size. In Fig. 12, the best performing kernel is the one that performs a convolution with filter size 17 $\times$ 43 and achieves about 938.4 GFLOP/s, which uses a thread block size of 16 $\times$ 64 and tiling factor 6. More importantly, the smallest and most commonly used 2D filter size (3 $\times$ 3) improved from 135.9 with adaptive tiling to 251.3 GFLOP/s when combining adaptive tiling with unrolling.

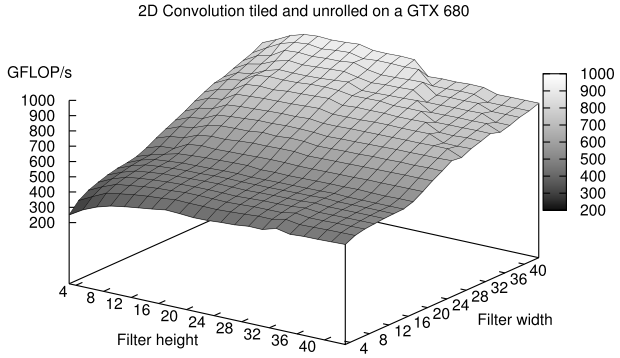

Fig. 12. Performance for 2D convolution using a uniquely generated tiled and unrolled kernel for each filter size.

The optimization approach presented here can also be applied to other operations from the image processing domain, including neighborhood operations, such as erosion, dilation, separable convolution, and anisotropic Gaussian filtering. Separable convolution, for example, considers convolution operations with 2-dimensional filters that may be split in two 1-dimensional filters, which are applied consecutively to obtain the same result as when performing a full 2D convolution. The separable convolution operation has considerably lower complexity, but unfortunately cannot be used for all 2D convolution filters. We have implemented separable convolution by implementing and optimizing the horizontal and vertical 1-dimensional filtering operations as two separate operations and applied the adaptive tiling combined with unrolling optimization to both operations.

The tested thread block sizes range from 16 to 64 in the x-dimension and 4 to 64 in the y-dimension. The tested range of tiling factors is from 1 to 20. Note that tiling for the vertical filtering operation is performed vertically, instead of in the x-direction as described in Section 2. This is necessary for increasing data reuse within thread blocks, where horizontal tiling would only increase the number of border pixels each thread block loads, without actually increasing data reuse. Because of the 1D filters, the separable convolution operation uses considerably less shared memory than 2D convolution. Therefore, it is possible to allow for much higher tiling factors.

The resulting lookup table for separable convolution is considerably smaller than the lookup table for 2D convolution and is therefore shown in Table 1. The lookup table again provides insights that could guide future optimization searches for similar kernels. For the horizontal filtering operation we observe thread block size 32 $\times$ 4 was the most efficient for the 10 smallest filters. For larger filters, thread block size 16 $\times$ 8 created the most efficient kernels when combined with tiling factors of 6 or higher. In 11 out of the 21 filter sizes a thread block width of 16 is used, which means the kernel also uses our technique for avoiding shared memory bank conflicts as explained in Section 3. However, for the vertical 1D-filtering operation bank conflicts cannot occur, because no border pixels are loaded in the x-direction. For the vertical filtering, thread block size 32 $\times$ 4 is the most efficient for 18 out of 21 filters using tiling factors ranging from 3 to 8. Fig. 13 shows the performance of the kernels from Table 1 for both horizontal and vertical filtering.

6. Evaluation

In this section, we present a short evaluation of the performance effects of each individual optimization step for the GTX680. The performance results of each optimization step are also given for the

Table 1
Lookup table for horizontal (left) and vertical (right) filtering as used in separable convolution on the GTX680.

F_w	Tiling factor	Block size	F_h	Tiling factor	Block size
3	4	32 × 4	3	5	64 × 4
5	4	32 × 4	5	4	32 × 4
7	5	32 × 4	7	4	32 × 4
9	4	32 × 4	9	3	32 × 4
11	4	32 × 4	11	4	32 × 4
13	8	32 × 4	13	3	32 × 4
15	8	32 × 4	15	6	32 × 8
17	8	32 × 4	17	6	32 × 8
19	8	32 × 4	19	7	32 × 8
21	8	32 × 4	21	7	32 × 8
23	7	16 × 8	23	6	32 × 8
25	7	16 × 8	25	5	32 × 8
27	7	16 × 8	27	6	32 × 8
29	7	16 × 8	29	6	32 × 8
31	7	16 × 8	31	8	32 × 8
33	7	16 × 8	33	4	32 × 8
35	6	16 × 8	35	5	32 × 8
37	6	16 × 8	37	5	32 × 8
39	6	16 × 8	39	7	32 × 8
41	8	16 × 8	41	8	16 × 8
43	8	16 × 8	43	6	16 × 8

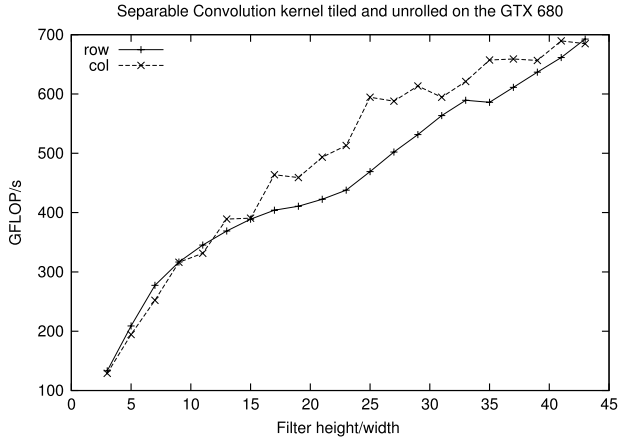

Fig. 13. Performance of separable convolution on the GTX680 using adaptive tiling combined with unrolling.

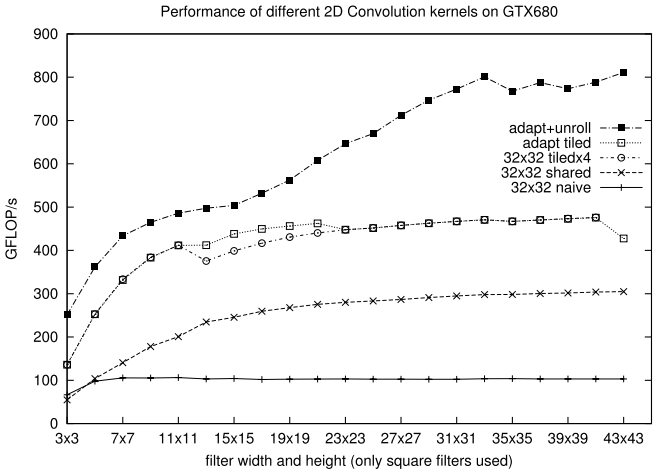

Fig. 14. Performance for several different implementations of the 2D convolution problem on the GTX680.

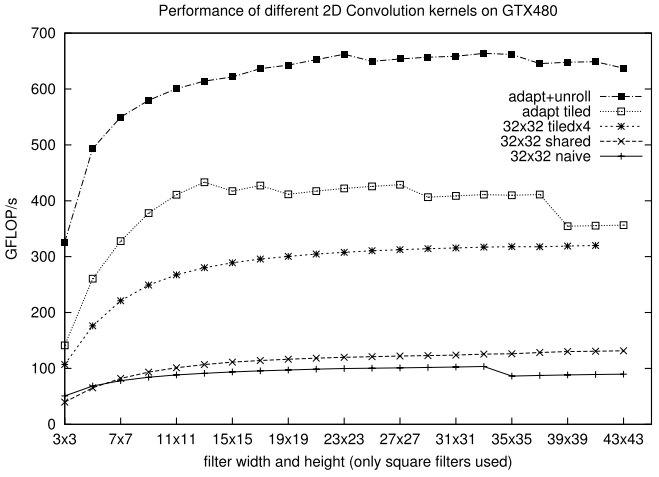

Fig. 15. Performance for several different implementations of the 2D convolution problem on the GTX480.

GTX480 and Tesla K20 GPUs to show the effects on different architectures. To simplify the discussion we only show the performance results for the square filter sizes in our test range, i.e. 3×3, 5×5, etc.

Fig. 14 shows the performance of each optimization step discussed in this paper on the GTX680. The performance of the naive kernel implementation is limited by the memory bandwidth and does not exceed 106.3 GFLOP/s. Fig. 14 clearly shows that each optimization step we apply further improves performance. The adaptive tiling optimization again uses a fixed thread block size of 32×32 and dynamically chooses the tiling factor at runtime. The performance of the adaptive tiling and the statically 1×4 tiled implementation is similar, because the adaptive tiling selects tiling factor 4 for the majority of the square filters on the GTX680. Note that for filter size 43×43 the statically 1×4 tiled implementation cannot execute, as it requires more shared memory than available per SM, while the adaptive tiling algorithm simply selects to execute the 43×43 filter with tiling factor 3. Our final optimization step demonstrates a performance improvement of up to a factor of 9.14 over the naive implementation. This is not only due to the unrolling optimization, but also because the best performing combination of thread block size and tiling factor is selected for each individual filter. Note that unrolling the loops is only possible when the number of iterations of all loops is known at compile time. This means that an individual implementation is created for each filter size.

The performance effects of each optimization step on the GTX480 is shown in Fig. 15. The performance of the naive implementation on the GTX480 is quite similar to the naive performance on the GTX680. For filter sizes 3×3 and 5×5, the naive implementation outperforms the implementation that explicitly uses shared memory. This is because the naive approach benefits from the fact that global memory loads are cached in L1 on the GTX480. The shared memory implementation also benefits from caching in L1, albeit with the introduction of a small overhead for additional instructions and synchronization. However, using shared memory is very important to achieving high performance, for larger filter sizes or when the tiling factor is increased. For filter size 43×43, the tiled 1×4 implementation cannot execute, because it requires more memory than available on the device. The adaptive tiling implementation does not suffer from this issue, because it simply selects the implementation with the largest possible tiling factor that still fits in shared memory. The adaptive tiling implementation uses a thread block size of 16×32, which means it also uses our approach for avoiding shared memory bank conflicts as explained in Section 3. The adaptive tiling combined with unrolling implementation again improves significantly over the other implementations.

The GTX680 (Fig. 14) outperforms the GTX480 (Fig. 15) for most filter sizes. However, the difference is less than what one would expect from the theoretical peak performance of both devices.

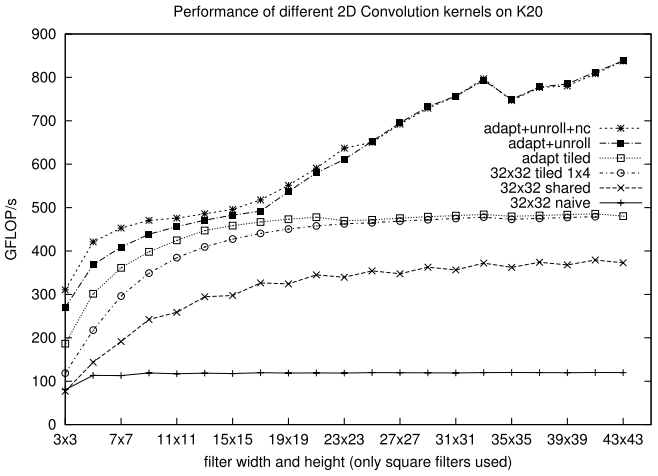

Fig. 16. Performance for several different implementations of the 2D convolution problem on the K20.

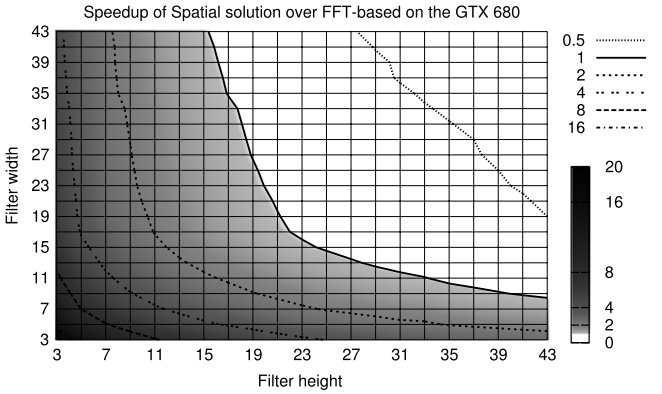

Fig. 17. Speedup of spatial solution over the FFT-based approach. Filter sizes in the gray area are executed faster with our implementation than with the FFT-based implementation.

There are several architectural differences between Kepler and Fermi that limit the performance of 2D convolution operations on Kepler cards. The most important one for 2D convolution is that Kepler has 6 times less shared memory per compute core in each SM compared to Fermi. Therefore, it becomes much harder to keep all 192 cores in an SM occupied. When there is not enough shared memory, it is not possible to either increase the thread block size or to increase the number of thread blocks that run simultaneously on each SM. This causes low occupancy of each SM and therefore low utilization of the 192 compute cores, as well as lower utilization of global memory bandwidth and higher dependence on long latency operations. All involved factors explain why performance is expected to be further from the theoretical peak.

Fig. 16 shows the performance of each optimization step on the K20. The performance of the naive implementation on the K20 is about 24% faster than the naive performance on the GTX480 and GTX680. It is important to note that Fig. 16 again shows that each optimization step we apply further improves performance.

The K20 also has 48KB of read-only cache at L1 that may be used by global memory loads for read-only data [7]. This requires that the global memory loads are explicitly enclosed by the `__ldg()` intrinsic. The compiler transforms these loads into non-coherent loads that pass through the read-only cache. In Fig. 16, our most efficient implementation (adapt+unroll+nc) uses this specific optimization. The fact that the performance of many of the kernels in Fig. 16 improves by this optimization also indicates that the performance is related to the achieved global memory bandwidth.

Note that Fig. 16 shows only a very small subset of the filters that were tested and that performance varies with the filter size in both dimensions. For example, for filter size 17×43 our adaptive tiling combined with unrolling optimization executes at just over a teraflop. Finally, the adaptive tiling combined with unrolling optimization, both with and without using the read-only cache, demonstrates a performance improvement of up to a factor 8.3 over the naive implementation.

We observe that the tiling factors used in the best performing kernels on the GTX680 and the K20 are generally lower than on the GTX480. However, selecting the tiling factor at run time is even more important on the GTX680 and the K20, than on the GTX480. While the tiling factors for the best performing kernels range from 2 to 10 on the GTX480 and from 1 to 8 on the K20, tiling factor 4 is the most common for both cards accounting for 44.2% on the GTX480 and only for 24.9% on the K20.

The lookup tables for both the GTX680 and the K20 also demonstrate that selecting the thread block size at run time is still very important. We observe that the thread block sizes used in the best performing kernels on the Kepler cards are on average 56.5% larger than on the GTX480. This confirms that larger thread blocks are required to achieve sufficient occupancy on the Kepler cards. Thread block size 16×64 is the most common among the best performing kernels on the GTX680 and K20, accounting for 49.7% and 36.5%. For both Kepler cards, the vast majority of the best performing kernels use thread block width of 16, 83.2% on the GTX680 and 92.1% on the K20. Therefore, for most filter sizes in our test range, the best performing kernel has a thread block width that requires the use of our approach to avoid shared memory bank conflicts, as described in Section 3.

7. Limitations

A 2D convolution operation in geometric space can also be implemented as a point-wise multiplication in frequency space. Nvidia's Whitepaper [15] explains their implementation of 2D convolution based on the CUFFT library. The advantage of this method is that it has lower complexity for very large filters. This presents a limitation that is inherent to spatial solutions to the 2D convolution problem. We therefore present a short comparison of both approaches.

The FFT approach has some minor limitations in the fact that it uses more memory than a spatial solution. The input image needs to be rounded up to the nearest power of two if the padded dimension is less than or equal to 1024, or to the nearest multiple of 512 otherwise. Then the filter is wrapped and stored in another image of the same size, roughly doubling the amount of GPU memory required for the operation. However, due to the lower complexity the FFT-based approach will become increasingly faster than any spatial solution as the filter size increases. The question is which approach is fastest for which range of filter sizes.

We have tested both our implementation and Nvidia's implementation for the FFT-based approach [15] that uses the CUFFT library, on the GTX680 given an image size of 4096×4096 for a filter range of 3 to 43 elements in both dimensions. The contour plot, shown in Fig. 17, shows the relative speedup of our approach over the FFT approach. For the most commonly used filter sizes, 3×3 and 5×5, the speedup of our approach over the FFT-based approach is 18.96 and 10.09, respectively. Operations with filter sizes below and left of the '1' line, the gray area in the figure, execute faster with our implementation than with the FFT-based implementation. However, for very large filter sizes the FFT-based approach is faster, simply because of the lower complexity of the algorithm.

It should be noted that the use of very large filters is rare in MMCA applications. It is however also possible to implement a

library routine for 2D convolution such that a run time choice is made between our implementation and an FFT-based implementation based on the filter size of the operation at hand. We consider this as future work.

8. Related work

The work presented in this paper extends earlier work on implementing and optimizing convolution operations for the GPU. Very early work on implementing convolution operations on GPUs used shader programs in graphics APIs [16,17], which can support only limited filter sizes due to the limited number of instructions per pixel. This section discusses more recent work which is all based on CUDA. We also present performance results to compare our method of using shared memory for tiled convolution against a different method suggested by Hwu [9]. Finally, we compare our optimized kernels with kernels created by the auto-optimizing source-to-source compiler by Yang et al. [12].

OpenCV [18] is an open source library for Computer Vision and contains several implementations for many image processing operations. OpenCV has a special module named GPU operations, which also contains various implementations for convolution operations. The 2D convolution operations discussed in this paper correspond with the filter2D method in OpenCV. OpenCV's current implementation of 2D convolution for GPUs uses CUDA and is limited to 2D filters with a total size not greater than 256 elements. The only optimization that is applied to their CUDA implementation is the use of constant memory to store the convolution filter. No shared memory or other optimizations are used.

Russo et al. [19] compare various implementations of separable convolution on GPUs using CUDA, on FPGAs using Verilog and on CPUs using C and Matlab. The optimizations that are applied to their CUDA implementation consist of using shared memory and static $1 \times N$ tiling. They conclude that CUDA on GPUs outperforms their FPGA and CPU implementations.

Hartung et al. [20] compare various implementations of 2D convolution using CUDA on GPUs and ANSI-C and OpenMP on CPUs. They refer to the implementation of 2D convolution in the spatial domain as applying a spatially-varying kernel. The optimizations they apply to their CUDA implementation are using shared memory and static $1 \times N$ tiling. They conclude that their CUDA implementation outperforms their various CPU implementations.

Nugteren et al. [21] introduce a classification of algorithmic skeleton implementations for image processing, which enable GPU code generation and mapping of the algorithm to the GPU hardware. 2D convolution is discussed as an example of the skeleton class for neighborhood-to-pixel image processing operations. Local data reuse is exploited through the use of shared memory. The thread block size is increased to the maximum allowed by the hardware to minimize the amount of border pixels shared between thread blocks. No further optimizations to increase arithmetic intensity or instruction efficiency, such as $1 \times N$ tiling, are applied.

Al Umairy et al. [22] attempt to improve the performance of Nvidia's FFT-based implementation of 2D convolution. They optimize towards a specific use case within the domain of electromagnetic diffraction modeling. While they argue that many applications use small convolutions, it is unclear if this refers to small input data or small convolution filters. Their conclusion is that the CUFFT library should be extended with an operation that directly performs a complete 2D convolution operation, as opposed to using separate forward and inverse FFT transformations. Their solution is not compared to an implementation in the spatial domain.

Dastgeer et al. [23] have developed two performance metrics based on either occupancy or tiling factor for our Adaptive Tiling optimization, based on our earlier work [24]. These metrics help to guide the decision making process in selecting the tiling factor at run time. Their experimental results confirm that maximizing the tiling factor rather than occupancy results in the best performance for GPUs of the Fermi architecture and older cards. As we have shown in Section 4, GPUs of the Kepler architecture require a more balanced approach. In this paper, we also show that even more efficient implementations can be obtained by combining adaptive tiling with unrolling and choosing the best performing combination of thread block size and tiling factor for each filter size.

The 2D convolution problem is well known in the context of GPU Computing and is often used as example to illustrate or teach how constant and shared memory may be used to reduce the global memory bandwidth consumption of a kernel [25,9,26]. However, these descriptions of the optimization process do not go beyond the discussion of well-known techniques presented in Section 2 and only consider a static 5×5 filter size.

In contrast to our approach of using shared memory (explained in Section 2), Hwu [9] suggests keeping the number of elements in shared memory equal to the number of threads in the block. This simplifies the loading process, as each thread loads only one element from the input image. As a consequence, not all threads can be active during the computation as the threads near the edge of the tile do not have enough information to complete their computation. There are two important drawbacks to this approach. First, as some threads are not participating in the computation, more thread blocks must be created to complete the computation. This limits the opportunity for data reuse within the thread block and increases overall global memory bandwidth consumption. Secondly, this approach does not scale with increasing filter sizes. For the 5×5 filter size, threads in the last four rows and columns do not participate in the computation. As the filter size is increased, the number of active threads within each thread block drops dramatically. More importantly, while the opportunity for data reuse increases with larger filter sizes, data reuse in this approach actually decreases as there are less active threads in each thread block to reuse data.

The question that remains is which approach executes more efficiently considering only the 5×5 $2D$ convolution filter. We have implemented both approaches to using shared memory and unrolled both kernels since the filter size is known at compile time. We have tested on GTX480 with an image of 4096×4096 and a static filter of 5×5. Our implementation of Hwu's approach achieves 180.8 GFLOP/s, while our approach of having some threads load multiple elements and keeping all threads active during computation achieves 229.8 GFLOP/s.

We have also compared our optimized kernels with kernels created by the auto-optimizing source-to-source compiler by Yang et al. [12]. Their GPGPU compiler takes a slightly modified naive kernel as input and outputs an optimized kernel. The optimization strategy that the source-to-source compiler applies to the 2D convolution kernel is primarily focused on using shared memory to convert noncoalesced global memory accesses into coalesced memory accesses. To this end, the compiler assumes that the number of loop iterations is always a multiple of 16. This results in a kernel where the data required for the first 16 iterations of the innermost loop is loaded into shared memory by all threads in the thread block, following the design that some threads load multiple elements. This guarantees coalesced memory accesses, as well as data reuse within each block of 16 iterations for the innermost loop.

Unfortunately, many opportunities for data reuse are unexploited. After the first 16 iterations the data in shared memory is overwritten with the data required for the next 16 iterations which are again loaded from global memory, although the elements could have been reused from shared memory as well. In contrast, our implementation tries to maximize data reuse in both dimensions and

for all iterations, instead of only within each block of 16 iterations in the horizontal dimension. However, we do believe that the optimization techniques presented in this paper could be integrated into an auto-optimizing compiler such as the GPGPU Compiler.

To confirm that our approach yields a more efficient kernel, we have tested the performance of a 2D convolution kernel created for a static filter size of 32×32. On the GTX480 with an input image of 4096×4096, the optimized kernel created by the GPGPU compiler for a 32×32 filtering operation takes 143.7 ms, achieving 239.0 GFLOP/s. Our optimized kernel took 45.8 ms for the same operation, achieving 749.6 GFLOP/s.

9. Conclusions and future work

Convolution operations are an essential tool in signal and image processing and are typically responsible for a large fraction of the application's execution time. In this paper, we have discussed the implementation and optimization of convolution operations for the domain of multimedia content analysis (MMCA). As part of our work on transparent parallelization tools for the MMCA domain, our goal is to obtain an efficient library based implementation for convolution operations.

We have evaluated the performance effects of many different implementations for the 2D convolution operation and separable convolution on the GTX680, GTX480 and Tesla K20 graphics cards. We have introduced a new approach for solving shared memory bank conflicts in the context of convolution operations as well as an optimization approach, called *adaptive tiling*, for implementing efficient, yet flexible, convolution operations on modern GPUs. We have also presented an implementation that combines adaptive tiling with loop unrolling to obtain a less flexible, yet highly efficient implementation. The approach is less flexible, as the filter sizes used by the application at runtime have to be known at compile time. While the approach presented in this paper focuses on 2D convolution, the techniques also apply to separable convolution.

All optimizations combined do have a very large impact on performance. More importantly, each optimization step we introduce provides a significant performance improvement for each tested GPU. For the GTX680, the most commonly used 2D filter size in image processing (3×3) executes at 66.6 GFLOP/s using the first naive implementation, 133.2 GFLOP/s when adaptive tiling is used, and finally 251.3 GFLOP/s using tiling combined with loop unrolling. The best performing kernel on the GTX680 is the one that performs a convolution with filter size 17×43 and achieves 938.4 GFLOP/s, using a thread block size of 16×64 and tiling factor 6. GPUs of the Kepler architecture have much less shared memory and less global memory bandwidth per compute core compared to GPUs of the Fermi architecture. As a result, the compute performance of 2D convolution operations does not come as close to the theoretical peak performance as on the GTX480. For example, our best performing kernel on the K20 executes a 2D convolution operation with filter size 17×43 and achieves about 1003 GFLOP/s, which is only at 28.5% of the theoretical peak performance.

The lookup table stores the results of a search through a set of generated unrolled CUDA kernels and stores which thread block size and which tiling factor was used in the creation of the best performing kernel for each filter size. The resulting table essentially describes some of the properties of the underlying device and could potentially be used to guide the optimization process of other kernels. The lookup table also demonstrates the importance of our approach for solving shared memory bank conflicts, which enabled us to also use thread blocks with a width of 16 threads in an efficient manner. For 2D Convolution on the GTX480, 76.2% of the best performing kernels use a thread block width of 16 threads. For the GTX680 and the K20, even 83.2% and 92.1% of the best performing kernels use a thread block width of 16 and therefore require padding to avoid bank conflicts. While padding increases resource usage in terms of shared memory, and possibly reduces the occupancy, the fact that the thread block size is small, allows for higher tiling factors. High tiling factors significantly reduce the number of redundant instructions previously spread across different threads, which has a large impact on the performance of compute-bound kernels.

Our evaluations show the limitations of implementing convolution operations with a compile time fixed tiling factor. The lookup tables for 2D convolution and separable convolution demonstrate the importance of our adaptive tiling approach. For example in 2D convolution on the GTX480, tiling factor 2 is used for 24.0% of the filter sizes, factor 4 in 44.2%, and tiling factors larger than 4 in 31.8%. Selecting the tiling factor at run time is even more important on the GTX680 and the K20. While the tiling factors for the best performing kernels range from 2 to 10 on the GTX480 and from 1 to 8 on the K20, tiling factor 4 is the most common for both cards accounting for 44.2% on the GTX480 and only for 24.9% on the K20. This again shows why our adaptive tiling approach is so important to optimizing convolution operations. Our evaluation also shows that selecting the thread block size at run time is also very important for high performance. We observe that the thread block sizes used in the best performing kernels on the Kepler cards are on average 56.5% larger than on the GTX480. This confirms that larger thread blocks are required to achieve sufficient occupancy on the Kepler cards.

FFT-based implementations of the 2D convolution operation have a lower complexity for convolution operations with very large convolution filters, compared to spatial implementations such as the ones discussed in this paper. Therefore, for very large 2D convolution filters an FFT-based implementation will outperform any spatial solution to the 2D convolution problem. However, for the most commonly used 2D filter sizes, 3×3 and 5×5, the speedup of our approach over Nvidia's FFT-based approach on the GTX680 is a factor 18.96 and 10.09, respectively.

The lookup tables used for selecting the most efficient tiled and unrolled kernel implementation are tightly linked to a specific GPU architecture. Therefore, future work is directed towards developing performance models, as constructing the lookup table currently requires a considerable amount of performance testing.

We have made the source code of our kernels available from the first author's homepage as part of the data-parallel Parallel-Horus programming model. The lookup tables for all tested GPUs and our performance measurement data is also made available on the web. See http://www.cs.vu.nl/~ben/mp/Convolution2D_data/.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This publication was supported by the Dutch national program COMMIT.

References

[1] C. Snoek, M. Worring, J. Geusebroek, D. Koelma, F. Seinstra, A. Smeulders, The semantic pathfinder: using an authoring metaphor for generic multimedia indexing, IEEE Transactions on Pattern Analysis and Machine Intelligence 28 (10) (2006) 1678–1689.
[2] A. Galizia, D. D'Agostino, A. Clematis, A grid framework to enable parallel and concurrent TMA image analysis, International Journal of Grid and Utility Computing 1 (3) (2009) 261–271.
[3] A. Plaza, et al., Commodity cluster-based parallel processing of hyperspectral imagery, Journal of Parallel Distributed and Computing 66 (3) (2006) 345–358.
[4] F. Seinstra, J. Geusebroek, D. Koelma, C. Snoek, M. Worring, A. Smeulders, High-performance distributed image and video content analysis with parallel-horus, IEEE Multimedia 14 (4) (2007) 64–75.
[5] Nvidia, GeForce GTX 680, NVIDIA Corporation Whitepaper, 2012.

[6] Nvidia, Fermi Compute Architecture, NVIDIA Corporation Whitepaper, 2010.
[7] Nvidia, Kepler GK110 Architecture, NVIDIA Corporation Whitepaper, 2012.
[8] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Communications of the ACM 52 (4) (2009) 65–76.
[9] W. Hwu, Lecture notes, 2011. http://courses.engr.illinois.edu/ece408/fall2011/ece408_syll.html.
[10] Nvidia, CUDA Programming Guide, 2013. http://docs.nvidia.com/cuda/.
[11] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, W. Hwu, Program optimization space pruning for a multithreaded GPU, in: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM, 2008, pp. 195–204.
[12] Y. Yang, P. Xiang, J. Kong, H. Zhou, A GPGPU compiler for memory optimization and parallelism management, in: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2010, pp. 86–97.
[13] G. Ruetsch, P. Micikevicius, Optimizing matrix transpose in cuda, in: NVIDIA CUDA SDK Application Note, 2009.
[14] Nvidia, Kepler Tuning Guide, NVIDIA Corporation Whitepaper, 2012.
[15] V. Podlozhnyuk, FFT-based 2D Convolution, NVIDIA Corporation Whitepaper, 2007.
[16] B. Payne, S.O. Belkasim, G. Owen, M.C. Weeks, Y. Zhu, Accelerated 2D image processing on GPUs, Computational Science–ICCS 2005 (2005) 256–264.
[17] O. Fialka, M. Cadik, FFT and convolution performance in image filtering on GPU, in: Tenth International Conference on Information Visualization, IEEE, 2006, pp. 609–614.
[18] Open Source Computer Vision, OpenCV 2.4.5, 2013.http://www.opencv.org/.
[19] L.M. Russo, E.C. Pedrino, E. Kato, V.O. Roda, Image convolution processing: a GPU versus FPGA comparison, in: 2012 VIII Southern Conference on Programmable Logic, SPL, IEEE, 2012, pp. 1–6.
[20] S. Hartung, H. Shukla, J.P. Miller, C. Pennypacker, GPU acceleration of image convolution using spatially-varying kernel, in: 2012 19th IEEE International Conference on Image Processing, ICIP, IEEE, 2012, pp. 1685–1688.
[21] C. Nugteren, H. Corporaal, B. Mesman, Skeleton-based automatic parallelization of image processing algorithms for GPUs, in: 2011 International Conference on Embedded Computer Systems, SAMOS, IEEE, 2011, pp. 25–32.
[22] S.A. Al Umairy, A.S. Van Amesfoort, I.D. Setija, M.C. Van Beurden, H.J. Sips, On the use of small 2d convolutions on GPUs, in: Computer Architecture, Springer, 2012, pp. 52–64.
[23] U. Dastgeer, C. Kessler, A performance-portable generic component for 2d convolution computations on GPU-based systems, in: Proc. MULTIPROG-2012 Workshop at HiPEAC-2012, Paris, 2012, pp. 1–12.
[24] B. van Werkhoven, J. Maassen, F. Seinstra, Optimizing convolution operations in cuda with adaptive tiling, in: Second Workshop on Applications for Multi and Many Core Processors, A4MMC at ISCA 2011, San Jose, California, 2011, pp. 1–12.
[25] J. Stam, Convolution Soup: A case study in CUDA optimization, 2009. http://www.nvidia.com/content/GTC/documents/1412_GTC09.pdf.
[26] W. Hwu, Convolution Lab, 2011. http://courses.engr.illinois.edu/ece408/fall2011/MPs/MP3-README.txt.

Ben van Werkhoven is a PhD student in the Department of Computer Science at VU University Amsterdam, the Netherlands. His current research interests include high-performance distributed computing on large collections of multi- and many-core processors with the main focus on developing efficient high-level programming models for such platforms.

Jason Maassen is an eScience engineer at the Netherlands eScience Center and currently works on the eSalsa climate modelling project.

Henri E. Bal is a full professor in the Department of Computer Science, where he heads the High Performance Distributed Computing research group, at VU University Amsterdam, the Netherlands.

Frank J. Seinstra is a senior eScience Engineer and Executive Board member at the Netherlands eScience Center, where he coordinates the research and development of innovative eScience solutions in multidisciplinary research projects.