Optimizing Convolution Operations in CUDA with Adaptive Tiling

Ben van Werkhoven, Jason Maassen, and Frank J. Seinstra

Department of Computer Science, VU University
De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands
{ben,jason,fjseins}@cs.vu.nl

Abstract. Convolution operations are essential to signal and image processing applications and are typically responsible for a large fraction of the execution time. Existing approaches for optimizing convolution operations that support a wide range of filter sizes are too limited. In this paper, we present an optimization approach, called *adaptive tiling*, to implement a highly efficient, yet flexible, convolution operation for modern GPUs. We evaluate the performance of each optimization step on the GTX 480 graphics card and show that adaptive tiling improves performance by 34% on average over differently optimized kernels. 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

The massive amount of data in today's signal and image processing applications makes storing, cataloging, processing, and retrieving of information a challenging task. 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. The execution time of convolution operations can be reduced dramatically, as individual parts of the input data can be processed independently. Graphics Processing Units (GPUs) have many small processing elements and, therefore, are a well-suited computing platform for performing convolutions.

The performance of a CUDA kernel can vary greatly depending on how its resource requirements correspond to the resources available on the device. As such, optimizing CUDA kernels is not a simple or straightforward process, as a slight increase in resource usage could result in a dramatic reduction in achieved performance. Additionally, the underlying hardware and threading model contain hard usage restrictions that make the optimization space discontinuous [1].

Ryoo et al. [2] introduced 1xN *tiling* as an optimization for a matrix multiplication kernel in CUDA. The key idea is to view the input data as a collection of tiles, each processed by different thread blocks. When 1xN tiling is used, the amount of work per thread block is increased such that each thread block computes N tiles, and thus the total number of thread blocks required to execute the kernel is reduced by a factor of N. This approach drastically reduces the number

of redundant instructions, such as array index calculations, loading constants, and loop accounting that were previously distributed across different threads.

In kernels such as the matrix multiplication kernel described in [2], 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. However, 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.

This paper follows a library-based approach for implementing convolution operations. This means that no assumptions are made about the input data, such as the size of the convolution filter. Some applications of convolution, like *combinatorial thresholding* [3, 4] used for radio frequency interference filtering in radio astronomy, determine the size of the convolution filter *at runtime* based on the statistical properties of the input data. As such, it is crucial that the implementation can achieve high-performance without sacrificing flexibility.

The need for flexibility means that 1xN tiling can not be used for optimizing our convolution kernels. However, based on the resource limitations of the device and the size of the input data it is possible to select the correct tiling factor at runtime. We refer to this approach as *adaptive tiling*, and demonstrate that it achieves high-performance, without sacrificing flexibility. This paper provides the following contributions:

- We present adaptive tiling, a new optimization approach for implementing optimized GPU-enabled library functions.
- Performance tests on the GTX 480 graphics card show that adaptive tiling improves the performance of our kernels by 34% over optimized kernels.
- We discuss the performance effects of many different optimization steps for our 2D convolution kernel on the GTX 480 graphics card.
- Preliminary results indicate that adaptive tiling outperforms Nvidia's FFT-based convolution for the range of most commonly used filter sizes in image processing.

Our work extends previous work on implementing and optimizing convolution operations for the GPU. Many existing implementations require that the convolution filter has a fixed size that is known at compile time [5, 6] or consider only separable filters [7, 8]. Other implementations that use shader programs in graphics APIs are limited by the number of instructions allowed per processed pixel [9, 10]. While the examples given in this paper focus on 2D convolution, all techniques equally well apply to separable convolution. To the best of our knowledge, our implementation is the most optimized and best performing implementation in the spatial domain available to date.

This paper is organized as follows. Section 2 provides an introduction to CUDA. Section 3 presents the starting point implementation of the 2D convolution operation in CUDA. Section 4 describes the hardware used in the performance measurements. Section 5 discusses techniques for reducing memory bandwidth consumption. Section 6 shows how 1xN tiling may be used to improve the performance of the kernel for a static range of filter sizes. Section 7 presents the adaptive tiling optimization and discusses the performance improvement. Section 8 discusses future work and concludes.

2 Introduction to CUDA

In CUDA [1], a system consists of a *host* (the CPU), and one or more *devices*, which are massively parallel processors. The host can move application data between host and device memory, and invoke operations (called *kernels*) that execute on the device. The kernels exhibit a large amount of data parallelism, allowing arithmetic operations to be performed on different parts of the data simultaneously. Kernels are executed by a very large number of threads concurrently. As CUDA threads are much more lightweight than CPU threads, kernels can be executed by millions of threads in parallel [1].

Threads executing the same kernel are organized in a two-level hierarchy, i.e. a *grid* of *thread blocks*. Each thread block contains at most 512 or 1024 threads depending on the device. Built-in `threadIdx` and `blockIdx` variables provide the grid and thread block index to each thread, and are used to divide the work among the threads. Threads in the same thread block are executed by the same *streaming multiprocessor* (SM) and are able to synchronize with each other. Continuous sections of 32 threads in a thread block are grouped into *warps*, in which all threads execute the same instruction in parallel. When threads in a warp take different execution paths, multiple passes with suppression of certain threads are required to complete execution.

CUDA supports different memory types. *Global memory* is a large high-latency memory that is typically used to store large input and output data structures. Although global memory bandwidth can be very high (177 GB/s on a GTX480), it generally limits the performance of CUDA kernels if no other memory types are utilized. Due to the design of modern DRAMs, the peak global memory bandwidth can only be achieved if memory accesses are *coalesced*: all threads in a warp access consecutive global memory locations.

Constant memory is a small read-only memory, supporting low-latency, high bandwidth access when all threads simultaneously access the same location. *Shared memory* is an on-chip memory that can be allocated to thread blocks and accessed at very high speed in a highly parallel manner. As all threads in a thread block can read and write the shared memory, it is an efficient way for threads to share their input data and intermediate results.

The number of thread blocks that can execute on each SM in parallel is called the *occupancy*, and depends on how the SM resources are partitioned among thread blocks. The most important resources are the amount of registers and shared memory used by each thread block. The partitioning of SM resources often results in subtle interactions between resource limits, as a slight increase in resource usage could result in a dramatic reduction in the achieved performance.

3 A First Naive Implementation

This section discusses the straightforward translation of a sequential C implementation for the 2D convolution operation into a CUDA kernel. As such a straightforward translation is generally inefficient, we refer to this as a *naive*

```
for (y=0; y<I_h; y++) {
  for (x=0; x<I_w; x++) {
    sum = 0;
    for (j=0; j<F_h; j++) {
      for (i=0; i<F_w; i++) {
        sum += S[y+j][x+i] * F[j][i];
      }
    }
    I[y][x] = sum / (F_w * F_h);
  }
}
```

(a)

```
x = threadIdx.x+blockIdx.x*BLOCK_SIZE;
y = threadIdx.y+blockIdx.y*BLOCK_SIZE;
sum = 0;
for (j=0; j<F_h; j++) {
  for (i=0; i<F_w; i++) {
    sum += S[y+j][x+i] * F[j][i];
  }
}
I[y][x] = sum / (F_w * F_h);
```

(b)

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

CUDA kernel. The translation for a separable convolution operation to CUDA proceeds similarly.

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 data structure called a *convolution filter*, the size of which determines the size of the neighborhood. To ensure that every pixel can be evaluated (even at the edge of the image) a slightly larger copy of the image (a *scratch image*) is created. This scratch image is large enough to contain extra border pixels, which can be filled in various ways, for example, using a constant value or by mirroring the edges of the image. With the addition of these border pixels every pixel in the original image can now be evaluated.

The C implementation for the 2D convolution kernel, shown in Figure 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.

The 2D convolution operation has a high degree of data parallelism and can easily be written as a simple CUDA kernel by unrolling the outer two loops and letting every CUDA thread compute a single iteration. In other words, the outer two loops are replaced with an index calculation that determines which pixel each thread needs to work on (using `threadIdx` and `blockIdx`, see Figure 1(b)). This way, every CUDA thread computes the weighted average of a single pixel's neighborhood and writes a single pixel to the output image. Thus, thread blocks are responsible for a group or *tile* of pixels in the output image. The input and output images can be padded to a multiple of the thread block size, to allow images of any size to be processed by the kernel.

4 Hardware Environment and Methodology

In the remainder of this paper we optimize the abovementioned naive kernel in a number of steps. Since we are interested in the effect of each optimization step, we report the achieved performance immediately after discussing each optimization step. We measured the performance of our kernels on the GTX 480 Fermi GPU, which has 1.6 GBs of global memory and a global memory bandwidth of 177

GB/s. The GTX480 has 15 multiprocessors each with 32 CUDA cores, 48 KB of shared memory, and up to 1024 threads per thread block. GPUs of the Fermi architecture have a L1/L2 cache hierarchy, which makes it very hard to predict performance. Therefore, we once use an older GPU, the Tesla C1060, to illustrate the difference in performance behavior.

In each measurement, the kernel performs a 2D convolution of a 4096x4096 floats image and filter dimensions ranging from 7 up to 43. Using different image sizes only has a very limited effect on the performance behavior of the kernel in terms of GFLOP/s.

5 Constant Memory, Shared Memory, and Bank Conflicts

When optimizing CUDA kernels, it is important to realize that other optimizations are irrelevant as long as the compute kernels are global memory bandwidth-bound [1, 2, 11, 12]. Global memory bandwidth consumption of a kernel is easily computed using the following formula:

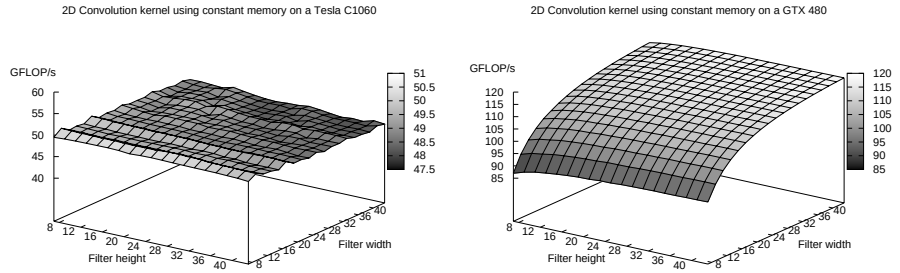
$$bandwidth = SPs \times loads \times bytes \times f, \tag{1}$$

where *SPs* represents the number of stream processors, *loads* the fraction of instructions that load a value from global memory in the main loop of a given kernel, *bytes* the number of bytes per load from global memory, and f the clock frequency of the stream processors.

When the bandwidth required by a kernel exceeds the bandwidth supported by the device, the kernel performance will be limited. For example, the naive kernel, shown in Figure 1(b), has 8 instructions in the main loop of the kernel, 2 of which are global memory loads each loading 4 bytes. Using Equation 1, we can calculate that on a Tesla C1060 GPU with 240 stream processors at 1.30 GHz, the required bandwidth is 312 GB/s, where the device has a maximum bandwidth of only 102 GB/s.

We can also calculate the maximum compute performance the kernel can obtain. For every 2 flops (the multiply and add in the inner loop of Figure 1(b)), 8 bytes must be loaded. As the maximum bandwidth of a Tesla C1060 is 102 GB/s, the kernel performance is limited to (102 / 8) * 2 = 25.5 GFLOP/s. GPUs of the Fermi architecture [13], such as the GTX480 or Tesla C2050, have an L1/L2 cache hierarchy and therefore some values may be fetched from the cache, which yields slightly better performance for larger filters and makes it much more difficult to predict the performance ceiling of a given kernel.

To improve the performance of our naive kernel, part of the data can be stored in different device memories. For example, half of the global memory loads in 2D convolution are values from the convolution filter (which all threads in a warp load simultaneously). This access pattern is ideally suited for constant memory. Storing the convolution filter in constant memory reduces the global memory bandwidth consumption and improves the performance by a factor of 2. On a Tesla C1060, that is 156 GB/s and 51 GFLOP/s as shown in Figure 2(a). The performance on the Tesla C1060 (Figure 2(a)) does not depend on the filter size,



(a) (b)

Fig. 2: Performance of the 2D convolution kernel using constant memory on (a) a cacheless Tesla C1060 (b) the L1/L2 cached GTX 480 GPU.

because the kernel is still memory bandwidth-bound. On the GTX 480 shows quite different performance behavior, as shown in Figure 2(b). The performance of the kernel on the GTX 480 increases for wider filters, because more values can be reused from the L1 cache.

Additionally, threads in a single thread block can share their neighborhoods by storing them in shared memory. Since these threads process adjacent pixels, their neighborhoods largely overlap. Loading the neighborhood of an entire thread block into shared memory only once can save a very large number of redundant memory reads. Since the size of the neighborhood is application dependent, shared memory must be allocated dynamically.

For example, in the naive kernel, each thread loads $F_w \times F_h$ pixels. With a 23×23 filter size and 16×16 thread block this would lead to 135.424 pixels loaded per thread block. When the neighborhood of an entire thread block is loaded only once, $(16 + 23 - 1) \times (16 + 23 - 1) = 1444$ pixels loads are required, a reduction by a factor 93.8. Hence, with the use of shared memory, our kernel is no longer limited by the global memory bandwidth of the device. Instead, the kernel is now compute-bound and its performance depends on the size of the convolution filter as can be seen in Figure 3.

On Fermi cards, such as the GTX 480, the performance improvement of using shared memory as a software managed cache is often limited, because of the presence of hardware managed L2 and L1 caches. The achieved performance shown in Figure 3 shows very little improvement compared to Figure 2(b). In fact, for very small filter sizes there is a slight decrease in performance, because of the synchronization overhead that is involved with using shared memory. However, the advantage of using shared memory is that the kernel becomes suitable for applying further optimization techniques.

The high edge at filter width 33 in Figure 3(a) is caused by the absence of bank conflicts. The GTX 480 has 32 memory banks. Data structures in shared memory are wrapped row-wise around those banks, that is consecutive pixel will be stored in different banks. A bank conflict occurs when two or more threads access different values in the same memory bank. For example, when a thread block has only 16 threads in the x-direction, care must be taken that the first 16 threads in each warp access values in banks different from those accessed by the last 16 threads.

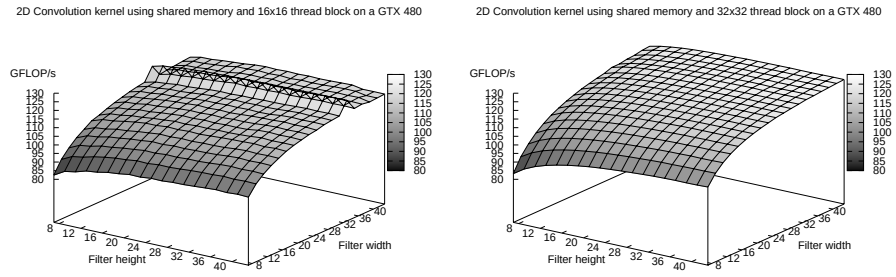


Fig. 3: Performance of the 2D convolution kernel with shared memory using (a) a 16x16 thread block size causing bank conflicts, except when width=33 and (b) a 32x32 thread block size.

When the convolution filter is 33 elements wide, each row of the data needed by a thread block contains exactly 16 left border pixels, followed by the 16 pixels that are actually processed by this thread block, followed by another 16 right border pixels. In shared memory the first 32 pixels are stored across the 32 different banks, the next 16 right border pixels will be stored in the same banks as the first 16 pixels. With only 16 threads in the x-direction, the first 16 threads in a warp access pixels from a different row in the input data, than the last 16 threads. Therefore, the first 16 threads will always load values from other banks than the last 16 threads and the kernel is free of bank conflicts. For all other filter widths shown in Figure 3(a) bank conflicts occur and reduce performance.

Memory bank conflicts can be prevented in two ways. First, through the use of *padding*, meaning that each row of the data needed by the thread block is extended in the x-direction, such that all threads in the warp will access different memory banks. Second, the thread block size may be adjusted to contain exactly 32 threads in the x-direction. For example, when a thread block size of 32x32 is used, each warp contains 32 threads in the x-direction, who always access values in subsequent memory banks. Therefore, no bank conflicts occur as shown in Figure 3(b).

6 Tiling

In the convolution kernels, discusses 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, this approach is often referred to as 1xN tiling [1, 2] or thread-block merge [6]. 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 Figure 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 reducing the memory bandwidth consumption. The total amount of shared memory allocated to each thread block increases to $(F_w - 1 + TB_w \times N) \times (F_h - 1 + TB_h)$, when 1xN tiling

```
sum0 = 0, sum1 = 0;
for (j=0; j<F_h; j++) {
  for (i=0; i<F_w; i++) {
    sum0 += S[j*S_w+i] * F[j][i];
    sum1 += S[j*S_w+i+TB_w] * F[j][i];
  }
}
```

Fig. 4: Pseudo code for the main loop of the 1x2 tiled 2D convolution kernel. S is the dynamically allocated shared memory, TB_w is the width of the thread block.

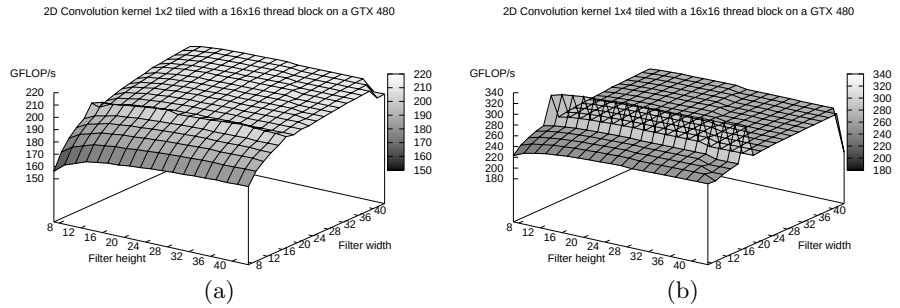
is used. When the amount work of per thread is doubled, only half the number of thread blocks are created to execute the same kernel.

Figure 4 shows how 1x2 tiling may be applied to the 2D convolution kernel. Register usage increases by one register per thread to store intermediate results. 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.

Figure 5 shows the achieved performance of the 1x2 and 1x4 tiled kernels on a GTX 480. As a thread block size of 16x16 is used, memory bank conflicts can occur and have a very large impact on performance. The 1x4 tiled kernel (Figure 5(b)) executes more efficiently then the 1x2 tiled (Figure 5(a)), except when the filter size is exactly 43x43. This is because of a drop in the occupancy. Because of the 1x4 tiling, the kernel uses more than half of the shared memory available on the SM. This means that only a single thread block can be executed by each SM simultaneously.

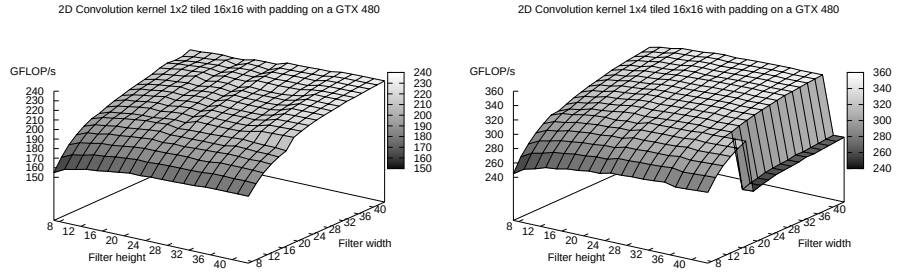
Figure 6 shows the performance achieved by the 1x2 and 1x4 tiled kernels that use padding to prevent memory bank conflicts. Again the 1x4 tiled kernel executes more efficiently, but the number of filters that execute with a lower occupancy has increased. This is caused by padding, which increases the amount of shared memory used by each thread block for most filter sizes. From Figures 5 and 6, we can conclude that padding improves the performance of the kernel for most filter sizes.

Figure 7 shows that memory bank conflicts can also be avoided by using a 32x32 thread block size. Using larger thread blocks further reduces memory bandwidth consumption of the kernel as there are fewer overlapping neighborhoods between the tiles processed by different thread blocks. However, the



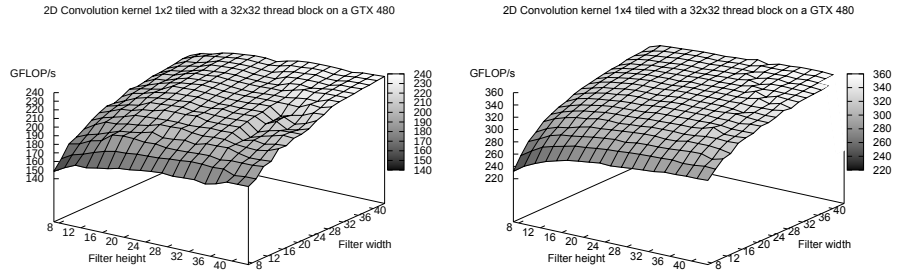
(a) (b)

Fig. 5: Kernels executed with 16x16 as thread block dimensions, most filters widths cause bank conflicts. (a) using a 1x2 tiled kernel (b) using a 1x4 tiled kernel.



(a) (b)

Fig. 6: Kernels executed with 16x16 as thread block dimensions. Padding is used to prevent bank conflicts. (a) using a 1x2 tiled kernel (b) using a 1x4 tiled kernel.



(a) (b)

Fig. 7: Kernels executed with 32x32 as thread block dimensions. (a) using a 1x2 tiled kernel (b) using a 1x4 tiled kernel, which was unable to execute for the (41x43) and (43x43) filters, because of shared memory limitations.

amount of shared memory used by each thread block also increases, which reduces the occupancy. Despite the different execution parameters, such as occupancy and memory bandwidth, the performance behavior is shown in both Figures 6 and 7 is quite similar.

Convolution operations with large filter sizes have large overlapping borders between different tiles, and thus require more shared memory per thread block. Tiling further increases the use of shared memory. In fact, the 1x4 tiled kernel, shown in Figure 7(b), requires more shared memory than available on the device when executed with filter sizes (41x43) and (43x43). This shows the very limited flexibility of the 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, a high tiling factor may be applied to achieve more efficient execution on the GPU.

7 Adaptive Tiling

For many CUDA kernels, including the matrix multiplication kernel described in [2], 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. However,

it is possible to select the tiling factor at runtime depending on the input data and the resource limitations of the device. We refer to this approach as *Adaptive Tiling*. 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.

There are two main resource constraints that need to be considered when increasing the tile factor: shared memory and the register file. First, a relatively small filter implies relatively small overlapping borders between tiles, and therefore, relatively little use of shared memory. More efficient execution is achieved when the tiling factor is increased. Increasing the tiling factor increases the amount of shared 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. Reducing the number of threads per thread block slightly increases in 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 further increase in the tile factor, the total number of thread blocks does not necessarily increase. This suggests that it is worthwhile to lower the number of threads per thread block to allow for more and higher tiling factors.

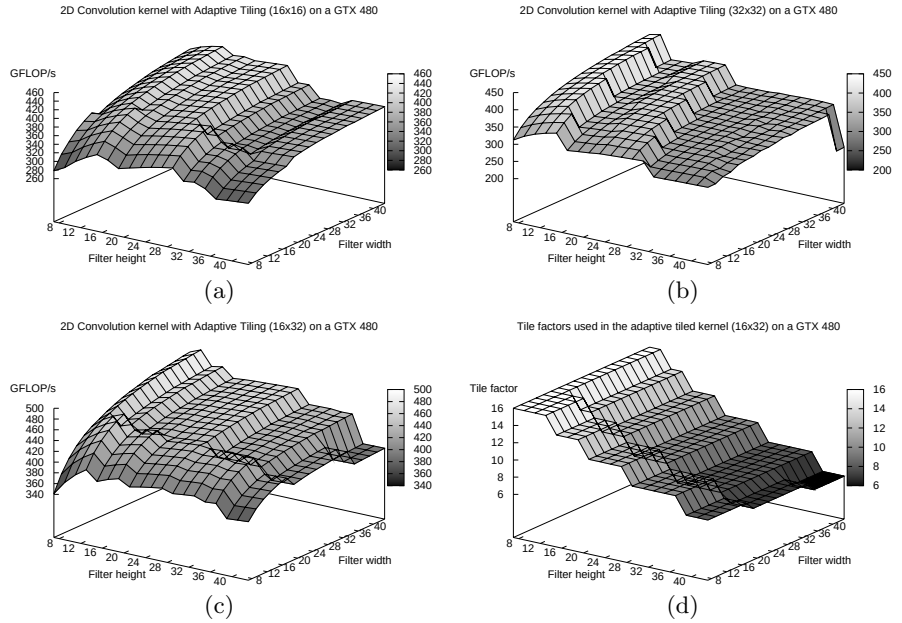


Fig. 8: (a), (b), and (c) the achieved performance for the 2D convolution kernel using the adaptive tiling optimization with three different thread block sizes on the GTX 480. (d) the tile factors applied for a thread block size of 16x32 (c).

There are a large number of possible thread block sizes. We have performed an exhaustive search through the set of reasonable thread block sizes, which revealed 16x32 as the thread block size that yields the highest performance.

While the tiling factor is determined *at runtime*, the thread block size is set *at compile time* instead. When the thread block size is set at runtime, register usage further increases with N registers, where N is the tiling factor. Setting the thread block size at compile time allows for higher tiling factors, and thus improves performance.

Figure 8 shows the achieved performance for the 2D convolution kernel that uses the adaptive tiling optimization for three different thread block sizes. The filter dimensions are used to determine the resource requirements of the kernel, which are then used to select the highest possible tiling factor within the resource limitations of the device present at runtime. The edges in the performance graphs (a), (b), and (c) are caused by changes in either the occupancy or the tiling factor. Figure 8(d) shows the actual tile factors that are used by the kernel with a thread block size of 16x32 on a GTX 480.

The adaptive tiling approach improves the performance of our 2D convolution kernel by, on average, a factor of 1.34 compared to performance achieved by the 1x4 tiled kernel, shown in Figure 7(b), and a factor of 4.73 compared to the naive implementation. As shown in Figure 8(c), the performance of the 2D convolution kernel reaches up to 500 GFLOP/s, which is at 37% of the theoretical peak of the GTX 480.

Nvidia's whitepaper [14] describes a different approach for implementing 2D convolution operations through the use of the Fourier Transform and the CUFFT library. The approach applies an FFT to both the image and the filter, followed by a point-wise multiplication of the two results, then the inverse FFT is applied to the result of the multiplication. Preliminary results indicate that our approach outperforms the FFT-based convolution for the range of most commonly used filter sizes (up to 17x17 for square filters). Specifically, for the most commonly used filter size (3x3) our implementation is 10 times faster. Additionally, the adaptive tiling approach may be extended to use FFT-based convolutions for very large filters.

8 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 proposed an optimization approach, called *adaptive tiling*, for implementing highly efficient, yet flexible, convolution operations on modern GPUs. We have evaluated the performance of many different implementations for convolution kernels on the GTX 480 graphics card. While this paper focusses on 2D convolution, all techniques equally well apply to separable convolution.

Results have shown that the adaptive tiling optimization improves performance over the fixed tiling approach for all filter sizes. Specifically, adaptive tiling improves performance by 34% on average over other optimized implemen-

tations on the GTX 480. Because of its flexible nature, adaptive tiling is a very suitable approach for implementing optimized library routines that are able to efficiently execute on the GPU. In our future work, we aim to apply the adaptive tiling approach to optimize kernels from different domains as well.

Future work will present a more detailed comparison of the performance of the 2D convolution implementation based on adaptive tiling and the FFT-based approach.

References

1. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands-on Approach. 1 edn. Morgan Kaufmann (February 2010)
2. Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Stratton, J., Hwu, W.: 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) 195–204
3. Offringa, A., De Bruyn, A., Biehl, M., Zaroubi, S., Bernardi, G., Pandey, V.: Post-correlation radio frequency interference classification methods. Monthly Notices of the Royal Astronomical Society **405**(1) (2010) 155–167
4. Offringa, A., de Bruyn, A., Zaroubi, S., Biehl, M.: A LOFAR RFI detection pipeline and its first results (2010)
5. Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proceedings of the 37th annual international symposium on Computer architecture, ACM (2010) 451–460
6. Yang, Y., Xiang, P., Kong, J., Zhou, H.: 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) 86–97
7. Podlozhnyuk, V.: Image convolution with CUDA. NVIDIA Corporation white paper, June (2007)
8. Sinha, S., Frahm, J., Pollefeys, M., Genc, Y.: GPU-based video feature tracking and matching. In: EDGE, Workshop on Edge Computing Using New Commodity Architectures. Volume 278., Citeseer (2006)
9. Payne, B., O. Belkasim, S., Owen, G., C. Weeks, M., Zhu, Y.: Accelerated 2D image processing on GPUs. Computational Science–ICCS 2005 (2005) 256–264
10. Fialka, O., Cadik, M.: FFT and convolution performance in image filtering on GPU. In: Information Visualization, 2006. IV 2006. Tenth International Conference on, IEEE (2006) 609–614
11. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Hwu, W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ACM (2008) 73–82
12. Ueng, S., Lathara, M., Baghsorkhi, S., Hwu, W.: CUDA-lite: Reducing GPU programming complexity. Languages and Compilers for Parallel Computing (2008) 1–15
13. N. Leischner, V. Osipov, and P. Sanders: Fermi Compute Architecture White Paper (2009)
14. Podlozhnyuk, V.: FFT-based 2D convolution. NVIDIA white paper (2007)