Usman Dastgeer, Christoph Kessler. A performance-portable generic component for 2D convolution computations on GPU-based systems. Proc. MULTIPROG-2012 Workshop at HiPEAC-2012, Paris, Jan. 2012.

A performance-portable generic component for 2D convolution computations on GPU-based systems

Usman Dastgeer and Christoph Kessler

PELAB, Dept. of Computer and Information Science,
Linköping University, Sweden
`{usman.dastgeer,chrisotph.kessler}@liu.se`

Abstract. In this paper, we describe our work on providing a generic yet optimized GPU (CUDA/OpenCL) implementation for the 2D MapOverlap skeleton. We explain our implementation with the help of a 2D convolution application, implemented using the newly developed skeleton. The memory (constant and shared memory) and adaptive tiling optimizations are applied and their performance implications are evaluated on different classes of GPUs. We present two different metrics to calculate the optimal tiling factor dynamically in an automated way which helps in retaining best performance without manual tuning while moving to new GPU architectures. With our approach, we can achieve average speedups by a factor of `3.6`, `2.3`, and `2.4` over an otherwise optimized (without tiling) implementation on NVIDIA C2050, GTX280 and 8800 GT GPUs respectively. Above all, the performance portability is achieved without requiring any manual changes in the skeleton program or the skeleton implementation.

1 Introduction

Multi- and many-core architectures are increasingly becoming part of mainstream computing. There is a clear evidence that future architectures will be heterogeneous, containing specialized hardware, such as accelerator devices (e.g. GPUs) or integrated coprocessors (e.g. Cell's SPUs) besides general purpose CPUs. These heterogeneous architectures provide desired power efficiency at the expense of increased programming complexity as they expose the programmer to different and possibly low-level programming models for different devices in the system. Besides the programming problem, the lack of a universal parallel programming model for these different architectures immediately leads to a portability problem.

Skeleton programming [9] is an approach that could solve the portability problem to a large degree. It requires the programmer to rewrite a program using so-called *skeletons*, pre-defined generic components derived from higher-order functions that can be parameterized in sequential problem-specific code, and for which efficient implementations for a given target platform may exist. A program gets parallelism and leverages other architectural features almost for free for skeleton-expressed computations as skeleton instances can easily be expanded or bound to equivalent expert-written efficient target code that can encapsulate low-level details. This could include details about managing parallelism,load balancing, communication, utilization of SIMD instructions etc. SkePU [1] is such a skeleton programming framework for modern heterogeneous systems that contain multiple CPU cores as well as some specialized accelerators (GPUs).

In earlier work [1], we described the design and implementation of *SkePU*, a new C++ based skeleton programming framework for single- and multi-GPU systems that also has support for multi-core CPUs. The SkePU framework provides several data parallel skeletons including `Map`, `Reduce`, `MapReduce`, `MapArray`, `Scan` (Prefix) and `MapOverlap`, with CUDA and OpenCL implementations for single and multiple GPUs and OpenMP implementations for multi-core CPUs. Besides skeleton implementations, it also provides 1D vector and 2D matrix container types that can be used to pass operand data for a skeleton call.

The MapOverlap skeleton is an important data-parallel skeleton as many applications such as image processing applications [2] and iterative equation solvers [3] can be implemented using that skeleton. It is basically a variation of the Map skeleton and could be defined in the following way for 2D matrix operands (aka. 2D MapOverlap): each element $r[i, j]$ of the result matrix is a function of several adjacent elements of one input matrix that reside at a certain constant maximum *logical* distance from i, j in the input matrix[1]. The maximum logical distance of these elements is controlled by the parameter `overlap`.

In this paper, we discuss our recent work on implementing a generic 2D MapOverlap skeleton and evaluating its performance with the help of a 2D convolution application implemented using that skeleton. This paper provides the following contributions: (1) We develop a generic but optimized 2D MapOverlap implementation for both CUDA and OpenCL backends. (2) We evaluate the effect of applying memory optimizations (usage of constant and shared memory) on cache[2] and non-cache based GPUs. (3) We present two metrics to maximize GPU resource utilization and evaluate their impact on achieved performance. (4) We show how we can achieve *performance portability* while moving from one GPU (CUDA/OpenCL) architecture to another one by an automatic calculation of an optimal configuration for the new architecture.

In the next section, we provide a brief description of the 2D convolution application, followed by a discussion of different GPU optimizations including adaptive tiling in Section 3. In Section 4, we describe two metrics for selection of the tiling factor and their implementation followed by an initial evaluation in Section 5. Related work is presented in Section 6 while Section 7 concludes and proposes future work.

```
int filter_size = filter_width * filter_height;
for(int i=0;i<out_height;i++){
  for(int j=0;j<out_width;j++){
    float sum = 0;
    for(int k=0; k<filter_height; k++){
      for(int l=0; l<filter_width; l++){
         sum += in[i+k][j+l] * filter_weight[k][l];
      }
    }
    out[i][j] = sum / filter_size;
  }
}
```

Listing 1.1: An excerpt from a simple 2D convolution implementation in C++.

2 2D convolution

2D convolution is a kernel widely used in image and signal processing applications. It is a kind of MapOverlap operation where a value for every output pixel is computed based on a weighted average of the corresponding input pixel alongside neighboring input pixels. Listing 1.1 shows an excerpt from a C++ implementation of 2D convolution. In the implementation, the outer two loops iterate over all pixels in the image while the inner two loops are used to compute a new value for a pixel by iterating over its neighboring elements and calculating their weighted average.

3 GPU Optimizations

As a data parallel operation, the MapOverlap computation is well-suited for GPU execution especially for large problem sizes. The naive CUDA implementation can be defined by assigning one thread for computing one output element. Starting from the naive implementation, the CUDA implementation is further optimized in the following ways.

Usage of constant memory

The constant memory is a limited (normally 64KB) cache-buffer that can be used for data stored in GPU device memory. It is hardware optimized for the case when all threads read the same location. The access latency for constant memory ranges from one cycle for in cache data to hundreds of cycles depending on cache locality.

In our framework, the 2D non-separable convolution can be done with or without usage of a filter weight matrix. In case the filter weight matrix is used, as in the 2D convolution application, the filter weights are multiplied with the

[1] The actual access distance between Matrix elements could be different.
[2] GPUs with L1 cache support such as NVIDIA C2050.

corresponding neighboring elements for each output element. As an element in the filter weight matrix is accessed in *read-only* mode by all threads in parallel, it is an ideal candidate for placement in fast constant memory. The performance gains from this optimization depend upon the architecture as it could yield up to 100% performance improvement on non-cache GPUs (e.g. NVIDIA GPUs before the Fermi architecture). However, for NVIDIA Fermi GPUs with explicit L1 cache, the performance improvement with usage of constant memory can be much less due to implicit caching and reuse capabilities of these architectures. For instance, for NVIDIA C2050, performance improvement up to 30% is observed over different filter sizes.

Usage of shared memory

In the naive implementation, every thread in a thread-block loads all the neighboring elements from the GPU device memory, which could be very inefficient especially on non-cache GPUs. Instead, threads in a thread block can use shared memory to store their neighborhood and can subsequently access it from there. This optimization can significantly reduce the global memory bandwidth consumption, especially for large filter weight matrices. For example, for a thread block of 16×32 and filter size of 29×29, each thread block loads 430592 values without usage of shared memory. With the usage of shared memory, the loads are reduced to $(16 + 29 - 1) \times (32 + 29 - 1) = 2640$ values per thread block, a reduction by a factor of 163. Again, the optimization may not yield that much difference in performance while executing on GPUs with L1 cache such as NVIDIA Fermi GPUs.

Adaptive Tiling

Besides memory optimizations mentioned above, another optimization that can be applied to this class of applications on modern GPUs is known as $1 \times N$ tiling [10]. A *tile* refers to a block of input data simultaneously processed by multiple threads in a thread block. $1 \times N$ tiling refers to a technique of increasing workload for a thread block by assigning N tiles to a thread block to process instead of 1 tile. This approach reduces the number of thread blocks by a factor of N. Besides reducing the overhead associated with thread blocks (e.g. array index calculations, loading constant values), this technique also decreases the amount of overall neighborhood loads as the number of thread blocks is decreased. Despite of its potential advantages, tiling can also result in increased shared memory usage by a thread block as now each thread block processes N elements instead of 1. Similarly, register usage can also increase as extra registers are normally used to save intermediate results.

As shown by van Werkhoven et al. [11], using any fixed tiling factor for an image convolution application can result in sub-optimal performance over different filter sizes. Furthermore, with a fixed tiling factor (e.g. 4), the program may simply not work on certain GPUs due to resource limitations (e.g. shared memory size, number of registers). Rather, the *adaptive tiling* introduced in [11] where the tiling factor is chosen based on different filter sizes and resource limitations can be used.

The dynamic selection of the tiling factor is interesting for several reasons. First, there could be several different mechanisms to determine the tiling factor based on different performance characteristics. Furthermore, an automatic way of determining the tiling factor over different machine and problem configurations can help in attaining performance portability.

4 Maximizing resource utilization

Modern GPUs have many types of resources that can affect the performance of an executing kernel. These resources can be broadly categorized into *computational resources* such as ALU and *storage resources* such as registers and shared memory. Effective usage of both kind of resources is important for performance but sometimes a tradeoff exists as both cannot be optimized at the same time.

The adaptive tiling is focused on maximizing utilization of storage resources of a multiprocessor such as shared memory and registers. On the other hand, *warp occupancy* (aka. occupancy) strives for maximizing the computational resource utilization of a multiprocessor[3]. In our work, we consider the tradeoff between these two related but different maximization functions, i.e., maximizing computational resource utilization (i.e. maximizing occupancy) or maximizing storage resource utilization (i.e. maximizing the tiling factor).

Tiling metrics

We define the following two metrics to calculate tiling factors dynamically:

- In the first metric ($\Phi_{occupancy}$), maximizing occupancy is defined as the primary objective while tiling is maximized as a secondary objective. The objective function first strives to achieve maximum occupancy (possibly 100%) while keeping tiling to 1 and later choose to increase the tiling factor to the maximum level possible without decreasing the already determined occupancy level.
- In the second metric (Φ_{tiling}), we do the other way around by maximizing tiling as the primary objective while keeping occupancy to the minimum (i.e. assuming only one thread-block per multiprocessor). The occupancy is considered in case tiling cannot be increased any further (i.e. in our case, we use at most 1×16 tiling)[4].

The metrics differ in their aggressiveness for tiling. As later shown in section 5, $\Phi_{occupancy}$ often results in small tiling factors but greater occupancy while Φ_{tiling} often results in relatively large tiling factors with very low occupancy.

[3] The warp occupancy, as defined by NVIDIA [4], is the ratio of active warps per multiprocessor to the maximum number of active warps supported for a multiprocessor on a GPU.

[4] The limit on the tiling factor is set to allow the secondary objective (i.e. maximizing occupancy) to be considered besides maximizing tiling only.

Performance portability support

Both tiling metrics are implemented in the 2D MapOverlap implementation and the GPU implementation of 2D non-separable convolution can be configured to use either one of them. The input for these objective functions is input (including filter) dimensions and CUDA architecture parameters. The former are automatically inferred by the program execution while the latter are determined based on the compute capabilities of the CUDA architecture. By defining such metrics which can be calculated with very low overhead, we can calculate the tiling filters for different problem sizes on different architectures. In the next section, we demonstrate how performance portability can be achieved with automatic calculation of tiling factors when moving to a new architecture.

	C2050	**GTX280**	**8800 GT**
Compute capability	2.0	1.3	1.1
Number of multiprocessors (MP)	14	30	14
Number of cores in a MP	32	8	8
Processor Clock (GHz)	1.15	1.2	1.5
Local memory (KB)	48	16	16
Cache support	yes (16KB)	no	no
Memory bandwidth (GB/sec)	144	141.7	57.6
Memory interface	384-bit	512-bit	256-bit

Table 1: Evaluation setup.

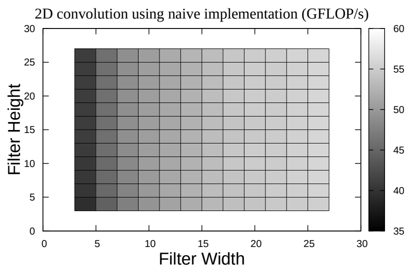
5 Evaluation

The experiments are conducted on three different NVIDIA GPUs with different compute capabilities, as shown in Table 1. For experiments on C2050, an input image of 4096×4096 is used with filter dimensions ranging from 3 up to 27. For GTX 280 and 8800 GT experiments, an input image of 2048×2048 is used with filter dimensions ranging from 3 up to 25.

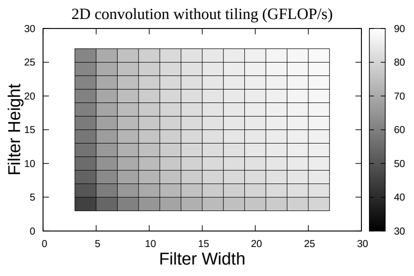
The C2050 was the development platform while the GTX280 and 8800 GT are used to show performance portability and effect of L1 cache. To be realistic, the overhead of calculating the tiling factors for a given objective function is also considered as part of the execution time, which proves to be very negligible. The following implementations are referred to in the evaluation:

- *naive* implementation: The very simple CUDA implementation without any explicit optimization.
- *optimized* implementation: The *naive* implementation with constant memory and shared memory usage.
- *tiling-optimized* implementation: The *optimized* implementation with tiling. The tiling factor could be based upon either $\Phi_{occupancy}$ or Φ_{tiling}.

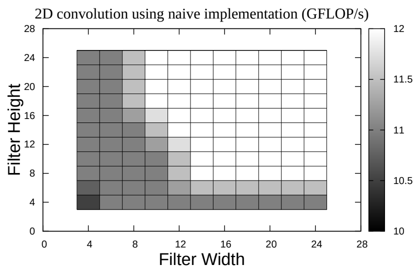
2D map projection is used to present 3D results in Figure 1, 2 and 3. Besides scales on x- and y-axis, please consider the differences in the color-scale in each (sub-)figure for the correct interpretation of results.



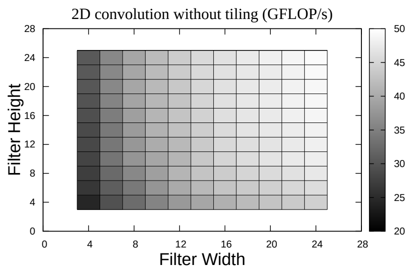
(a) C2050 (50 GFLOP/s)



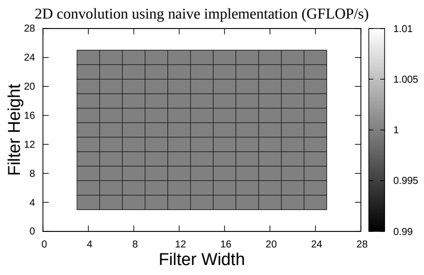
(b) C2050 (76 GFLOP/s)



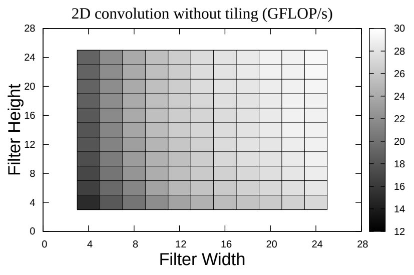
(c) GTX280 (12 GFLOP/s)



(d) GTX280 (41 GFLOP/s)



(e) 8800 GT (1 GFLOP/s)



(f) 8800 GT (25 GFLOP/s)

Fig. 1: 2D convolution with *naive* (a,c,e) and *optimized* (b,d,f) implementations over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.

Usage of shared and constant memory

As mentioned earlier, the effect of applying shared memory and constant memory optimizations is largely influenced by the caching capabilities of a GPU. Figure 1 shows performance improvements over different GPU architectures for the *optimized* implementation over the *naive* implementation. On a cache based C2050, performance improvements are, on average, a factor of almost 1.5. However, on a GTX280 GPU which has no cache, the performance is different by a factor of 3.4. On 8800 GT, the performance difference is by a factor of 25 which is much higher than for the GTX280. This is because of substantial difference in memory bandwidth of 8800 GT and GTX280 (see Table 1) which has a big performance impact for global memory accesses done frequently in the naive implementation.

	$\Phi_{occupancy}$		Φ_{tiling}	
	Tiling factor	Occupancy	Tiling factor	Occupancy
C2050	3.83	100%	14.33	33.34%
GTX280	1.63	75%	3.83	25%
8800 GT	7.35	33.34%	7.35	33.34%

Table 2: Average tiling factor and occupancy achieved with 2 metrics on different GPUs.

Tradeoff between $\Phi_{occupancy}$ and Φ_{tiling}

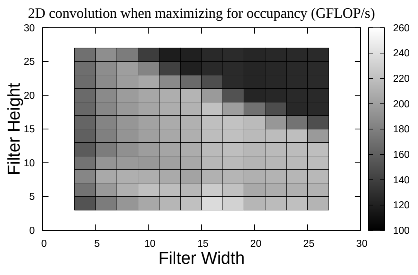
Table 2 highlights the tradeoff between the two metrics on different GPUs. For C2050, when maximizing occupancy ($\Phi_{occupancy}$), the tiling factor is reduced by a factor of 3.74 to gain the last 67% in occupancy. Similarly, for GTX280, the occupancy was much less when optimizing for tiling (Φ_{tiling}) in comparison to when optimizing for occupancy. However, for 8800 GT, the two metrics do not yield any difference. This is because of constraints in maximizing occupancy ($\Phi_{occupancy}$) any further than what is assumed initially in Φ_{tiling} (i.e. 1 thread block per multi-processor).

Performance implications ($\Phi_{occupancy}$ and Φ_{tiling})

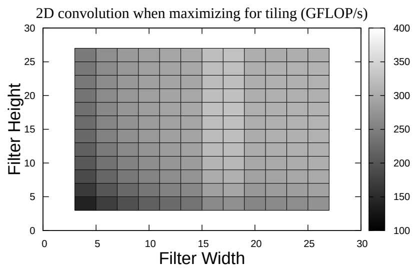
The maximization goal may have a profound impact on the tiling factors chosen and consequently on the achieved performance as shown in Figure 2. Furthermore, on all GPUs, Φ_{tiling} yields better or equal performance than $\Phi_{occupancy}$ for this particular application.

Performance portability

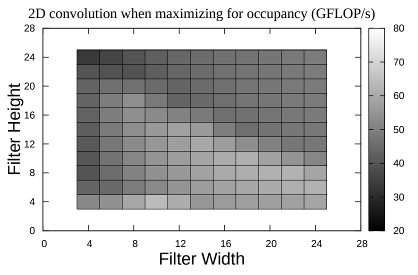
Table 3 shows the CUDA architecture specific parameters that are used to calculate the tiling factors. The table also shows the parameter values for C2050, GTX280 and 8800 GT GPUs which can be obtained easily e.g., by querying



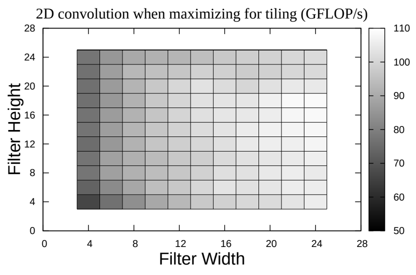
(a) C2050 (180 GFLOP/s)



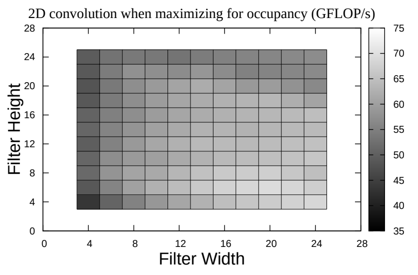
(b) C2050 (274 GFLOP/s)



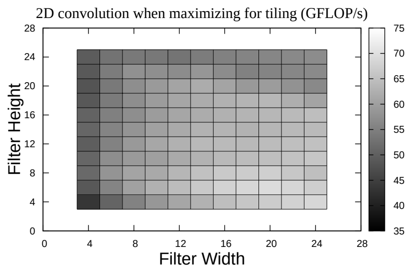
(c) GTX280 (50 GFLOP/s)



(d) GTX280 (95 GFLOP/s)



(e) 8800 GT (59 GFLOP/s)



(f) 8800 GT (59 GFLOP/s)

Fig. 2: 2D convolution when tiling factors are chosen for maximizing either occupancy ($\Phi_{occupancy}$, a,c,e) or tiling (Φ_{tiling}, b,d,f) over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.

	C2050	**GTX280**	**8800 GT**
BLOCK_SIZE_X	16	16	16
BLOCK_SIZE_Y	32	16	16
THREADS_PER_MP	1536	1024	768
NUM_REGISTERS_PER_MP	32768	16384	8192
SHARED_MEM_SIZE_BYTES	48800	16384	16384
THREADS_PER_WARP	32	32	32
WARPS_PER_MP	48	32	24
THREAD_BLOCK_PER_MP	8	8	8

Table 3: CUDA architecture specific parameters.

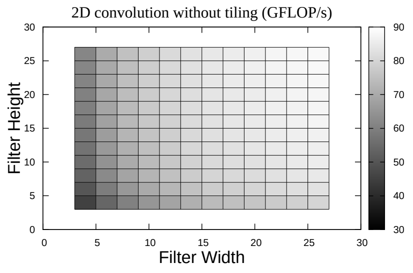
the device capabilities. Based on these parameters alongside the problem size and filter dimensions, the two metrics generate the tiling factors for their corresponding maximization goal. As the tiling factors are automatically calculated and considered for execution, this gives us performance portability when moving from one GPU architecture to another without requiring manual changes in the implementation.

To illustrate performance portability, we compare performance of our `solution` with a `baseline` implementation for a given GPU architecture. For the `solution`, we have used the *tiling-optimized* implementation with Φ_{tiling} as its maximization metric. For the `baseline` implementation, we have considered two alternatives: 1) To use a platform-specific optimized implementation for a given GPU architecture. 2) To use a generic fairly optimized implementation that can be executed on different GPU architectures without requiring rewriting the code.

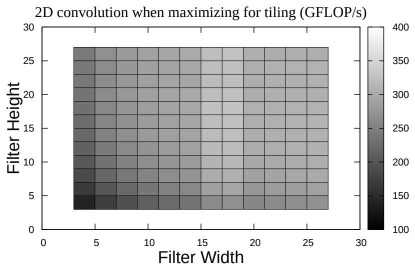
We opted for the second alternative as the first one would require lot of extra effort for writing optimized implementations for each of the three GPUs that we have considered. Following the second alternative, we have chosen our *optimized* implementation as the `baseline` implementation for comparison. The choice is justified as the *optimized* implementation provides significant speedups over *naive* implementation for different class of GPU architectures (see Figure 1) and it is also fairly generic as it can run on any modern GPU with a shared memory support.

We define *relative performance* on a platform as ratio between the average performance of `solution` and the `baseline` implementation. By measuring this ratio, we consider our solution as performance portable if it can retain the relative performance to a potentially higher level (at least `>1`, i.e., better than the baseline implementation for every invocation).

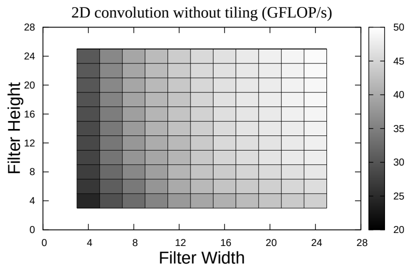
Figure 3 compares the performance of 2D convolution with `solution` over `baseline` implementation. The relative performance is `3.6`, `2.3`, and `2.4` on average for C2050, GTX280 and 8800 GT GPUs respectively which is much higher than our threshold value i.e., `1`.



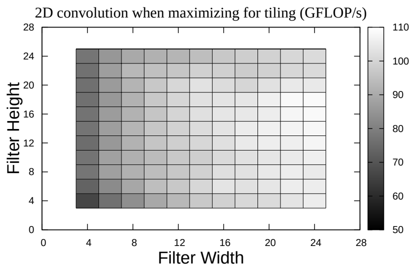
(a) **C2050 (76 GFLOP/s)**



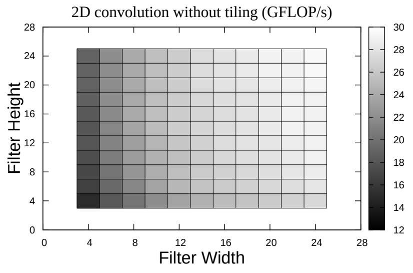
(b) **C2050 (274 GFLOP/s)**



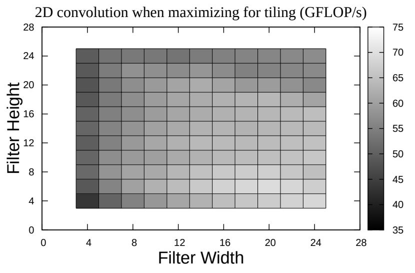
(c) **GTX280 (41 GFLOP/s)**



(d) **GTX280 (95 GFLOP/s)**



(e) **8800 GT (25 GFLOP/s)**



(f) **8800 GT (59 GFLOP/s)**

Fig. 3: Performance (GFLOP/s) for 2D convolution with *optimized implementation* (a,c,e) and *tiling-optimized implementation* (Φ_{tiling}, b,d,f) over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.

6 Related work

The work related most to our work is by van Werkhoven et al. [11]. Our work is more about designing a generic skeleton implementation which can be used for different applications other than 2D convolution. Secondly, we present and evaluate the tradeoffs between different resource utilization criteria with the help of separate metrics. Another important difference is that in our work, we show how to achieve performance portability by automatic calculation of these metrics while moving between different GPU architectures.

There are several other convolution implementations for CUDA. They either work only for separable filters [6] or are based on FFT convolution with fixed filter sizes [7, 8].

7 Conclusions and Future work

Based on our findings, we conclude that performance gains achieved from constant and shared memory optimizations are largely influenced by non-/existence of L1 cache and global memory bandwidth of a GPU architecture. Furthermore, we have shown how to retain performance while porting the application between different GPU architectures by automatic calculation of tiling metrics for the new architecture.

For future work, the approach presented in this paper for achieving performance portability can be used with other skeleton implementations.

References

1. J. Enmyren and C. W. Kessler. SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, USA, Sep. 2010.
2. A. Offringa et al. Post-correlation radio frequency interference classification methods. Monthly Notices of the Royal Astronomical Society, pages 155-167, 2010.
3. M. Korch and T. Rauber. Optimizing locality and scalability of embedded Runge Kutta solvers using block-based pipelining. *Journal of Parallel and Distributed Computing*, 66(3):444–468, 2006.
4. CUDA Occupancy Calculator, NVIDIA corporation, 2007.
5. CUDA Programming Guide v4.0. NVIDIA corporation, 2011.
6. V. Podlozhnyuk. Image Convolution with CUDA NVIDIA corporation, white paper, 2007.
7. V. Podlozhnyuk. FFT-based 2D convolution, NVIDIA corporation, white paper, 2007.
8. S. M. Olesen and S. Lyder. Applying 2D filters using GPUs and CUDA Vision 3 - Advanced Topics in Computer Vision, 2010.
9. C. Kessler et al. Skeleton Programming for Portable Many-Core Computing. *Programming Multi-Core and Many-Core Computing Systems*. Wiley, June 2012.
10. S. Ryoo et al. Program optimization space pruning for a multithreaded GPU. In *CGO '08: Proc. Int. IEEE/ACM symposium on Code generation and optimization*, pages 195-204, USA, 2008.
11. B. Van Werkhoven et al. Optimizing Convolutions Operations in CUDA with Adaptive Tiling, In *A4MMC '11: Proc. Workshop on Applications for Multi and Many Core Processors*, USA, 2011.