# Practical examples of GPU Computing Optimization Principles

Patrik Goorts[1]
[1,2,4]Hasselt University - tUL - IBBT
Expertise centre for Digital Media
Wetenschapspark 2
BE-3590 Diepenbeek
Belgium
[1]patrik.goorts@uhasselt.be

Sammy Rogmans[2]
[2]Multimedia Group, IMEC
Kapeldreef 75
BE-3001 Leuven
Belgium
[2]sammy.rogmans@uhasselt.be

Steven Vanden Eynde[3] Philippe Bekaert[4]
[3]Lessius Hogeschool – Campus De Nayer
J. De Nayerlaan 5
BE-2860 Sint Katelijne Waver
Belgium
[3]steven.vandeneynde@gmail.com
[4]philippe.bekaert@uhasselt.be

***Abstract*—In this paper, we provide examples to optimize signal processing or visual computing algorithms written for SIMT-based GPU architectures. These implementations demonstrate the optimizations for CUDA or its successors OpenCL and DirectCompute. We discuss the effect and optimization principles of memory coalescing, bandwidth reduction, processor occupancy, bank conflict reduction, local memory elimination and instruction optimization. The effect of the optimization steps are illustrated by state-of-the-art examples. A comparison with optimized and unoptimized algorithms is provided. A first example discusses the construction of joint histograms using shared memory, where optimizations lead to a significant speedup compared to the original implementation. A second example presents convolution and the acquired results.**

***Index Terms*—CUDA, GPGPU, optimization principles, visual computing, Fermi.**

## I. Introduction

In the last years, generic parallel computing on graphical devices is becoming popular for versatile problems. GPUs are used in generic visual computing and games, but also in medical fields [Shams et al., 2010], biology [Phillips et al., 2005], physics and many more. Thanks to the great interest in off-the-shelf parallel devices, a lot of research and work is conducted to enable ease of programming of graphical devices. These advances remove the need to map every problem to a visual representation processable by the graphical pipeline. With the introduction of CUDA [NVIDIA, 2010], it is possible to use generic instructions on a parallel device; no graphical pipeline is used. While CUDA eases the use of GPUs for parallel computation, the efficient programming is still difficult. Due to the different limitations and the memory hierarchy, very fast execution is possible, but some effort must be made to maximize the performance.

In this paper, we will investigate some considerations when implementing computer vision algorithms on CUDA-enabled devices. These algorithms typically need a lot of memory to store and save data. The copying of these data between on-chip and off-chip memory can take more time than the actual calculations, as stated by the infamous memory wall [Asanovic et al., 2006]. To tackle this problem, special care should be taken to efficiently perform memory operations.

Another less-known problem includes the access strategy of on-chip memory. While the access time is short, multiple threads shall access the memory simultaneously. This simultaneous access can significantly reduce performance. This kind of optimizations is less documented and investigated. We applied the reduction of these so-called bank conflicts to two examples and reduced the execution time.

We will provide examples where all of this optimizations are effective. These examples demonstrate the importance of optimizations which are often forgotten in state-of-the-art algorithms. We will focus on CUDA-enabled devices, but these optimizations still hold for OpenCL [Munshi, 2008] and DirectCompute [Boyd, 2008], as these use the same execution model as CUDA.

The paper is divided as follow: in section 2 we will provide a general overview to optimize the parallel execution of algorithms on SIMT-based architectures and section 3 will provide some optimization examples. We will conclude in section 4.

## II. Parallelization of algorithms

In this section, we will provide an overview to optimize algorithms implemented on SIMT architectures. The first step is the division of the algorithm in threads. Threads should be chosen to reduce the communication between them. CUDA allows fast communication through shared memory, but global communication is slow and unsynchronized. However, using shared memory can reduce redundant memory reads and improve performance. After defining threads, these must be grouped in blocks. These choices must be made according to the code optimization guidelines, including:

1) **Global memory coalescing**: Optimize off-chip memory access by grouping successive memory loads in one instruction. The memory bus is less occupated, thus allowing more throughput.
2) **Global memory bandwidth reduction**: Reduce total amount of memory requirements.
3) **Increasing occupancy**: Use more processing power simultaneously by defining enough threads to keep every processor busy.

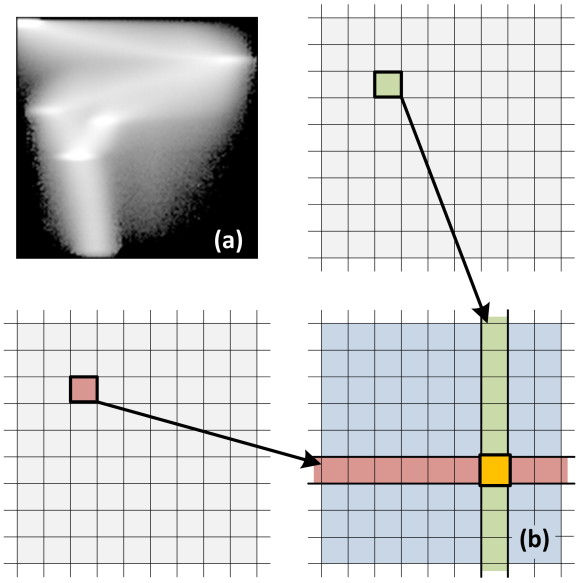


Fig. 1. Calculation of a joint histogram. The pixel value in the first image and the corresponding pixel value in the second image defines which position in the joint histogram should be updated. (a) Example of a joint histogram. (b) Matrix representation.

4) **Reduce bank conflicts**: Optimize on-chip memory access by avoiding operations in the same memory segment (bank).
5) **Eliminate local memory usage**: Reduce costly register swapping to slow off-chip memory.
6) **Optimize instructions**: Reduce clock cycles of the calculations.

These principles go further than previous research [Ryoo et al., 2008], providing more low-level and often forgotten strategies.

## III. Optimization examples

In this section we will provide two state-of-the-art examples where optimization increased the performance. The first example calculates joint histograms based on the method of [Shams and Kennedy, 2007]. The second example investigates the effect of coalescing and bank conflicts on convolution calculation and is based on the work described in [Goorts et al., 2009].

### A. Joint Histogram Calculations

We will now provide an example of a visual computing algorithm parallelized with a SIMT architecture, the calculation of joint histograms. Given two images with given pixel correspondences, a 2D histogram is created. For every pixel in the first image and the corresponding pixel in the second image, a greyvalue pair is obtained. The 2D histogram counts the occurrence of every pair (see figure 1). Joint histograms are widely used in image registration algorithms.

Dividing the calculation of joint histograms in threads is possible in different ways. A thread can process a pair of greyvalues and fill in the histogram, or a thread can process a field in the histogram and counting the corresponding pairs in the image. In the former case every thread must have write-access to every element in the histogram and a concurrent write system must be provided. In the latter case every thread must read the full image, and as a consequence the algorithm doesn't scale well for large images.

| Implementation | Time |
|---|---|
| Original implementation | 350 msec |
| Optimized implementation (atomic) | 196 msec |
| Optimized implementation (tagged) | 190 msec |

TABLE I
Results of joint histogram calculations

The implementation used here is described in [Shams and Kennedy, 2007]. This method uses a histogram per warp (16 concurrent running threads), located in shared memory, producing subhistograms. These subhistograms are added together first at block level and later globally in a second kernel. Because threads in the same warp can access the same memory location, some write protection must be provided. In [Shams and Kennedy, 2007], this problem is solved by tagging the values with the thread ID. The written values are read again after the write, and the tag is compared. If the tag is wrong, another thread has written its value and the write will be retried. Eventually, every thread will have updated the memory location.

```
int bin = ...;
unsigned int tagged;
do
{
  \\ Remove previous tag
  unsigned int val =
     localHistogram[bin] & 0x07FFFFFF;

  \\ Tag the value with the thread id
  tagged = (threadid << 27) | (val + 1);

  \\ Write to the histogram
  localHistogram[bin] = tagged;
} while (localHistogram[bin] != tagged);
```

This method was originally developed before atomic operations in shared memory were provided. Here, we will investigate the original method, but using these atomic operations and other optimization techniques.

We have performed joint histogram calculations of images of size 4096x4096 on a NVIDIA GTX 280 with 30 multiprocessors, a clock speed of 1.3 GHz and 1 GiB of off-chip memory. As shown in Table I, the version of [Shams and Kennedy, 2007] is more efficient when using optimization techniques. More specifically, we have reduced the size of the data to speed up memory reads, enabled colaesced loads and decreased bank conflicts. These optimizations can be

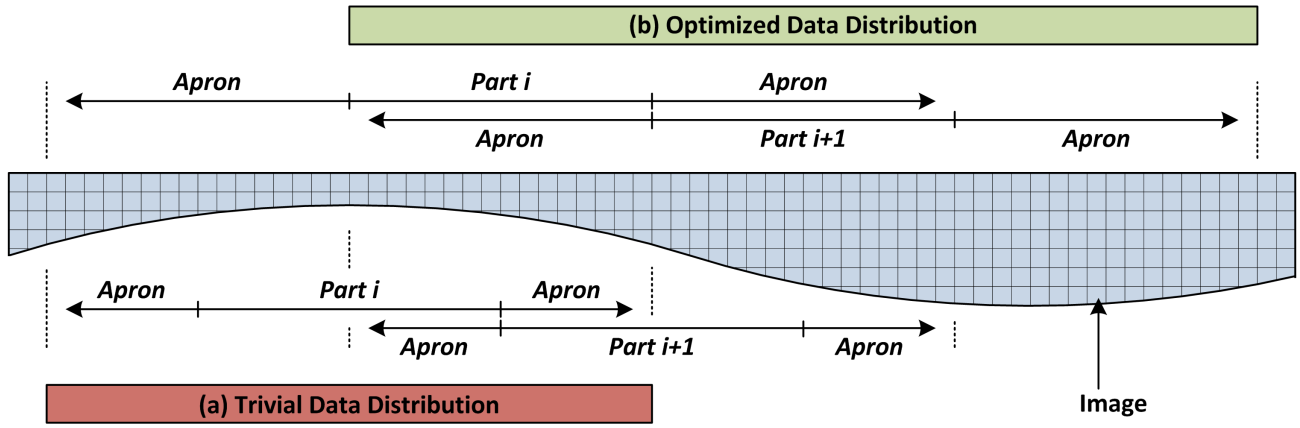


Fig. 2. Alignment of the threads when performing convolution. The apron is a multiple of 16 to enable coalescing in all blocks.

accomplished by rearranging the threads, and thus control the memory operations per thread. Additionally, we have modified the calculation instructions to reduce registry usage and thus reduce register swappng to global memory. By using these optimizations, we accomplished a speedup of 150 msec, which demonstrates the importance of attention to optimization.

Using atomic add operations on shared memory to remove the tagging technique does not result in faster execution. This demonstrates that care should be taken when using atomic operations, even in small kernels, and other techniques should be implemented and benchmarked if possible.

### B. Convolution

Convolution is an image processing method where for every pixel a new pixel value is calculated in function of the values surrounding the current pixel. Convolution is often referred to as finite impulse response filtering. Effects of block sizes and implementation strategies are extensively investigated in [Goorts et al., 2009]. Here, we will discuss the importance of the position of threads to enable coalescing and reduce bank conflicts. We will only consider normal convolution

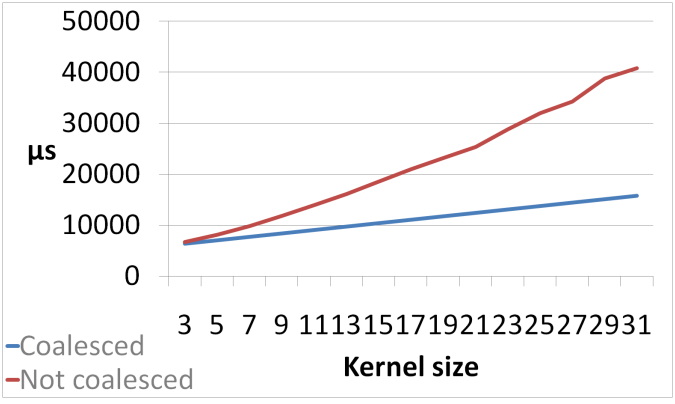


Fig. 3. Horizontal convolution of a 4096x4096 image with different filter sizes. The importance of coalescing increases if the filter sizes increase; bigger filters require larger aprons with more uncoalesced reads and more memory operations, while the coalesced algorithm requires only one read per 16 words.

algorithms; techniques using fast Fourier transformations and singular value decompositions are not in the scope of this paper.

One thread per pixel is assigned. Every thread now needs the value of the surrounding pixels, requiring to load multiple pixels from off-chip memory. Because threads in the same block have a shared memory, it is possible to use the data read by other threads and thus enabling data reuse to reduce memory reads. However, threads at the border of the block do not have sufficient data available and therefore a border of threads is created to read the missing data. This border is called the apron. Because these threads do not calculate new output, the pixels located in the apron must be read by different blocks. These redundant reads must be reduced to a minimum.

Because of the apron, it is not sufficient to use blocks with the for coalescing required width of a multiple of 16. The width of the apron must also be a multiple of 16 (see figure 2) to enable coalescing in every block. By doing this, we will create a lot of unnecessary threads, but the performance will increase thanks to coalesced memory operations and the elimination of bank conflicts. This result is visible in figure 3. The results are dependent on the filter size; the larger the filter, the larger the apron. Because the non-coalesced algorithm must read every word in the apron apart, the memory instructions will increase if the apron size increases. In the coalesced case, the read instructions stay constant and optimal. There will be more thread and more blocks, but the performance is still higher, which proves the importance of coalescing in this application. Because the thread positions are chosen correctly, bank conflicts are removed from the coalesced case.

## IV. Conclusion

We have applied optimization principles to increase the performance of algorithms executed on SIMT architectures. By coalescing off-chip memory loads, reducing bandwidth, increasing occupancy, reducing bank conflicts, eliminating local memory usage and optimizing instructions, one can maximize the utilization of the resources of the parallel device and reduce

execution time. Reducing bank conflicts is forgotten by most programmers, but these can increase performance significantly. We have demonstrated the effectiveness of the optimizations with two state-of-the-art examples.

## REFERENCES


[Asanovic et al., 2006] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The Landscape of Parallel Computing Research: A View From Berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley*, 18(183):19.

[Boyd, 2008] Boyd, C. (2008). The DirectX 11 Compute Shader. Shading Course SIGGRAPH.

[Goorts et al., 2009] Goorts, P., Rogmans, S., and Bekaert, P. (2009). Optimal data distribution for versatile finite impulse response filtering on next-generation graphics hardware using cuda. *Parallel and Distributed Systems, International Conference on*, pages 300–307.

[Munshi, 2008] Munshi, A. (2008). OpenCL: Parallel Computing on the GPU and CPU. Shading Course SIGGRAPH.

[NVIDIA, 2010] NVIDIA (2010). What is cuda? http://www.nvidia.com/object/what_is_cuda_new.html.

[Phillips et al., 2005] Phillips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R. D., Kal, L., and Schulten, K. (2005). Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, 26(16):1781–1802.

[Ryoo et al., 2008] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and mei W. Hwu, W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPOPP*, pages 73–82.

[Shams and Kennedy, 2007] Shams, R. and Kennedy, R. A. (2007). Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, Gold Coast, Australia.

[Shams et al., 2010] Shams, R., Sadeghi, P., Kennedy, R. A., and Hartley, R. I. (2010). A survey of medical image registration on multicore and the GPU. *IEEE Signal Processing Mag. (to appear)*.