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