Towards User Transparent Parallel Multimedia Computing on GPU-clusters

Ben Van Werkhoven, Jason Maassen, Frank J. Seinstra

Towards User Transparent Parallel Multimedia Computing on GPU-clusters

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

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

Abstract. The research area of Multimedia Content Analysis (MMCA) considers all aspects of the automated extraction of knowledge from multimedia archives and data streams. To satisfy the increasing computational demands of MMCA problems, the use of High Performance Computing (HPC) techniques is essential. As most MMCA researchers are not HPC experts, there is an urgent need for 'familiar' programming models and tools that are both easy to use and efficient.
Today, several *user transparent* library-based parallelization tools exist that aim to satisfy both these requirements. In general, such tools focus on data parallel execution on traditional compute clusters. As of yet, none of these tools also incorporate the use of many-core processors (e.g. GPUs), however. While traditional clusters are now being transformed into GPU-clusters, programming complexity vastly increases — and the need for easy and efficient programming models is as urgent as ever.
This paper presents our first steps in the direction of obtaining a user transparent programming model for data parallel and hierarchical multimedia computing on GPU-clusters. The model is obtained by extending an existing user transparent parallel programming system (applicable to traditional compute clusters) with a set of CUDA compute kernels. We show our model to be capable of obtaining orders-of-magnitude speed improvements, without requiring *any* additional effort from the application programmer.

1 Introduction

In recent years, the generation and use of multimedia data, in particular in the form of still pictures and video, has become more and more widespread. The application and research domain of Multimedia Content Analysis (MMCA) investigates new methods of image and video processing, to arrive at automated techniques of extracting knowledge from multimedia data. In part, research in the MMCA domain is driven by requirements of emerging applications, ranging from the automatic comparison of forensic video evidence, to searching publicly available digital television archives, and real-time analysis of video data obtained from surveillance cameras in public locations [1].

In the very near future, computerized access to the content of multimedia data will be a problem of phenomenal proportions, as digital video may produce

high data rates, and multimedia archives steadily run into petabytes of storage space. As a result, high-performance computing (HPC) on clusters or even large collections of clusters (e.g., grids) is rapidly becoming indispensable.

Unfortunately, writing efficient parallel applications for such systems is known to be hard. Worse, with the integration of many-core technologies (e.g., GPUs), programming complexity is increased even further. As most MMCA researchers are not also HPC or many-core computing experts, there is a need for *user transparent* programming models and tools [2–6] that can assist in creating efficient parallel and hierarchical MMCA applications. Ideally, such tools require little or no extra effort compared to traditional (sequential) MMCA tools, and lead to efficient execution in most application scenarios.

Existing user transparent programming tools are based on a software library of pre-parallelized compute kernels that cover the bulk of all commonly applied MMCA functionality [2–7]. These tools, however, all aim at data parallel execution on traditional clusters, and do not incorporate the use of many-cores.

In this paper we describe our first steps in obtaining a user transparent programming model for executing MMCA applications on GPU-clusters. The model extends the successful Parallel-Horus library for user transparent parallel MMCA [7] with the capacity of executing special purpose compute kernels implemented using CUDA [8]. As such, the model is the first programming model in the MMCA domain that completely hides the complexities of (data) parallel and hierarchical computing behind a sequential programming interface.

This paper is organized as follows. Section 2 briefly describes the Parallel-Horus library. Section 3 presents the GPU-extensions to Parallel-Horus, and discusses some of the more important design issues. In Section 4, we describe a simple example MMCA application. This is followed by an evaluation in Section 5. Finally, we present future work and conclusions in Sections 6 and 7.

2 Parallel-Horus

In our earlier work, we have designed and implemented Parallel-Horus [7], a user transparent parallelization tool for the MMCA domain. Parallel-Horus, implemented in C++ and MPI, allows programmers to implement *data parallel* multimedia applications as fully sequential programs. The library's API is made identical to that of an existing sequential library: Horus [9]. Similar to other frameworks [3], Horus recognizes that a small set of *algorithmic patterns* can be identified that covers the bulk of all commonly applied functionality.

Parallel-Horus includes patterns for commonly used functionality such as unary and binary pixel operations, global reduction, neighborhood operation, generalized convolution, and geometric transformations (e.g. rotation, scaling). Recent developments include patterns for operations on large datasets, as well as patterns on increasingly important derived data structures, such as feature vectors. For reasons of efficiency, all Parallel-Horus operations are capable of adapting to the performance characteristics of the cluster computer at hand, i.e. by being flexible in the partitioning of data structures. Moreover, it was realized

that it is not sufficient to consider parallelization of library operations *in isolation*. Therefore, the library was extended with a run-time approach for communication minimization (called *lazy parallelization*) that automatically parallelizes a fully sequential program at run-time by inserting communication primitives and additional memory management operations whenever necessary [10].

Results for realistic multimedia applications have shown the feasibility of the Parallel-Horus approach, with data parallel performance (obtained on traditional cluster systems) consistently being found to be optimal with respect to the abstraction level of message passing programs [7]. Notably, Parallel-Horus was applied in earlier NIST TRECVID benchmark evaluations for content-based video retrieval, and played a crucial role in achieving top-ranking results in a field of strong international competitors [1, 7]. Moreover, recent extensions to Parallel-Horus, that allow for services-based distributed multimedia computing, have been applied successfully in large-scale distributed systems, involving hundreds of massively communicating compute resources covering the entire globe [7].

Despite these successes, Parallel-Horus currently does not incorporate any solutions for the use of many-core technologies. This is unfortunate, as many of the basic compute kernels in Parallel-Horus are particularly suited for execution on (e.g.) a GPU. The following, therefore, describes our first steps towards obtaining a single, integrated solution for GPU-clusters.

3 GPU-based Extensions to Parallel-Horus

Parallel-Horus extends the Horus library by introducing a thin layer right in the heart of the small set of algorithmic patterns, as shown in Figure 1. This layer uses MPI to communicate image data and other structures among the different nodes in a cluster. In the most common case, a digital image is *scattered* throughout the parallel system, such that each compute node is left with a *partial image*. Apart from the need for additional pre- and post-communication steps (such as the common case of *border handling* in convolution operations), the original sequential compute kernels available in Horus are now applied to each partial image.

From a software engineering perspective, the fact that the MPI extensions 'touch' the original sequential Horus implementation in such a minimal way provides Parallel-Horus with the important properties of sustainability and easy extensibility. In the process of extending Parallel-Horus for GPU-based execution, we aim at a similar minimal level of intrusiveness. To this end, we have decided to leave the thin communication layer as it is, and focus on introducing CUDA-based alternatives to the sequential compute kernels that implement the algorithmic patterns (see bottom half of Figure 1). In this manner, MPI and CUDA are able to work in concert, allowing the use of multiple GPUs on the same node, and on multiple nodes simultaneously, simply by creating one MPI process for each GPU. In other words, this approach directly leads to a system that can execute fully sequential Horus applications in data parallel fashion, while exploiting the power of GPU hardware.

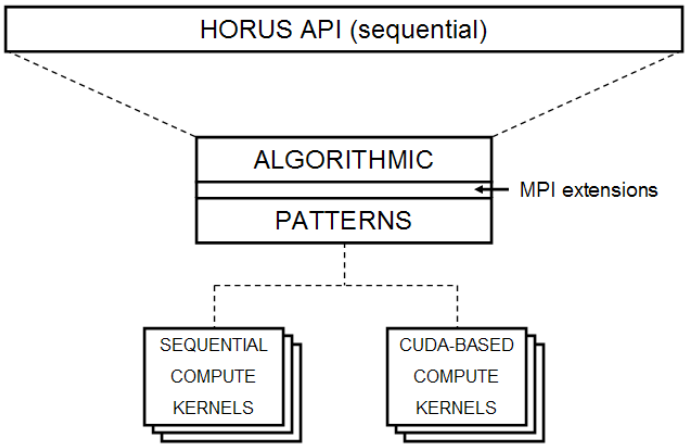


Fig. 1. General overview of Parallel-Horus and its CUDA-based extensions.

A typical Parallel-Horus application spends most of its total execution time performing compute-intensive image processing operations, such as Gaussian filtering. As these operations have a high degree of data parallelism, they can be easily rewritten to execute on many-core processors. As an example, we focus here on a compute kernel that implements the algorithmic pattern of generalized convolution. The pseudo code for this compute kernel is shown in Figure 2(a). The sequential algorithm uses two loops to iterate over all pixels in the image. In the process, for each image pixel a weighted average is calculated using the current pixel and all pixels in its *neighborhood*. The average is obtained by taking an average of all pixels in the neighborhood, weighing each pixel using the values stored in the convolution filter. The size of the neighborhood equals the size of the convolution filter.

The simplest way to obtain a CUDA-implemented compute kernel for generalized convolution is to unroll the outer two loops and let every CUDA thread perform a single iteration. The resulting CUDA code, which we will refer to as the *naive* implementation, is shown in Figure 2(b). The effort required for writing the naive kernel is minimal, while the speedups obtained with this implementation are already significant. As will be further discussed in Section 5, a simple application based on this naive compute kernel was found to improve its absolute speed by a factor of 29 on our testbed.

To further improve the performance of our generalized convolution we need to increase the *compute to global memory access ratio* [11] or, in other words, the number of floating point operations per global memory access. The compute-to-memory ratio for the naive kernel is 1.0 or 2:2, namely: one multiply operation and one add operation versus one load for the image pixel and one load for the filter value. A simple strategy to nearly double the compute-to-memory ratio of our compute kernel is to store the convolution filter in *constant memory*.

```
FOR all rows y in I_h DO
  FOR all columns x in I_w DO
    sum := 0
    FOR all rows j in F_h DO
      FOR all columns i in F_w DO
        sum += I[y-F_h+j][x-F_w+i] * F[j][i]
      OD
    OD
    result[y][x] = sum / F_w ×  F_h
  OD
OD
```

(a)

```
x = threadIdx.x+blockIdx.x*BLOCK_SIZE;
y = threadIdx.y+blockIdx.y*BLOCK_SIZE;
sum := 0
FOR all rows j in F_h DO
  FOR all columns i in F_w DO
    sum += I[y-F_h+j][x-F_w+i] * F[j][i]
  OD
OD
result[y][x] = sum / F_w ×  F_h
```

(b)

Fig. 2. (a) Pseudo code for generalized convolution in Horus. 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. (b) Pseudo code for the same algorithm implemented as a CUDA kernel (naive). The outer two loops have been replaced by two lines that compute which iteration of the outer two loops is performed by each thread.

Constant memory is a cached read-only memory which is visible to all threads. Accesses to the convolution filter are independent of each thread's `threadIdx`. As a result, all threads will try to load the same filter element simultaneously. This access pattern is ideally suited for the cached constant memory, as accesses to the convolution filter will be broadcast from the cache to all threads. We have implemented a version of our compute kernel in this manner, which we will refer to as the *constant* implementation in the remainder of this paper. As further optimizations of our CUDA compute kernels are outside the scope of this paper, we will address them in future work.

A problem in the implementation of our CUDA kernels is that not all of the Horus kernels are sufficiently compute intensive for many-core processors. For example, a binary pixel operation that performs a pixelwise addition of two images, has a compute-to-memory ratio of only 0.5, namely: one add operation, and two operations that load the pixels from global memory. Moreover, the naive approach of unrolling the outer two loops and creating a CUDA thread for each iteration, leaves each thread with only a single operation to perform. Therefore, in our implementation of such compute kernels only one row of threads is created, and each thread works on one column of image data. The row of threads then processes the image data row by row. The column-wise access pattern that arises in this manner is common in CUDA programming, since it ensures coalesced access to global memory [11]. In fact, to increase the number thread blocks, multiple rows of threads are created, each of which processes a horizontal partition of the image.

Apart from optimizing the performance of each compute kernel *in isolation*, it is also important to optimize the execution of multiple compute kernels applied in sequence (as is typical for Horus applications). As such, several implementation strategies can be considered when replacing the original Horus kernels with their CUDA counterparts. In the simplest approach, every call to a CUDA kernel requires allocating device memory, copying the image(s) to device memory, executing the kernel, copying the resulting data structure back to main memory,

and finally deallocating device memory. Although this approach is very simple to implement, a hand-coded version of the same application will most likely be more efficient. In typical applicatons, the above approach will lead to unnecessary memory allocation operations and redundant movements of data between device memory and host memory.

To overcome the problem of redundant memory operations, we apply an optimization approach which we refer to as *lazy data copying*. In this approach, the library assumes that the most up-to-date copy of the data required in the execution of a next compute kernel is always on the GPU, unless it is explicitly marked as being stale. While this approach avoids all unnecessary data movements between host memory and device memory, linking up the CUDA kernels with the Parallel-Horus library now must be done with utmost care. For one, this is because GPU copies of data structures can become stale due to the normal workings of an imaging application itself, e.g. when using I/O operations that replace an already existing image with a new one. More importantly, this is because some parallel versions of the imaging operations require the host copy of an image to be up-to-date. One example is the generalized convolution, that requires all participating compute nodes in a parallel system to exchange border data using MPI. This exchange is necessary as the evaluation of each individual pixel requires information about that pixel's neighbors. Hence, just before the border exchange, our extended library must synchronize the host copy and the device copy of an image, if the device copy is more recent.

In Section 5, we will present a thorough evaluation of all CUDA implementations described, and of the related optimization strategies. First, however, we will present a well-known application that is used in our evaluations.

4 A Line Detection Application

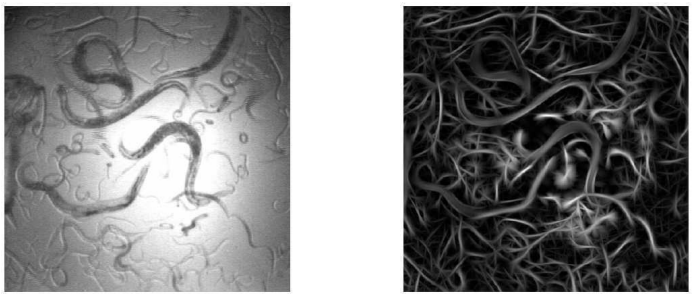
The following describes a typical, yet simple, example application from the MMCA domain. The example is selected because results for data parallel execution with Parallel-Horus are well available, and presented in [12].

4.1 Curvilinear Structure Detection

As discussed in [13], the computationally demanding problem of line detection is solved by considering the second order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation θ, smoothing scale σ_u in the line direction, and differentiation scale σ_v perpendicular to the line, given by:

$$r''(x, y, \sigma_u, \sigma_v, \theta) = \sigma_u \sigma_v \left| f_{vv}^{\sigma_u, \sigma_v, \theta} \right| \frac{1}{b^{\sigma_u, \sigma_v, \theta}}, \tag{1}$$

with b the line brightness. When the filter is correctly aligned with a line in the image, and σ_u, σ_v are optimally tuned to capture the line, filter response is



(a) (b)

Fig. 3. Detection of *C. Elegans* worms (courtesy of Janssen Pharmaceuticals, Belgium).

maximal. Hence, the per pixel maximum line contrast over the filter parameters yields line detection:

$$R(x,y) = \arg \max_{\sigma_u,\sigma_v,\theta} r''(x,y,\sigma_u,\sigma_v,\theta). \tag{2}$$

Figure 3(a) gives a typical example of an image used as input. Results obtained for a reasonably large subspace of $(\sigma_u, \sigma_v, \theta)$ are shown in Figure 3(b).

The anisotropic Gaussian filtering problem can be implemented in many different ways. In this paper we consider two possible approaches. First, for each orientation θ it is possible to create a new filter based on σ_u and σ_v. In effect, this yields a rotation of the filters, while the orientation of the input image remains fixed. Hence, a sequential implementation based on this approach (which we refer to as *Conv2D*) implies full 2-dimensional convolution for each filter.

The second approach (referred to as *ConvUV*) is to decompose the anisotropic Gaussian filter along the perpendicular axes u, v, and use bilinear interpolation to approximate the image intensity at the filter coordinates. Although comparable to the Conv2D approach, ConvUV is expected to be faster due to a reduced number of accesses to the image pixels. Pseudo code for the two algorithms is almost identical, as presented in Figure 4.

```
FOR all orientations θ DO
  FOR all smoothing scales σu DO
    FOR all differentiation scales σv DO
      FiltIm1 = GenConvOp(OriginalIm, "func", σu, σv, 2, 0);
      FiltIm2 = GenConvOp(OriginalIm, "func", σu, σv, 0, 0);
      ContrastIm = BinPixOp(FiltIm1, "absdiv", FiltIm2);
      ContrastIm = BinPixOp(ContrastIm, "mul", σu × σv);
      ResultIm = BinPixOp(ResultIm, "max", ContrastIm);
    OD
  OD
OD
```

Fig. 4. Conv2D and ConvUV, with `"func"` either `"gauss2D"` or `"gaussUV"`.

	1x1	1x2	2x1	3x1	4x1	2x2	3x2	4x2
ConvUV	244.1	122.2	122.1	82.0	61.9	61.9	43.0	32.3
Conv2D	2302.7	1151.0	1151.6	768.3	577.7	577.2	384.8	291.2

Table 1. The total execution times in seconds of the ConvUV and Conv2D applications, using Parallel-Horus without GPU extensions.

5 Evaluation

In this section we evaluate multiple versions of the ConvUV and Conv2D applications that we have implemented using the original Parallel-Horus system, and with the CUDA-based extentions. We have tested the applications using the Lisa GPU Cluster, located at SARA (Stichting Academisch Rekencentrum Amsterdam). Although the traditional part of the cluster is much larger, the system currently has a total of 6 nodes available that are equipped with GPU accelerators. Each of these nodes is equipped with two Quad-Core Intel Xeon CPUs running at 2.50GHz, with 32 GBytes of host memory. The nodes have 2 Nvidia Tesla M1060 graphics adapters, each having 240 cores and 4 GBytes of device memory. In our experiments we use many different configurations, each of which is denoted differently by the number of nodes and CPUs/GPUs used. For example, measurements involving one compute node and one MPI process per node are denoted by 1x1. Likewise, 4x2 means that 4 nodes are used with 2 MPI processes executing on each node. For the CUDA-based executions, the latter case implies the concurrent use of 8 GPUs.

For each configuration we have first tested the performance of the two applications using the orignal Parallel-Horus implementation. In addition, we have performed the same measurements using four different implementations of Parallel-Horus with GPU extensions. These versions are: (1) *naive*: the original naive implementation, without any further optimizations, (2) *constant*: the improved implementation that uses constant memory, (3) *naive lazy*: the improved implementation that uses lazy data copying, and (4) *const lazy*, the implementation that uses constant memory as well as lazy data copying.

Table 1 shows the total execution times for the ConvUV and Conv2D applications, using Parallel-Horus without the GPU extensions. As expected, the Conv2D application demands far more CPU time since it uses full 2-dimensional convolution for each filter, while the ConvUV application only performs bilinear interpolation to approximate the image intensity at the coordinates of the separable filter. As more MPI processes are added to the configuration the total execution time reduces linearly. These results are entirely in line with earlier speedup characteristics reported in [12] for much larger cluster systems. We would like to stress that these speedup results are obtained without requiring *any* parallelization effort from the application programmer.

The execution times of the four GPU versions of the ConvUV application are shown in Figure 5. Note that the total execution times also include the sequential part of the application, consisting mainly of reading and writing the

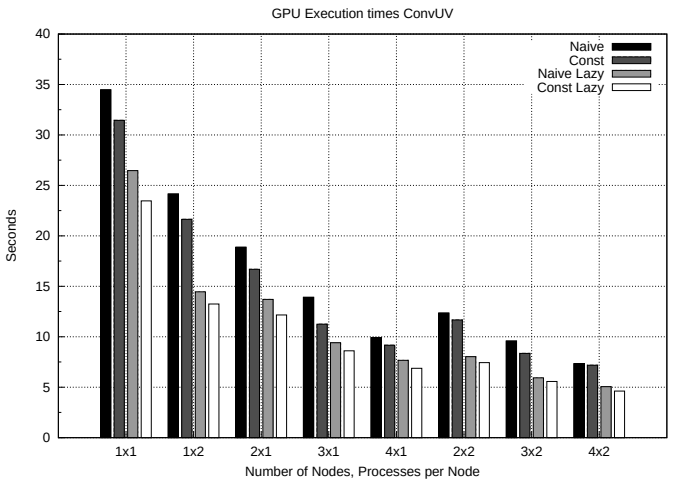


Fig. 5. The total application execution times for four different versions of the GPU-extended ConvUV application.

input and output images to a file. The CUDA kernels that use constant memory reduce the total execution time of the ConvUV application by 10% on average, compared to the naive implementation. However, even more time (up to 38%) can be saved when lazy data copying is used to reduce the movements of data between device and host memory. For a single node in our testbed, the execution time of the ConvUV application improves by a factor of 10.4 when using a single GPU, compared to running Parallel-Horus without GPU-extensions.

Configurations 1x2 and 2x1 both employ the same number of GPUs, while using a different number of nodes. The same holds for the 2x2 and 4x1 configurations. Despite the fact that for these configurations the number of GPUs is pairwise identical, the naive and const implementations perform better for the 2x1 and 4x1 cases than for 1x2 and 2x2 cases, respectively. When lazy data copying is used to reduce the number of memory management operations and data movements, the difference in execution times between these case becomes significantly less. This suggests that the performance penalty caused by using multiple GPUs on the same node, instead of different nodes, is most likely caused by the large number of calls made to the CUDA run-time. Another possible cause for the slowdown could be the memory bus. However, since the images used to test the application are fairly small (1088×1088 4-byte pixels), we assume that the memory bus is not saturated with copying data between host and device.

The execution times of the four GPU versions of the Conv2D application are shown in Figure 6. As expected the constant memory implementation is much more efficient than the naive implementation, and reduces the total execution time by 40% on average. The performance improvement caused by using constant memory for the convolution filter is more dramatic for Conv2D than for ConvUV.

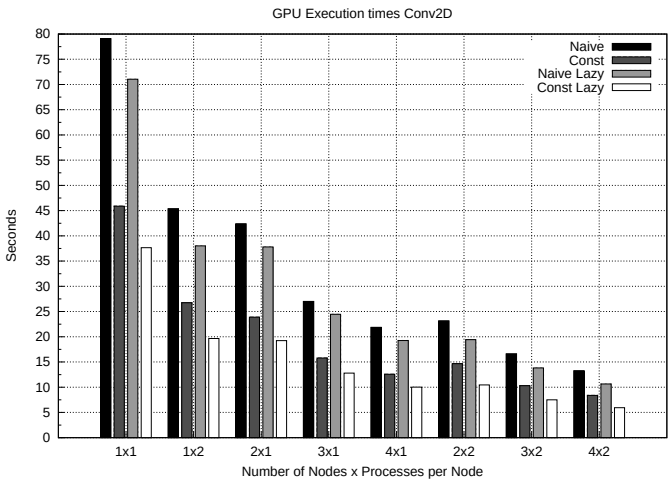


Fig. 6. The total application execution times for four different versions of the GPU-extended Conv2D application.

This is because the Conv2D algorithm uses the values in the convolution filter far more often than the ConvUV algorithm. Lazy data copying further improves performance, leading to a combined performance improvement of well over 50% for all measured cases.

The GPU-extensions to Parallel-Horus show a dramatic performance improvement in comparison with the original version of the library, even in the 1x1 case. For the naive implementation, the Conv2D application obtains a speedup of 29.1 in this configuration. For the const lazy implementation, the total execution time is even reduced by a factor of 61.2.

The speedup gained from executing on a GPU cluster compared to a traditional cluster, demonstrates why traditional clusters are extended with GPUs as accelerators. For example, and as shown in Table 2, our Conv2D application, executing on 4 nodes with 2 MPI processes per node, experiences a speedup of 387 with GPU-extensions and 7.9 without GPU-extensions. Again, these results are obtained without *any* parallelization effort from the application programmer.

	1x1	1x2	2x1	3x1	4x1	2x2	3x2	4x2
ConvUV CPU	1.0	2.0	2.0	3.0	3.9	3.9	5.7	7.5
ConvUV GPU	10.4	18.4	20.1	28.4	35.5	32.8	43.9	52.8
Conv2D CPU	1.0	2.0	2.0	3.0	4.0	4.0	6.0	7.9
Conv2D GPU	61.2	117.3	119.7	180.0	229.8	220.3	307.2	387.1

Table 2. The speedups of the ConvUV and Conv2D applications, with (const lazy) and without GPU-extensions, compared to the total execution time of the application without GPU-extensions running on a single node with one MPI process (1x1).

6 Future Work

In the near future we plan to develop further optimized versions of our CUDA kernels, especially for generalized convolution. Threads in the same thread block could use shared memory to drastically increase the compute to global memory access ratio of the compute kernel. This optimization strategy has its limits, however. Each thread block has to load an overlapping area from the original image into shared memory. When the filter size increases, the bandwidth spent on loading the overlapping areas will become larger than the bandwidth spent on the actual data being processed by the thread block.

Another limitation of this approach is due to the current GPU architecture. The amount of shared memory that is required by each thread block is likely to limit the total number of thread blocks that can concurrently run on a single multiprocessor. In Nvidia's new architecture Fermi [11], the amount of shared memory available to each multiprocessor is increased by a factor of 3.

Fermi also supports function calls within kernel functions. In the current programming model all functions are inlined at compile-time. The ability to select which device function will be executed at run-time, vastly increases programmability and sustainability of the GPU-extensions we made to Parallel-Horus. As discussed earlier in Section 2, a number of algorithmic patterns can be recognized among the operations offered by Horus. Currently, we have to implement multiple (close-to-identical) CUDA kernels for each algorithmic pattern. As an example, for the binary pixel operation we have to implement separate compute kernels for performing addition, multiplication, subtraction, etcetera. For the Fermi architecture, we will be able to write a single CUDA kernel for each pattern and have the kernel select a device function at run-time that performs the desired operation.

Further extensions to the Parallel-Horus system itself will include the creation of compute kernels for alternative special-purpose hardware architectures (e.g., FPGAs), and using other programming systems (e.g., OpenCL). Having different compute kernel implementations will allow the library to select at run-time which device will execute a particular compute kernel. Moreover, all extensions described in this paper will be integrated in a task-parallel extention to the Parallel-Horus system that is currently being developed by us as well. As soon as we have access to much larger GPU-clusters, we will also perform a much more thorough scalability analysis.

7 Conclusions

In this paper we have introduced a user transparent programming model for data parallel and hierarchical execution of MMCA applications on GPU-clusters. The programming model is obtained by extending an existing user transparent programming system for data parallel execution on traditional compute clusters. The extensions consist of CUDA-implemented compute kernels, which have been integrated with a minimal level of intrusiveness. We have demonstrated different

strategies for implementing and integrating the CUDA kernels. Among these, the most prominent strategies include the use of constant memory (where possible), and the lazy copying of data to and from GPU memory. Finally, we have evaluated the performance of a simple line detection application using different configurations for hierarchical execution. Results have shown that our model is capable of obtaining even orders-of-magnitude speed improvements, without requiring any additional effort from the application programmer.

8 Acknowledgements

We would like to thank SARA, Amsterdam, and especially Willem Vermin, for their support in this project, and for granting us access to the Lisa Cluster and its GPU add-ons. This project is performed within the realms of the ProMM-Grid project ("Programming Models for Multimedia Grid Computing"), which is funded by the Executive Board of VU University, Amsterdam.

References

1. Snoek, C., Worring, M., Geusebroek, J., Koelma, D., Seinstra, F., Smeulders, A.: The semantic pathfinder: Using an authoring metaphor for generic multimedia indexing. IEEE Trans. Pat. Anal. Mach. Intell. **28**(10) (2006) 1678–1689
2. Galizia, A., D'Agostino, D., Clematis, A.: A Grid Framework to Enable Parallel and Concurrent TMA Image Analysis. International Journal of Grid and Utility Computing **1**(3) (August 2009) 261–271
3. Morrow, P.J., et al.: Efficient implementation of a portable parallel programming model for image processing. Concur. - Pract. Exp. **11**(11) (1999) 671–685
4. Lebak, J., et al.: Parallel VSIPL++: An Open Standard Software Library for High-Performance Signal Processing. Proc. IEEE **93**(2) (February 2005) 313–330
5. Juhasz, Z., Crookes, D.: A PVM Implementation of a Portable Parallel Image Processing Library. In: Proc. EuroPVM'96, München, Germany, October 7-9. (1996) 188–196
6. Plaza, A., et al.: Commodity cluster-based parallel processing of hyperspectral imagery. J. Parallel Distrib. Comput. **66**(3) (2006) 345–358
7. Seinstra, F., Geusebroek, J., Koelma, D., Snoek, C., Worring, M., Smeulders, A.: High-Performance Distributed Image and Video Content Analysis with Parallel-Horus. IEEE Multimedia **14**(4) (2007) 64–75
8. Garland, M., et al.: Parallel computing experiences with cuda. IEEE Micro **28**(4) (2008) 13–27
9. Koelma, D., et al.: Horus C++ Reference. Technical report, Univ. Amsterdam, The Netherlands (January 2002)
10. Seinstra, F.J., Koelma, D., Bagdanov, A.D.: Finite State Machine-Based Optimization of Data Parallel Regular Domain Problems Applied in Low-Level Image Processing. IEEE Trans. Parallel Distrib. Syst. **15**(10) (2004) 865–877
11. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. 1 edn. Morgan Kaufmann (February 2010)
12. Seinstra, F., Koelma, D., Geusebroek, J.: A software architecture for user transparent parallel image processing. Parallel Computing **28**(7–8) (August 2002) 967–993
13. Geusebroek, J.M., et al.: A Minimum Cost Approach for Segmenting Networks of Lines. International Journal of Computer Vision **43**(2) (2001) 99–111