GPU-Based Image Processing Use Cases: A High-Level Approach

Volkmar WIESER [a,1], Clemens GRELCK [b], Holger SCHÖNER [a] Peter HASLINGER [a]
Karoly BOSA [c], and Bernhard MOSER [a]

[a] *Software Competence Center Hagenberg, Software Park 21, Hagenberg, Austria*
[b] *University of Amsterdam, Science Park 904, Amsterdam, Nederlands*
[c] *Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria*

Abstract. This paper addresses the gap between envisioned hardware-virtualized techniques for GPU programming and a conventional approach from the point of view of an application engineer taking software engineering aspects like maintainability, understandability and productivity, and resulting achieved gain in performance and scalability into account. This gap is discussed on the basis of use cases from the field of image processing, and illustrated by means of performance benchmarks as well as evaluations regarding software engineering productivity.

Keywords. Hardware Virtualization, Functional Programming, Software Engineering Productivity

Introduction

This paper focuses on the parallelization of image processing algorithms, as there is a growing demand for performance improvements in order to meet the required performance criteria as they are encountered for example in quality inspection systems. However, for application engineers developing such algorithms not only the performance is of interest, but also the scalability, maintainability, understandability and productivity aspects need consideration. From the methodological problem solving point of view there is an increasing complexity [2] in terms of heterogeneity of involved mathematical concepts like low-level filtering techniques with complexity known in advance and content-dependent procedures e.g. for segmentation, clustering and classification. In order to exploit the computational power of emerging parallel processing hardware components there is an increasing complexity in terms of specifying a sound and optimal execution model taking various relevant aspects like memory and thread management as well as inter-device communication into account. In order to tackle the software engineering challenges posed by these complexity aspects a high-level programming paradigm based on hardware virtualization is desirable which avoids the application engineer to be involved in hardware specific details.

The central question addressed in this paper is, what is the trade-off between the gain with respect to maintainability, understandability and productivity and performance

[1]Corresponding Author: Industrial Researcher, Softwarepark 21, 4232 Hagenberg, Austria; E-mail: volkmar.wieser@scch.at

when applying conventional versus advanced programming concepts which support hardware virtualization. This question is tackled by making a comparative study by means of two image processing related methods, and representatives for the alternative programming paradigms that is conventional based on manually optimized GPU code on the one hand, and GPU code generated by means of the high level programming language Single Assignment C (SAC, [10]).

In Section 2 we discuss hardware virtualization in general, particularly concepts relying on SAC are introduced. GPU-based image processing use cases related to preprocessing based on an anisotropic duiffusion filter [14] and classification by means of a single class support vector machine [17], [20] are introduced . For these use cases, based on the NVIDIA CUDA framework two alternative implementations related to a manually code optimization and a high-level programming approach based on SAC are presented.

Experiences and benchmarks substantiate the potential of the high-level approach even if there is still sometimes a gap in achieved performance. Above all, the drawback of the remaining performance gap is compensated by the gain of overall software engineering productivity which gives reason to use the outlined high-level approach even at the current level of development.

1. Related Work

The industry standard for programming NVidia GPUs is CUDA [7]. CUDA is a vendor-specific, architecture-specific and, hence, very low-level API. It allows the experienced programmer to adapt a program to the architectural pecularities of GPU processing and to achieve high performance if effort is not a big concern. However, software engineering on this level of abstraction is tedious and cumbersome. If CUDA marks one end of the spectrum of GPU programming, then SAC [10] marks the other. SAC programs remain completely architecture-agnostic and it is solely up to compiler and runtime system to make efficient use of GPUs where and when present [11]. An analysis of the trade-off between peformance and productivity is the subject of this paper.

In between CUDA and SAC a number of other approaches aim at facilitating GPU programming. OpenCL [18] was originally proposed by Apple and is now promoted by AMD as the only major manufacturer of both multi-core CPUs and GPUs, in particular for its upcoming Fusion architecture that will soon combine both worlds on a single chip. OpenCL is only marginally more abstract than CUDA. Programmers defines computational kernels, which can be executed on different kinds of GPUs and even on multi-core CPUs. OpenCL abstracts from concrete architectural features and instead uses a machine model that captures essential properties of today's GPU-enhanced computing systems across individual manufacturers and models. Nonetheless, programmers are concerned with a variety of machine-level details that lower productivity.

OpenMP [5] has a track record of facilitating programming of symmetric shared memory systems (multi-core, multi-processor) through compiler directives. The OpenMPC [13] project aims at generating CUDA code from eligible standard OpenMP directives. This approach is particularly attractive if application code is already equipped with OpenMP directives. Still, OpenMP is on a much lower abstraction level than SAC. We want to mention a recent proposal to extend OpenMP by clauses for the explicit placement of computations on the host or an a GPGPU [3].

Last not least, HiCuda [12] is another compiler-directive based approach to programming NVidia GPUs. It essentially imitates the OpenMP approach for symmetric multicores and proposes a tailor-made directive language for Cuda-enables GPUs. Technicall, HiCuda does indeed simplify GPU programming, but nonetheless exposes the same variety of architectural features as CUDA. Programmers need to make all relevant design decisions in application engineering, but can express them much more concise than when using vanilla Cuda.

2. Hardware Virtualization

The multicore revolution has brought an unprecedented diversity of hardware to mainstream computing. The same software is supposed to make efficient use of small numbers of powerful cores as is characteristic for current Xeon and Opteron processors as well as of large numbers of less powerful hardware threads as in the SUN/Oracle Niagara processor family. A variable number of such processors may be combined into a single server system. Graphics accelerators may be present or not on the system level. And in the near future we will face heterogeneous systems-on-chip that combine aspects of conventional multi-core processors with properties of graphics cards like the AMD fusion architecture.

Writing explicitly parallel code for each and any of these architectures for each and any relevant part of a software system in theory would yield the best possible performance, but is highly uneconomical. Furthermore, high performance is often not even achieved in practice because of unavailability of highly skilled and motivated programmers or simply time-to-market constraints that rule out necessary manual optimization.

Looking back into the history of computing one must admit that heterogeneity of computing architectures is actually not a new development of the many-core era. Many different instruction set architectures have come and gone over the years. Even within one ISA, say x86, countless variations and generations exist. However, since the early days of computing, high-level programming languages, starting with Algol, Pascal, Fortran, Lisp or C, have shielded the application programmer from such low-level details as the concrete instruction set architectures.

The trouble of programming in the multicore era is that for various reasons programmers stick to the languages that served them well in the years before. In this paper, however, we will explore ways to virtualize multicore and many-core hardware in a similar way as high-level programming languages virtualized instruction set architectures. By raising the level of abstraction in expressing program code, just as high-level programming languages did 40 years ago compared with machine specific assembly, we employ the implicitly parallel programming language SAC to abstract from concrete properties of parallel hardware. Single Assignment C (SAC) is a strict, purely functional programming language, which is well suited for array based applications like image or signal processing as well as numerically intensive computations. One of the key benefits is the combination of high-level language constructs with a similar performance of manual-optimized low-level modules. SAC is a combination of C/Matlab-style syntax and is designed in order to support high-level multi-dimensional stateless array processing. The SAC compiler automatically generates competitive code for homogeneous multi-core/multi-processor systems [9] as well as for NVidia graphics accelerators [11].

In the same way as an expert assembly programmer can often write more efficient code than what is generated by a compiler, one of the interesting questions we pursue in this paper is the price that we need to pay for higher productivity in software engineering. We are confident that in the same way as today only tiny parts of software systems are coded in assembly we will see the same for parallel software in the future.

3. Industrial Applications

In the field of industrial quality inspection for endless materials like foils or industrially woven fabrics we have to cope with noisy textured surfaces with highly complex faults phenomenology on the one hand and with a high-speed manufacturing process which defines high requirements for computer hardware and software on the other hand. The need for cost-intensive algorithms for image processing as well as machine learning is given due to the complex phenomenology of textures and defects. Therefore, in order to achieve the industrial performance criteria we have to use multi/many-core systems and high-performance computational hardware like GPUs. Furthermore, it is typically necessary to re-design the applied methods using parallelization techniques.

To demonstrate the applicability of SAC especially for rapid parallel application development, two methods of the processing pipeline (image acquisition, pre-processing, feature extraction, registration, defect detection and classification) of a visual quality inspection system are benchmarked, i.e., preprocessing using the anisotropic diffusion filter and classification using the decision function of the support vector machine.

3.1. Preprocessing with Anisotropic Diffusion

Essential factors for robust and reliable defect detection are the enhancement of defects like scratches or blowholes and at the same time the attenuation of environmental influences, e.g., irregular reflections, noise or dust. Defect enhancement is supported by the Perona-Malik anisotropic diffusion filter [14], whose principal characteristic is to reduce noise and concurrently enhance higher contrast regions.

The data-independent characteristic of the anisotropic diffusion filter allows an objective performance analysis for manually coded as well as automatically SAC generated GPU code. The benchmarking results are presented in chapter 5.

3.2. Classification with One-Class Support Vector Machine

Support vector machines are based on the concept of separating data of different classes by determining the optimal separating hyperplanes [19]. The main idea behind support vector machines - and its distinctness to other learning algorithms - is the method of *structural risk minimization*. Instead of optimizing the training error (which often leads to the problem of over-fitting), the attention is turned to the minimization of an *estimate of the test error* [16]. Due to the underlying generalization concept SVMs have become widely used learning methods which provide state-of-the art solutions for various application areas, e.g. text categorization, texture analysis, gene classification and many more.

Typically, the SVM is a supervised learning algorithm working on two classes (binary classification, see also [16]). But for industrial quality inspection, where mostly

large amounts of good samples are available and just a small fraction of possible defects are known, the application of an outlier-detection version has been proposed (one-class or single-class SVM, see [17] and [15]). The training of the one-class SVM (OC-SVM) is performed on a set of positive samples and during the classification step anomalies are detected.

Often image processing applications are time critical systems, e.g. in-line process control, where speed can be a limiting factor for usability. So the most essential part is the speed-up of the classification step, therefore a parallelization of the following decision function was considered:

$$f(\mathbf{x}) = \mathrm{sgn}(\sum_i \alpha_i k(\mathbf{x}_i, \mathbf{x}) - \rho), \tag{1}$$

where $\mathbf{x}$ is a new sample that needs to classified. The kernel function $k(.,.)$ can be seen as similarity measure between the new sample point and the support vectors $\mathbf{x}_i$ (a sub-set of the good samples from training, describing the outer sphere of of the data cluster). The parameter ρ (decision boundary) and the non-zero weights α_i (of the according support vector $\mathbf{x}_i$) are determined during the training phase.

For further details on the determination of the parameters and support vectors, and on possible kernel choices (polynomial, Gaussian radial basis function, etc.) see [16] and [6].

4. Experiences

Based on developer statements, we want to evaluate various key values, i.e., programmability, understandability, productivity, maintainability, IDE support and CPU/GPU execution time, during an application development life cycle using C, Matlab, CUDA and SAC. As a starting point C programming skills are rated as neutral to allow the comparison of the characteristics with other tools, languages and frameworks.

	programmability	understandability	productivity	maintainability	IDE support	execution time	
						CPU	GPU
C	∘	∘	∘	∘	∘	∘	∘
Matlab	+	∘	∘	+	++	−−	∘
CUDA	−	−	−−	∘	+	∘	++
SAC	∘	∘	++	++	−−	∘	+

Table 1. Pros and cons of applied tools, languages and frameworks regarding various application development aspects

Basically, each language has more or less a similar *programmability* and *understandability* because of existing assets and drawbacks in specific application fields, e.g., Matlab is a simple to use programming language but normally the developer has no knowledge about the internal optimization strategies. For CUDA development the developer needs expertise in hardware architecture and parallelization techniques. Having programming skills in C, SAC is easy to learn and provides the programming comfort of Matlab, e.g., auto-parallelization feature or no pointer arithmetic.

For rapid prototype development Matlab offers a high *productivity* but in several cases a re-design/re-implementation using a more efficient language/framework is

needed. By using SAC it is possible to auto-generate code for the mentioned platforms, especially useful if the performance criteria of a project have changed. This is also an advantage concerning *maintainability* because hardware architecture changes, e.g., updating NVIDIA GeForce 8800 to NVIDIA Fermi architecture, only requires a re-compilation.

A drawback of SAC is the lack of an integrated development environment (*IDE support*), which means that debugging, code analysis, benchmarking, etc., can only be done via the command line, whereas the other tools, languages and frameworks offer consistently well-engineered tool support.

The *execution time on CPU and GPU* depends on several influence factors, e.g., hardware environment, parallelization strategies or concurrent processes that primarily occur in industry. However, in general the SAC performance on CPU is as good as the performance of C if no optimization framework is used (e.g., OpenCV, IntelIPP, etc.). Typically, the highest performance can be achieved with manually written CUDA code (there are only some exceptions) but in some cases SAC is able to surpass manually written CUDA code due to whole program optimization and consistently optimized parallelization strategies, especially for array-based algorithms.

5. Benchmarks

The benchmarks should demonstrate the current performance of auto-parallelized CUDA code generated by SAC-CUDA and of manually optimized CUDA code. For the benchmark tests a SONY VAIO™PCG-81112M with an Intel®Core™i7-740QM Processor, 8GB RAM and a NVIDIA GeForce GT 425M graphic card is used. The operating system is Ubuntu 10.10 with installed CUDA 4.0 and SAC_1.00_17510 frameworks.

5.1. Benchmarking Anisotropic Diffusion

The dimension of the input data for the anisotropic filter range from 256×256 pixels to 4096×4096 pixels with pseudo-randomly generated 8-bit values between 0 and 255 since in this example only the dimension of the data affects the execution time.

Table 2. Benchmarking results of support vector machine of manual-coded CUDA code and SAC-CUDA produced code on NVIDIA GeForce GT 425M

image dimension	execution time		
	SAC-CUDA	CUDA	speedup
$px256 \times px256$	0.012 sec	0.040 sec	3.3×
$px512 \times px512$	0.018 sec	0.047 sec	2.6×
$px1024 \times px1024$	0.070 sec	0.075 sec	1.07×
$px2048 \times px2048$	0.260 sec	0.180 sec	0.69×
$px4096 \times px4096$	1.821 sec	0.607 sec	0.33×

The presented benchmark results in Table 2 demonstrate that by increasing the amount of data, the performance decreases. For small datasets the manually written code has a slight initialization overhead whereas for huge datasets the initialization part of the application has lower influence on the execution time. Especially for huge amounts of data the manually optimized code can prove its superiority against auto-parallelized CUDA code.

5.2. Benchmarking One-Class SVM

A comparison of the two parallelized versions of the decision function (see Equation (1)) will be shown. The manual optimized implementation of the GPU-based OC-SVM Classifier is based on a third-party *C-Support Vector Classification* implementation called *GPUSVM* [4]. For processing SVM data in parallel on GPU-devices, the applied classification algorithm employs Map Reduce [8] techniques proposed by Google as well as a GPU-vendor supplied *Basic Linear Algebra Subroutines (CUBLAS)*. The developed GPU-based OC-SVM classifier is able to read LIBSVM data format, hence, LIBSVM can be used for the training of the SVM models (and providing support vectors for it).

For the presented test results (shown in Table 3), some publicly available data sets were used from the LIBSVM data sets repository [1]. Since this data repository does not contain data sets for OC-SVMs, we took binary sets and generated training data sets with certain size (300 samples), consisting of data belonging only to one class. A simplified training with the standard settings of LIBSVM was performed, using the Gaussian RBF kernel with $\gamma = 1/n$ (where n is the number of features of the input vectors) and $\nu = 0.5$. For an explanation of these parameters see [16].

Table 3. Benchmarking results of support vector machine of manual-coded CUDA code and SAC-CUDA produced code on NVIDIA GeForce GT 425M

data sets	# of test data	# of features	# of training data	execution time		
				SAC-CUDA	CUDA	speedup
a1a	30956	123	300	0.229 sec	0.320 sec	1.39×
a9a	32561	123	300	0.243 sec	0.336 sec	1.38×
australian	690	14	300	0.190 sec	0.230 sec	1.21×
w8a	49749	300	300	0.249 sec	0.389 sec	1.56×

Table 3 shows that there is a direct relation between the number of test samples and the speed-up factor. This holds also true for the number of features each data set consists of. A straightforward conclusion is, that larger chunks of data can be better handled by the SAC-CUDA -implementation. This is explainable due to the relatively simple composition of the parallelized decision function, which is more or less just a summation of terms.

6. Conclusion

In the field of industrial quality inspection we have to cope with noisy textured surfaces and a highly complex faults phenomenology and with a high-speed manufacturing process which demands high requirements for computer hardware and software. Therefore, in this paper the language Single Assignment C (SAC) was evaluated with respect to image processing and machine learning applications concerning runtime performance and software engineering aspects. Compared to other languages, SAC has a similar syntax like C and provides the programming comfort of Matlab, e.g., auto-parallelization features and no need for pointer arithmetic. Hence, during the software engineering life cycle SAC can assist the developer to achieve efficient development of performance critical software parts. From the economical point of view, SAC provides a good balance between time of development and performance.

However, a detailed analysis of the generated SAC machine code has still to be performed. Additionally further improvements are necessary to reduce the performance gap in several cases.

References

[1] LIBSVM Data: Classification, Regression, and Multi-label. `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

[2] G. Aubert and P. Kornprobst. *Mathematical problems in image processing*. Springer, 2006.

[3] E. Ayguade, R. Badia, D. Cabrera, A. Duran, M. Gonzalez, et al. A proposal to extend the openmp tasking model for heterogeneous architectures. In M. Mueller, B. de Supinski, and B. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism, 5th International Workshop on OpenMP (IWOMP'09), Dresden, Germany*, volume 5568 of *Lecture Notes in Computer Science*, pages 154–167. Springer-Verlag, 2009.

[4] B. C. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. Technical Report UCB/EECS-2008-11, EECS Department, University of California, Berkeley, Feb 2008.

[5] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[6] N. Cristianini and J. Shawe-Taylor. *Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[7] David B. Kirk, Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[9] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[10] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[11] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Towards Compiling SaC to CUDA. In Z. Horváth and Viktória Zsók, editors, *10th Symposium on Trends in Functional Programming (TFP'09)*, pages 33–49. Intellect, 2009.

[12] T. Han and T. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), Washington, USA*, pages 52–61. ACM, 2009.

[13] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10), New Orleans, USA*. IEEE, 2010.

[14] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:629–639, 1990.

[15] B. Schölkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Comput.*, 13:1443–1471, July 2001.

[16] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond (Adaptive Computation and Machine Learning)*. The MIT Press, 2001.

[17] B. Schölkopf, R. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt. Support vector method for novelty detection. *Advances in Neural Information Processing Systems*, 12, 2000.

[18] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, and S. Tagawa. *The OpenCL Programming Book*. Fixstars, 2010.

[19] V. Vapnik. *Statistical Learning Theory*. Wiley, New York, 1998. forthcoming.

[20] R. Vert and J.-P. Vert. Consistency and convergence rates of one-class svms and related algorithms. *J. Mach. Learn. Res.*, 15:817–854, 2006.