Programs and Algorithms of Numerical Matematics 16
J. Chleboun, K. Segeth, J. Šístek, T. Vejchodský (Eds.)
Institute of Mathematics AS CR, Prague 2013

GRAPHICS CARD AS A CHEAP SUPERCOMPUTER

Jan Přikryl

Institute of Information Theory and Automation
Pod Vodárenskou věží 2, CZ-18200 Praha 8, Czech Republic
prikryl@utia.cas.cz

Abstract

The current powerful graphics cards, providing stunning real-time visual effects for computer-based entertainment, have to accommodate powerful hardware components that are able to deliver the photo-realistic simulation to the end-user. Given the vast computing power of the graphics hardware, its producers very often offer a programming interface that makes it possible to use the computational resources of the graphics processors (GPU) to more general purposes. This step gave birth to the so-called GPGPU (general-purpose GPU) processors that – if programmed correctly – are able to achieve astonishing performance in floating point operations. In this paper we will briefly overview nVidia CUDA technology and we will demonstrate a process of developing a simple GPGPU application both in the native GPGPU style and in the add-ons for Matlab (Jacket and Parallel Toolbox).

1. Introduction

While 'standard' modern CPUs provide users with growing computational power, many scientists currently migrate towards general-purpose GPU (GPGPU) applications [3], using GPUs as parallel accelerators for memory-dense, floating-point intensive, applications. An accelerated linear algebra package exploiting the hybrid computation paradigm is currently under development [8] and GPGPU accelerators are becoming a tool of choice in many computationally-bound research tasks.

The concept of a GPGPU evolved from the needs of 3D-graphics-intensive applications that dictated the design of the processor such that most transistors were dedicated to the data processing, contrary to a regular CPU. The GPUs were then designed to be able to execute data-parallel algorithms on a stream of data, and consequently, the GPGPU processors are sometimes called 'stream processors' and are (not quite correctly) considered to be representatives of the SIMD processor architecture. The currently dominant architectures for GPGPU computing are the nVidia CUDA [5] and the AMD APP (formerly ATI Stream) [1].

The intrinsic parallel structure of a GPU (see Figure 1) allows a significant speed-up in comparison to the multi-threaded single-processor architecture. The GPU programs are called *kernels* and the processor typically processes only one kernel at

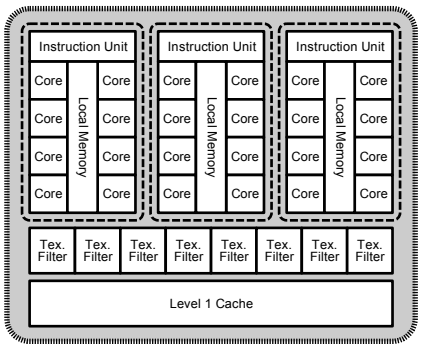

Figure 1: Thread processing cluster of a GTX280 GPU configured in 'compute mode'. The cluster contains three 8-core streaming multiprocessors, each of them has 16kB of fast local memory shared to all 8 cores. Adapted from [2].

a time by running it on several streaming multiprocessor units that form the so-called *thread block*. Every core in the GPU can access small but fast *shared* memory (local memory of a multiprocessor), large and slow *main* memory, constants can be placed to read-only and cached *constant* memory.

Although it is relatively easy to setup and perform basic operations with GPGPU even using the low-level programming (mostly ANSI C variants), it quickly becomes more complex when dealing with more demanding numerical problems – sometimes a small change in the order of instructions can have a dramatic impact on the overall performance. Additionally, special care must be taken when performing memory operations:

- due to the relatively slow memory transfer, data transfers between the host system and the GPU device shall be as few as possible, and shall be asynchronous if possible,
- improper kernel code design with respect to the operation on different memory types and ignoring memory access coalescing on the GPU device can cause a significant performance loss,
- shared memory is organised into banks and accessing elements not consecutively will cause a bank conflict.

The paper is composed as follows. The next section will introduce the covariance function, which is one of the bottlenecks of the modelling systems with Gaussian-process models. Different configurations of computation are described in Section 3, and the demonstration with a case study is described in Section 4. Conclusions are given at the end of the paper.

2. Modelling of dynamic systems with Gaussian processes

A Gaussian process [7] is a collection of random variables that have a joint multivariate Gaussian distribution. Assuming a relationship of the form $y = f(\mathbf{x})$

between an input $\mathbf{x}$ and an output y, we have $y_1, \ldots, y_n \sim \mathcal{N}(\mu(\mathbf{x}), \Sigma_{pq})$, where $\Sigma_{pq} = C(\mathbf{x}_p, \mathbf{x}_q)$ gives the covariance between the output points corresponding to the input vectors $\mathbf{x}_p$ and $\mathbf{x}_q$ and $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ denotes the multivariate Gaussian distribution with the mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$.

$C(\mathbf{x}_p, \mathbf{x}_q)$ can be any function having the property of generating a positive definite covariance matrix. A common choice is [7]

$$C(\mathbf{x}_p, \mathbf{x}_q) = v_1 \exp\left[-\frac{1}{2}\sum_{d=1}^{D} w_d (x_{dp} - x_{dq})^2\right] + \delta_{pq} v_0, \tag{1}$$

where $\boldsymbol{\Theta} = [w_1, \ldots, w_D, v_0, v_1]^T$ are the 'hyperparameters' of the covariance function, D is the dimension of the input regressors and $\delta_{pq} = 1$ if $p = q$ and 0 otherwise. The square exponential covariance function represents the smooth and continuous functional part and the constant covariance function represents the noise part, when it is presumed to be the white noise.

For a given problem, $\boldsymbol{\Theta}$ is identified using the data at hand and the function (1) is being evaluated many times before the process converges. This is one of the bottlenecks of the whole identification process of $\boldsymbol{\Theta}$ (although it is not the major one, unfortunately there are operations that can reach even $\mathcal{O}(n^3)$ [6], where n is the number of data used for identification).

3. Acceleration with various programming effort

The identification of a Gaussian-process model can be accomplished using a set of Matlab routines [4] that are an upgrade to the GPML toolbox [7] for machine learning with Gaussian processes. We will use this code base to demonstrate the process of upgrading the standard Matlab code to GPGPU code both with Jacket and Parallel Toolbox.

The code of the GPML toolbox relies heavily on linear algebra operations, which are considered to be fairly optimised even in the interpreted Matlab environment. We will therefore study the following scenarios which are ordered according to the working effort that has to be spent before actual computation:

Matlab on CPU only. We will use the native Matlab code on a multiple-core CPU. No changes are necessary.

Matlab on CPU using MEX file. We will use the original GPML MEX code on a multiple-core CPU. The publicly available ANSI C source code of a single MEX subroutine has to be compiled for the target architecture.

Matlab using Parallel Toolbox. We will use Mathworks' original interface to GPU and create our own replacement of the covariance code to compute the covariance matrix. This can be accomplished by simply retyping all GPU variables to `gpuArray`, carrying out the computation, and calling `gather` to transfer the covariance matrix back to the CPU.

Matlab using Jacket. We will update the code of the covariance routine to use the Jacket library, a third-party extension for GPU acceleration of Matlab code (see `http://www.accelereyes.com/`). We will compute the covariance matrix on a GPU using small modifications of the original GPML code: (1) all variables that will reside on GPU have to be retyped to `gdouble`, (2) we have to check that CPU and GPU variables do not occur within a single formula, and (3) the resulting covariance matrix has to be fetched back to the CPU by retyping it back to `double`.

Matlab using GPU MEX file. We will use our own replacement of covariance code to compute the covariance matrix on GPU using a hand-optimised GPU kernel. The kernel has been written in ANSI C, manually debugged and hand optimised for performance. Then a MEX file has to be created that takes care of moving data to GPU, calling the kernel and copying the result back to the CPU memory. The custom GPU kernel for the covariance function (1) relies on a coalesced memory access to move up to 16 elements of $\mathbf{x}_p$ and $\mathbf{x}_q$ to the shared memory of the thread block and computing an up to 16×16 sub-matrix of C in a single GPU kernel block. The main speedup is achieved by utilising as many kernels in a block as possible for a coalesced read of the elements from $\mathbf{x}$ into the shared memory, and by moving the elements of $\boldsymbol{\Theta}$ to the constant memory as they are used by all the invoked kernels.

In our tests, a standard PC equipped with an Intel i5/750 processor (42.56 GFLOPS in both single and double precision) and 4GB of RAM (bandwidth 17 GB/s) will be used. The GPU was nVidia GTX 275, which includes 240 processor cores (1010 GFLOPS in single, but only 124 GFLOPS in double precision; the double-precision performance is by design $8\times$ lower than that of a single-precision computation [2]) running at 1404 MHz, with the memory interface running at 1134 MHz. The board contains 896 MB of GDDR3 memory (bandwidth 127 GB/s), every processor may use up to 16 kB of fast shared memory. All computations will be carried out in Matlab R2012a in double-precision arithmetics as most current GPUs have already an unlimited support for doubles.

4. Case study

The following example demonstrates the potential of the above described scenarios for accelerating the computation of covariance function (1). We will consider computing mutual covariances of an output sequence $y[k]$ generated by

$$y[k+1] = \frac{y[k]}{1+y^2[k]} + u^3[k] + \epsilon \tag{2}$$

where ϵ is the normally-distributed white noise with $\sigma = 0.05$ that contaminates the system response and the sampling time is one second. The input signal $u[k]$ is uniformly distributed noise in the interval $[-1.5, 1.5]$ sampled every 10-th step to prevent oscillations in the system.

The comparison of the computation times for the computation of one covariance matrix as a function of the length of the $y[k]$ sequence is given in Figure 2. We

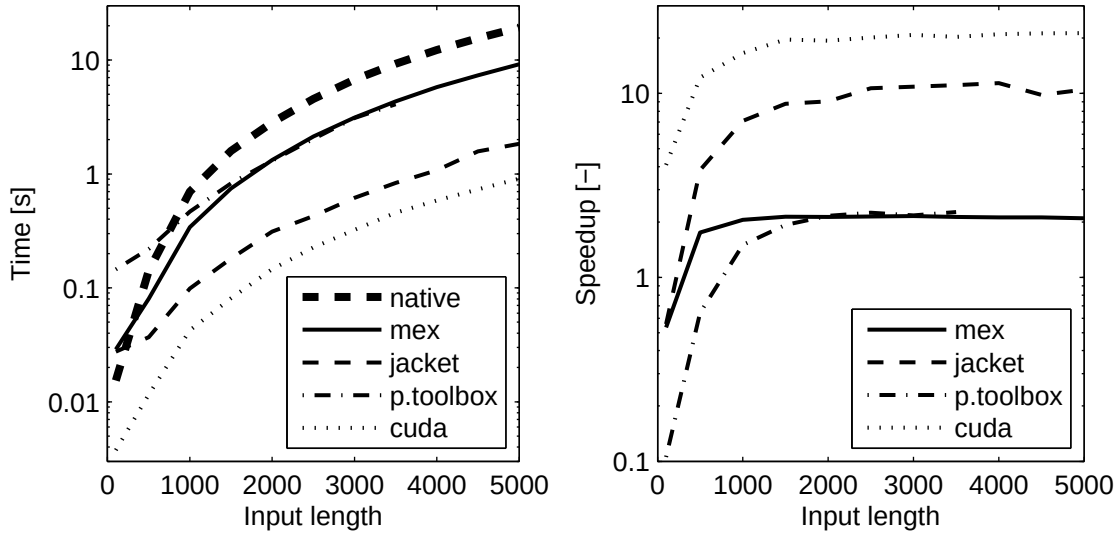

Figure 2: Computation times for the model identification versus input data dimension for different hardware configurations (left). Relative speed-ups of different hardware configurations with respect to the native CPU computation (right). Note that the GTX275 GPU has been used in the double-precision mode, where it reaches only 1/8 of the single-precision performance – hence, in a single-precision arithmetic, a GPU would be even more significantly faster than a CPU.

can see that for smaller dataset sizes below approximately 500 elements the native CPU computations may be faster than the MEX and GPGPU code, while for larger datasets the GPU-accelerated computations outperform the CPU by a factor up to 20.

The relatively poor performance for smaller input sizes is mainly due to the initialisation overhead required by the GPU and MEX code and due to the overhead of GPU data transfer (the overhead is almost 90 % of the total time for input length 100 and it is still about 30 % for input length 5000). The computation is faster on the host CPU unless this overhead can be eliminated or unless it represents a minor part of the whole computation time. Notice also the poor performance of the Parallel Toolbox code which is due to poor implementation of `repmat()` on GPU and the fact that probably due to memory leaks in the GPU code the maximum length of the input vector was 3500.

5. Conclusions

This paper provides a computational-time demonstration of how general-purpose graphics processors (GPGPU) may be used to accelerate a computation by offloading the most computational intensive parts of the code to the graphics hardware. The demonstration was performed from the user point of view to test the usability of different computational platforms for the Gaussian process model identification and

simulation. We can see that using a GPGPU computing architecture has its benefits, even in cases when the user is no expert in GPU computing: using the commercial Jacket library for Matlab or possibly Matlab Parallel Toolbox makes it possible to achieve speedup over 10 with virtually no or moderate Matlab code changes. The best results are of course provided by the hand-crafted code that has been optimised for the GPU. However, producing such a code requires a significant programming effort.

Source codes of all tested scenarios can be downloaded from `http://staff.utia.cas.cz/prikryl/panm16.zip`.

Acknowledgments

This work has been supported by the Technology Agency of the Czech Republic under project no. TA01030603 and in part by a bilateral project between Slovenia and Czech Republic no. MEB091015.

References

[1] Advanced Micro Devices, Inc., Sunnyvale, CA: *AMD Accelerated Parallel Processing OpenCL Programming Guide*, 2011.

[2] NVIDIA GeForce® GTX 200 GPU Architectural Overview. TB-04044-001, nVidia, 2008. URL `http://www.nvidia.com/object/io_1213615494642.html`.

[3] Kirk, D. B. and Hwu, W. W.: *Programming Massively Parallel Processors A Hands-on Approach*. Morgan Kaufmann, 2010, 1 edn.

[4] Kocijan, J., Ažman, K., and Grancharova, A.: The concept for Gaussian process model based system identification toolbox. In: *Proceedings of the International Conference on Computer Systems and Technologies (CompSysTech)*. Rousse, Bulgaria, 2007 pp. IIIA.23–1–IIIA.23–6.

[5] NVIDIA Corporation, Santa Clara, CA: *CUDA Programming Guide Version 2.3.1*, 2009.

[6] Quińonero-Candela, J. and Rasmussen, C. E.: A unifying view of sparse approximate Gaussian process regression. J. Mach. Learn. Res. **6** (2005), 1939–1959.

[7] Rasmussen, C.E. and Williams, C.K.I.: *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, 2006.

[8] Tomov, S., Dongarra, J., and Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput. **36** (2010), 232–240.