Przemysław MAZUREK
ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY W SZCZECINIE, KATEDRA PRZETWARZANIA SYGNAŁÓW I INŻYNIERII MULTIMEDIALNEJ,
26. Kwietnia 10, 71-126 Szczecin

Evolutionary GPGPU compilers and execution time measurements

Dr Eng. Przemysław MAZUREK

Assistant professor in the Department of Signal Processing and Multimedia Engineering at the Faculty of Electrical Engineering, West-Pomeranian University of Technology, Szczecin. Author of about 140 papers related to the digital signal processing, estimation of object kinematics, biosignals acquisition and processing.

e-mail: przemyslaw.mazurek@zut.edu.pl

Abstract

The problems of measurements of the execution time for CUDA kernels are considered in this paper. A few estimators are compared for different execution times. The proper measurements are important for code optimization using evolutionary compilers. The best estimator for fast time kernels (ms) is the minimal value estimator. The single run value is applicable for long time kernels (seconds). The disturbances of measurements are also related to the time between kernel runs, which is an unexpected result.

Keywords: compilers, GPGPU, time measurements.

Ewolucyjne kompilatory dla GPGPU i pomiar czasu wykonywania kodu

Streszczenie

Kompilatory ewolucyjne pozwalają na optymalizację kodu źródłowego i uzyskanie bardziej optymalnego (szybszego) kodu wynikowego. Wykorzystując metody optymalizacji nieliniowej możliwe jest znalezienie lepszej kombinacji instrukcji (rys. 3). Jest to istotne dla układów z nieznaną metryką wykonywania kodu. Tego typu sytuacja ma miejsce dla kart GPGPU z platformą CUDA, gdzie możliwe jest programowanie na poziomie języka C (CUDA) lub kodu pośredniego (PTX [3]) – rys. 1. Z uwagi na niemożność programowania na poziomie procesora GPU i brak informacji na temat architektury, konieczna jest optymalizacja na wyższym poziomie. W tym celu należy wykorzystać pomiary czasu wykonywania (rys. 2), jednak jest to trudne z uwagi na zakłócenia pomiaru ze strony systemu operacyjnego i innych urządzeń komputera. Sugerowany pomiar średniego czasu dla kilkunastu uruchomień jest nieadekwatny w wielu sytuacjach. O ile dla długich czasów wykonywania kodu GPGPU rzędu sekund jest to akceptowalne (rys. 6), to nie jest to prawidłowe dla krótkich czasów rzędu ms. Wykorzystując estymatory (2-5) można poszukiwać lepszego rozwiązania. Najbardziej nieoczekiwanym jest to, że istnieje silny wpływ opóźnienia między kolejnymi uruchomieniami kodu GPGPU na wynik (rys. 7). Estymator średniej jest lepszy od mediany, która potrafi fałszować wyniki, ale najlepszym jest wartość minimalna dla wielu uruchomień. Niestety także wartość minimalna zależy od czasu opóźnienia między uruchomieniami, przy błędzie 15%.

Słowa kluczowe: Kompilatory, GPGPU, Pomiary czasu.

1. Introduction

Reduction of the execution time of algorithms is of the most importance in modern computers and their applications. Many techniques, related to changes in the hardware architectures and software development tools, are used nowadays. The main limitation is the algorithm that should be implemented in the existing hardware using available software development tools. The optimization of the algorithm on its level is limited, so hardware and software implementation are most important for the processing time reduction.

Hundreds processing cores could be applied for improving computational performance of a computer system [1, 2]. Alternative approaches, like the application of FPGA processing chip, are possible, but the development of the FPGA structure is necessary and overall cost is higher. GPGPU processing is limited by many factors. The flexibility of GPGPU is moderate and only selected algorithms are well fitted. A few hundred faster computations in comparison to the single CPU are possible. Some algorithms are not easy to implement and the performance is rather poor, similar to the typical CPU.

There are many factors that limit possibilities of the efficient implementation of the GPGPU code [1, 2]. The algorithm could be not well fitted inherently, but the limitations related to the software developer are most important. The fitting of a new algorithm for the GPGPU architecture and finding a suboptimal solution is very difficult to obtain.

There many constraints related to the GPGPU, especially related to the memory accesses. A new version of GPGPUs adds new capabilities, so the optimal solution for an older version is not necessary optimal for a recent card.

The next important limitations are development tools, especially compilers. The code generation is based on the C-level language that should be fitted to the appropriate GPGPU architecture. There should be generated an assembly code for the GPGPU, but it is not possible. The intermediate code (PTX) [3] is generated instead of a true machine code. There are many reasons of this solution. The executable code, fitted into the specific GPGPU assembly code is fixed, due to ISA (Instruction Set Architecture). The intermediate code could be translated to another ISA and the forward compatibility could be obtained, even if ISA opcodes are modified. The GPGPU code is executed by a virtual machine from the programmer's point-of-view. It is a gray-box case, where the knowledge about the system is limited.

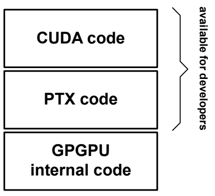
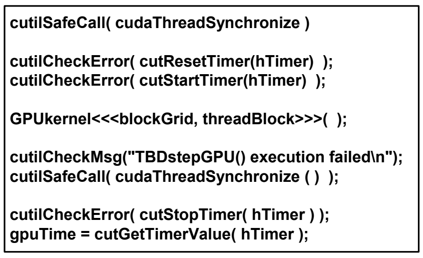


Fig. 1. Processing scheme of a CUDA kernel
Rys. 1. Schemat przetwarzania typowego jądra CUDA

Possibilities of the code optimization are limited by the lack of details about the hardware architecture and assembly language [1-3]. The run-and-measure strategy is suggested. The time measurement is the main indicator of performance. There are other indicators available, like coalescence statistics for memory accesses, but the time measurement is preferred. The typical GPGPU (CUDA platform) code uses the structure form Fig. 2.



```
cutilSafeCall( cudaThreadSynchronize )

cutilCheckError( cutResetTimer(hTimer) );
cutilCheckError( cutStartTimer(hTimer) );

GPUkernel<<<blockGrid, threadBlock>>>( );

cutilCheckMsg("TBDstepGPU() execution failed\n");
cutilSafeCall( cudaThreadSynchronize ( ) );

cutilCheckError( cutStopTimer( hTimer ) );
gpuTime = cutGetTimerValue( hTimer );
```

Fig. 2. The typical CUDA kernel call with execution time measurements
Rys. 2. Typowe wywołanie jądra CUDA wraz z pomiarami czasu wykonywania

Time measurements are based on functions that are called by the CPU, unfortunately. The scheduler of the operating system influences the measurements. The recommended measurement by NVidia is the mean value of some number of the CUDA kernel calls.

2. Evolutionary (adaptive) compilers

The trial and error method is not efficient for the code optimization if it is made by the software developer. An alternative approach is using evolutionary (adaptive) compilers [4-6]. The code is generated using the standard compiler and the source code is optimized by another compiler. This compiler implements evolutionary techniques for the code synthesis using alternative solutions and source code mixing without results influence. The time measurements are the cost function values. It is a different approach in comparison to the conventional compilers, where heuristics are commonly used.

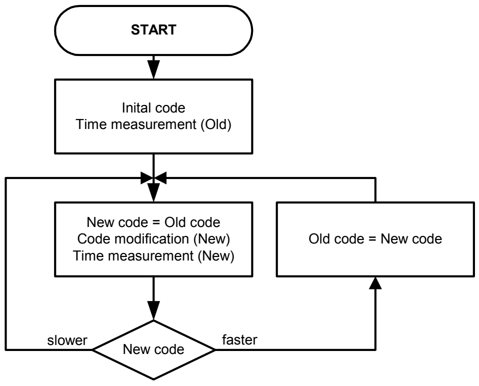


Fig. 3. Evolutionary compiler scheme
Rys. 3. Schemat kompilatora ewolucyjnego

The better solution (faster code) could be achieved after numerous iterations. It is an extremely slow technique in comparison to the single call of a conventional compiler. Better results could be obtained during hours or days depending on the algorithm, code, code variants and starting point. It is not acceptable for regular software development, but it is only one technique for the critical part of a code for the GPGPU. The processing path of the code cannot be driven by data, but it is typical for real-time applications.

3. Time measurement results

Time measurements are simple but not reliable on the CUDA platform, especially for evolutionary compilers, where the measurement error influences the convergence. The random factor is not good for local minima search, especially. Proper modifications of the code are rejected quite often, due to measurement errors.

NVidia GPGPU cards have additional limits related to the application of the single CUDA GPGPU. The same card is used as a graphic card and for computational purposes quite often. Such a card cannot be used in multi-monitor systems and the second limitation is related to the longest processing time. The CUDA processing kernel is aborted after about a few seconds, independently of the results. The recursive processing and algorithms without a specified stop moment are complicated, and multiple calls with higher granulation of the processing time and data are necessary. A solution of this problem is available and the computation using the dedicated (additional) GPGPU is necessary. One or more CUDA cards are desired in such systems. The suggested method of measurements by NVidia SDK is based on the mean value of multiple runs of kernels (1). Such a result is the reference to other techniques. This is a straightforward technique, but the real results show significant problems for this technique.

$$T_{exec} = \frac{1}{N}\sum_{i=1}^{N} T_i \tag{1}$$

An example distribution for 10000 code runs for the fixed computational cost kernel is shown in Fig. 4. The mean value is related to the distribution and it is about 2.256 [ms], but the minimal value is 2.243 [ms]. The computation cost is fixed and the unknown value is disturbed by numerous factors.

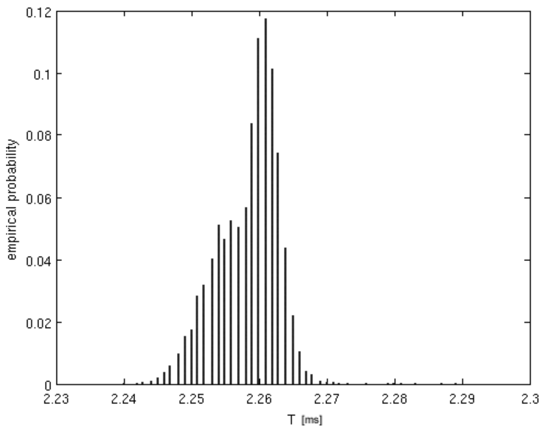


Fig. 4. Example distribution of the execution time (10000 code runs)
Rys. 4. Przykładowy rozkład czasu wykonywania (10000 uruchomień)

The Markov dependency (relation between the current and previous execution time) is depicted in Fig. 5. There are cases not depicted in Fig. 4, where the execution time is more than 6 [ms]

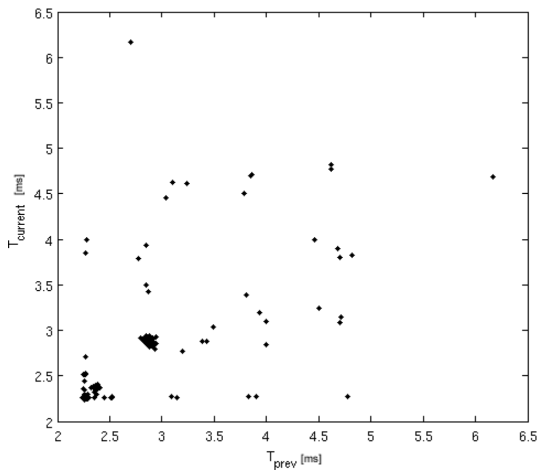


Fig. 5. Example phase diagram (10000 code runs)
Rys. 5. Przykładowy wykres fazowy (10000 uruchomień)

Some grouping of cases is visible where two tests give similar measurements. Most cases are located in the bottom-left corner and the Markov dependency for this test case is low.

There are multiple sources of disturbances related to the measurements. Interrupts related to the activity of computer devices and scheduler properties are typical. The most important disadvantage of the measurement technique (Fig. 2) is related to the possibility of the task switching between starting and stopping timer. High activity of the operating system and devices could change results and add bias to the empirical values, so the mean value (1) is not a promising estimator.

The next test is related to the execution time of the first start of the kernel. The first run is the warm-up of the CUDA and the next executions of the kernel are faster due to reusability of the initialization of the CUDA card. Kernels are started using a group of 20 code runs and the delay between two groups is set as a 2 seconds. The execution time is variable, but fixed for the group (controlled by the 'for' loop counter inside the kernel). The following estimators are verified:

$$T_{mean} = \frac{1}{N-1}\sum_{i=2}^{N} T_i \quad (2)$$

$$T_{\min} = \min(T_2, T_3, \ldots, T_N) \quad (3)$$

$$T_{median} = median(T_2, T_3, \ldots, T_N) \quad (4)$$

$$T_{\max} = \max(T_2, T_3, \ldots, T_N) \quad (5)$$

The differences between the warm-up and estimators (2-5) are shown in Fig. 6.

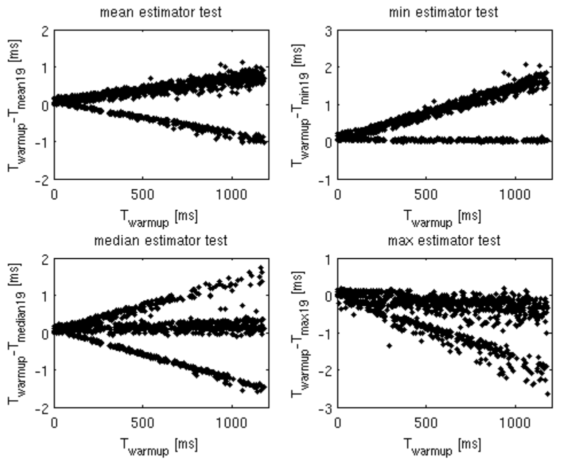


Fig. 6. Comparison of the execution time of warm-up and estimators (2-5)
Rys. 6. Porównanie czasu wykonywania dla pierwszego startu i estymatorów (2-5)

The maximal difference between the warm-up and estimator value is about 2 ms per 1 second of code execution. It is 0.2%, so the error is small for all estimators for quite long execution times (about seconds). It is sufficient for the measurement of such kernels. Consideration of the warm-up case as a part of the mean calculation is possible using the mean formula (1).

The reduced computation time kernels are harder to measure due to fluctuations (Fig. 4), because the probability of measurements of the minimal execution time is low. The effect obtained in the next test is more important in such a case.

The code has fixed the execution time and the additional and variable delay is added before kernel starting (between previous stop and next start of the timer). The all kernels from the group are started with this delay. The results are shown in Fig. 7 for the warm-up and group (19 code runs) using estimators (2-5).

The variations of the warm-up execution time are about 80% of the minimal value. Single tests for the specific warm-up delay are applied. Application of the estimators for 19 code runs should be more reliable. The delay between 0 a 2.1 seconds influences the results almost linearly. There is a gap between 2.1 and 4.3 seconds where the all estimators give the minimal value. A higher delay gives large disturbances of measurements. The maximal estimator is the reference of the worst case. The mean value estimator consists of about 40% of the worst case. The application of the median estimator gives very unreliable results. It behaves correctly up to 4.3 seconds and better in comparison to the mean estimator. Delays higher than 4.3 seconds give errors similar to the worst case.

The best results are obtained for the minimal estimator, but multiple code runs are desired. There are still errors about 15% between different delays. The similar results are obtained for a few independent tests.

4. Conclusions

The Linux OS 3.0.9, CUDA 4.0, GeForce 8600 GTS card, Pentium4D processor are used. Time measurements using CUDA platform are problematic. Long time (about seconds) measurements are possible using the mean value. Short time (milliseconds) measurements need multiple code runs (tens or hundreds tests) and the minimal value could be applied. Multiple tests increase the computation cost for a single iteration of the evolutionary compiler, unfortunately. There is a significant influence of other tasks between code runs. The evolutionary compiler may disturb measurements, adding delays before kernel start.

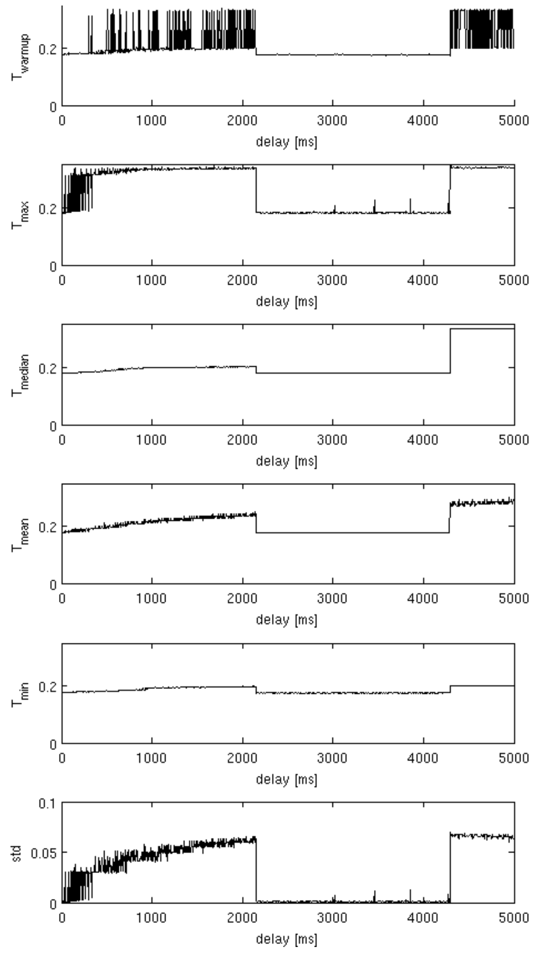


Fig. 7. Comparison of the execution time for variable delays between code runs
Rys. 7. Porównanie czasu wykonywania dla zmiennego czasu opóźnienia między kolejnymi przebiegami kodu.

This work is supported by the UE EFRR ZPORR project Z/2.32/I/1.3.1/267/05 "Szczecin University of Technology - Research and Education Center of Modern Multimedia Technologies" (Poland).

5. References

[1] NVIDIA CUDA C Programming Guide v.4.0, NVidia, 2011.
[2] NVIDIA CUDA, CUDA C Best Practices Guide v.4.0, NVidia, 2011.
[3] NVIDIA PTX: Parallel Thread Execution. ISA Version 2.3. NVIDIA, 2011.
[4] Cooper K.D., Schielke P.J., Subrarnanian D.: Optimizing for reduced code space using genetic algorithms, Proceedings of the Symposium on Languages, Compilers and Tool Support for Embedded Systems, 1999.
[5] Cooper K.D., Subramaniam D., Torczon L.: Adaptive compilers of the 21th century. Journal of Supercomputing 23, 7-22, 2002.
[6] Joseph P.J., Jacob M.T., Srikant Y.N., Vaswani K.: Statistical and Machine Learning Techniques in Compiler Design, in: Srikant Y.N., Shankar P. (eds.), The Compiler Design Handbook, Optimization and Machine Code Generation, CRC Press 2008.

otrzymano / received: 21.05.2013
przyjęto do druku / accepted: 03.07.2013 *artykuł recenzowany / revised paper*