IPSJ SIG Technical Report
Vol.2013-ARC-203 No.3
2013/1/31

# BEMAP: BEnchMark for Auto-Parallelizer

Yuri Ardila[1,a) Natsuki Kawai[1,b) Takashi Nakamura[1,c) Yosuke Tamura[1,d)

***Abstract:*** This paper presents BEMAP, a benchmark suite to measure an auto-parallelizer tool. BEMAP aims to assist a development of an auto-parallelizer tool by comparing the code compiled using it, to the corresponding hand-tuned parallelized OpenCL program. It is an open-source project, and the documentations on code explanations and experimental results of the hand-tuned parallelized programs are also provided.

***Keywords:*** parallel programming, benchmark, auto-parallelizer, OpenCL

## 1. Introduction

Major rapid development in multicore platforms recently proliferates the efforts in the parallel computing world. For instance NVIDIA has developed three GPGPU archictures in the last six years, namely Tesla, Ferm, and Kepler. Intel recently has also released a powerful coprocessor, Xeon Phi$^{\text{TM}}$. Many processor vendors are racing on the market for the highest FLOPS (Floating-point Operations Per Second) while maintaining the energy-efficiency and low price.

The improvements of multicore processors have forced software developers to be more subtle in retrieving the most efficient outcome of their programs. In order to acquire that, the program flow must be designed so that each processing element contributes equally and seamlessly to the computations, while maintaining the consistency of resources.

Traditionally, processors were only developed with only one core attached, which means those processors are only capable of calculating a consecutive and ordered set of instructions. Designing a sequential program is the elementary level of programming and very simple to debug. In contrast, a parallel program needs a tedious error-checking and the debugging process can be extremely complex, since the execution scheduling process may change over time.

The convertion of sequential program into multi-threaded or vectorized (or both) code is usually where time mostly used, albeit parallel programming APIs have become handier to use, and programmers can use language they prefer to port the code. For example, OpenCL, a parallel programming API aimed at easing code porting across different hardware platforms, has had its runtime library available for C, C++, Java, and Python.

However, the manual parallelization process tends to be time-consuming and error-prone. In the past several decades, an automatic parallelizer, which is a tool that implies automation in the convertion process, has been studied scrupulously by compiler researchers. Up to this point, no robust auto-parallelizer has been achieved. Due to the inherent difficulties in building a fully automated parallelizer, several semi-automatic parallelizer exist which gives the obligation to the programmers to inject some "hints" to the program in the form of preprocessor directives, such as OpenACC and OpenHMPP.

When developing an auto-parallelizer, a benchmark to run standard tests and trials is compulsory to investigate the credibility of the implemented parallelization analysis by comparing to a manually parallelized program using an existing API.

In this paper, we introduce BEMAP (BEnchmark for Auto-Parallelizer), which is a collection of sequential and manually parallelized programs, aimed at supporting the development of auto-parallelizer tool.

This paper is organized as follows. Section 2 describes BEMAP's overview, and experimental results of BEMAP's hand-tuned parallel programs on a certain platform, and Section 3 concludes the whole paper.

## 2. BEMAP Overview

BEMAP is a benchmark for parallelizer consisting of reference codes (C/C++ single thread) implementation and hand-tuned parallel codes. This is an open-source project and the software can be downloaded from. The main purposes of BEMAP benchmark suite are mainly:

- To investigate the work or the outcome of a compiler designed for a specific multi/many core platform by comparing the results (runtime, output, etc.) between compiled codes retrieved by compiling the reference code and the hand-tuned parallelized codes
- To identify performance bottlenecks, and to help finding several potential solutions for a particular problem to be executed in a particular platform. This applies for for both software's and hardware's point of view

The hand-tuned parallel codes are currently implemented using the OpenCL framework. Rising aside with the heterogeneous

1 Fixstars Corporation, Tokyo, Japan
a) y_ardila@fixstars.com
b) kawai@fixstars.com
c) nakamura@fixstars.com
d) tamura@fixstars.com

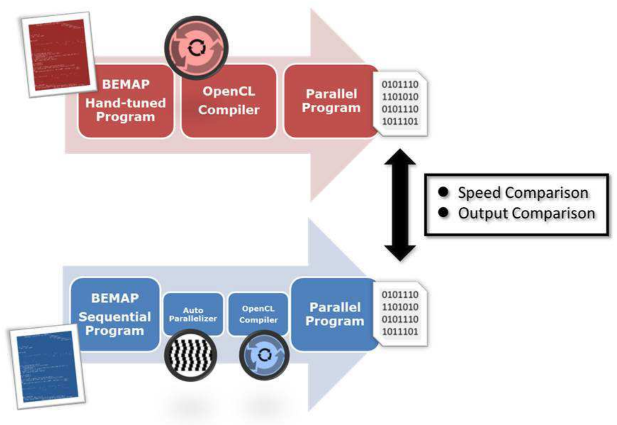

**Fig. 1** BEMAP Usage Flow

computing, the OpenCL framework gives a standardized parallel programming API, and aims to provide the base of parallel computing paradigm to simplify cross-platform code porting. Moreover, OpenCL has both C and C++ API interfaces (but not for the kernel code) in which most programmers are familiar with, and recently frontends in other language wrappers are also in progress, such as Python (PyOpenCL ) and Java (jocl and javacl ).

Big processor vendors such as Intel, NVIDIA, AMD, IBM, and ARM have released their own compiler for the OpenCL language, in which each of them is provided with its own way to optimize the code. In BEMAP, we strictly focus on optimizing reference codes for CPUs with Streaming SIMD Extensions (SSE) and NVIDIA GPUs.

Streaming SIMD Extensions (SSE) architecture is equipped with vector registers that can store a group of data elements concurrently (XMM 128-bit registers); for instance, float4 preserves 4 32-bit floats in one XMM register. With the use of these vector registers, a considerable speedup can be achieved by calculating two packed up 128-bit registers in a simultaneous execution. This kind of execution is commonly known as an SIMD execution.

NVIDIA's GPUs are optimized in a different way to conduct a parallel execution. These GPUs are not equipped with wide registers, instead they are fully covered by a bunch of powerful streaming multiprocessors (SMs). Therefore, to gain a maximum exploitation of the architecture, the amount of work has to be divided into a small unit of execution. Also, these GPUs do not have SIMD registers, they must load the data one-by-one from the global memory. The overhead of memory transfers caused by load-store from global memory may be a choking bottleneck for the whole program runtime. Fortunately, for this kind of case, a shared memory space for threadblock is provided. The transfer rate is incredibly fast compared to that of global memory space, and for a chunk of data that is oftenly loaded several times, it is more subtle to use this memory space.

## 2.1 Backprojection (BP)

Backprojection is an image enhancing/transforming technique where the image is brought to a higher dimensional space by using the predefined projectors' variables, such as coordinates and weights. In BEMAP, the Backprojection problem is used to solve a problem listed as one in the ACM Programming Contest, South Pacific Regionals 2000 . The problem detail and solution explanation can be found in. The computational complexity of Backprojection in the mentioned problem is $O(n^3)$.

The implementation of Backprojection in OpenCL consists of three kernels, each of which has different implementation for both CPUs and GPUs. The implementation for GPUs maintains a coalesced access memory, and also maximize the usage of shared memory. The implementation for CPUs use the SIMD instructions with the vector registers, and the amount of computation is heavier the GPU kernel. Also, to reduce the overhead caused by memory transfer, a memory mapping technique is also implemented. The comparison of calculation speed can be found in Table 2.

## 2.2 Black-Scholes for European Call-Put Option (BS)

Black-Scholes formula is a mathematical model used in a pricing of complex financial instruments; for instance, the one that exists in the European call-put option equation. The computational complexity for this formula is $O(n)$ and it takes the following inputs: a set of stock prices for the underlying asset ($S[]$), a set of strike price of the option ($K[]$), a set of time of maturity for the corresponding strike price ($\Delta t[]$), risk-free rate ($r$), and volatility of returns of the underlying asset ($\sigma$).

For the SIMD kernel, we use the XMM registers and execute 4 elements at once. Since CPU has powerful cores but not many threads execution, the weight of calculation is heavier than that of GPU's kernel. As for the scalar kernel, the implementation is very simple where each thread calculates the element one by one. The comparison of calculation speed can be found in Table 2.

## 2.3 Gaussian Blur (GB)

Gaussian Blur is a commonly used filter for image processing, especially to reduce the image noise. The resulting blurred image is retrieved by applying the two dimensional Gaussian Filter with a predefined standard deviation value ($\sigma$). In BEMAP, two ways to conduct Gaussian Blur are covered: a 4-nested loops method with a computational complexity of $O(w_{filter}h_{filter}w_{image}h_{image})$ , and the convolution separable method with $O(w_{filter}w_{image}h_{image}) + O(h_{filter}w_{image}h_{image})$.

The implementation of OpenCL kernels for GB is divided into four kinds: Scalar, SIMD, Scalar Fast, and SIMD Fast. The Scalar mode runs the Gaussian Blur with two-dimensional circular kernel in parallel by calculating each pixel in one workitem (thread), hence the number of workitems is equal to $w_{image}h_{image}$). The SIMD mode is executed by calculating one image row per workitem. This kind of implementation will only bring disadvantagesfor GPUs since executing less instructions with more workitems is better for them. The Scalar Fast and SIMD Fast mode execute with the same logic, and the only difference is that they are divided into two kernels: one for the row-wise calculation, and the other for column-wise calculation. From bemap-5.1 , we added another mode for GPUs, which is to execute the Convolution Separable method using the shared memory space. The comparison of calculation speed can be found in Table 2.

### 2.4 Grayscale (GS)

Grayscale (or Greyscale) is a method to convert a color image (or RGB image) to a monochromatic image. The algorithm is done by taking a weighted value of the red, green, and blue part of the image. It weighs green pixels more than the other two, since human's eyes are more sensitive to them. The computational complexity of Grayscale is $O(w_{image}h_{image})$.

Grayscale's kernel implementation is very simple. Basically we only need three multiplications and additions for each pixel and one time memory load; therefore shared memory usage will not give more speedup per se. For the scalar kernel, each workitem calculates one pixel, and the SIMD kernel calculates one image row per workitem. The comparison of calculation speed can be found in Table 2.

### 2.5 Linear Search (LS)

The Linear Search algorithm checks all $n$ elements one by one until a desired value is found which returns a sucess, or the opposite which returns a fail run. This algorithm is the most naive and simples implementation of a search algorithm, for it is a brute-force algorithm. Linear search computational complexity completely depends on the number of elements ($O(n)$).

To investigate memory latency caused by serialization due to bank conflicts, we demonstrate two kidns of implementations in this paper: Scalar and STMD. The Scalar implementation is a straight-forward implementation, where each workitem examines one item, wheter it meets the desired value or not. If it does, then that particular workitem writes its global ID to a single integer memory space with an atomic function. Note here that since two or more workitems may write concurrently to the same memory space, triggering a condition called a bank conflict, we have to use an atomic function to avoid an unwanted memory overwrite.

The STMD (Single Thread Multiple Data) implementation gives more work to a workitem, where each of them examines several elements. This implementation may seem weird since it executes more instructions for a workitem, with no explicit use of a vector data type register; however, throughout hypothesis and experiments, apparently it is more subtle and works better for CPU because it reduces the bottleneck caused by bank conflicts.

For both implementations, we run the kernel by a number of iterations to demonstrate the advantage of memory mapping technique in OpenLCL, where in each iteration, the result has to be transferred back to the host before the next iteration begins.

The comparison of calculation speed can be found in Table 2.

### 2.6 Monte-Carlo for Finance (MC)

The Monte-Carlo methods are a class of simulation algorithm which take the advantage of a (pseudo) random number sequence generator. In BEMAP, the Monte-Carlo method is used to simulate a finance pricing model, which is similar to that of Black-Scholes workload's problem. With the number of inputs of $n$, and the number of paths (random numbers) of $p$, the computational complexity of Monte-Carlo is $O(np)$.

**Table 2** Benchmark Kernel Peak Performance [ms]

| Workload | Intel Core i7-3770k | Intel Core 990X | GeForce GTX 680 | GeForce GTX 570 |
|---|---|---|---|---|
| GS (Ref) | 4.77 | 5.44 | | |
| (OCL) | 0.42 | 0.45 | 0.07 | 0.05 |
| BS (Ref) | 1556.96 | 6100.41 | | |
| (OCL) | 21.73 | 42.36 | 1.45 | 1.70 |
| LS (Ref) | 29.90 | 24.52 | | |
| (OCL) | 16.78 | 7.02 | 24.18 | 47.00 |
| GB (Ref) | 25.51 | 30.08 | | |
| (OCL) | 1.13 | 1.23 | 0.15 | 0.19 |
| BP (Ref) | 11881.17 | 11297.62 | | |
| (OCL) | 123.44 | 196.85 | 71.26 | 75.86 |
| MC (Ref) | 23944.01 | 193803.17 | | |
| (OCL) | 269.56 | 671.17 | 41.69 | 49 |

The implementation of the program in OpenCL is divided into three kernels: Scalar with no shared memory, Scalar with shared memory, and SIMD. The scalar version with no shared memory runs the formula in the traditional way of parallelization; which is to run the formula for one call option in each thread hence finding the sum for all paths is a gruesome bottleneck. The one using shared memory also executes with this algorithm, but uses a parallel reduction technique when finding the sum. The SIMD kernel uses the similar technique with the shared memory version, except that this kernel takes the advantage of vector data type instead of shared memory, since there is no such architecture in CPUs. The comparison of calculation speed can be found in Table 2.

## 3. Conclusion

We have introduced our tool, BEMAP, which is a benchmark suite to aid the development of an auto-parallelizer to measure its performance, and in the same time can also help to measure the performance of the targeted machine.

### References

[1] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, T. D. Uram, "GROPHECY: GPU performance projection from CPU code skeletons" *SC '11*, 2011.

[2] S. Hong, H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness" *ISCA '09*, 2009.

[3] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide Version 4.2*

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph" *ACM Transactions on Programming Languages and Systems Volume 13 Issue 4*, October 1991

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, October, 2009.

[6] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, J. S. Vetter, "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite," In *Proceedings of the Third Workshop on General-Purpose Computation on Graphic Processors (GPGPU 2010)*, March, 2010.

[7] BEMAP: BEnchmarks for Automatic Parallelizer. https://sourceforge.net/projects/bemap/

[8] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," June, 2011.

[9] R. Hochberg, "Matrix Multiplication with CUDA – A basic introduction to the CUDA programming model," August, 2012.

[10] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, A. W. Toga, "CUDA Optmization Strategies for Compute and Memory-Bound Neuroimaging Algorithms," June, 2010.

[11] S. Kawai, "'Program Promenade: Digital Tomography," In *IPSJ Mag-*

**Table 1** Testing Hardware Specifications

| Hardware Type | Intel Core i7-3770k | Intel Core 990X | GeForce GTX 680 | GeForce GTX 570 |
|---|---|---|---|---|
| Microarchitecture | Ivy Bridge | Nehalem | Kepler | Fermi |
| Core Frequency (GHz) | 3.5 | 3.47 | 1.058 | 1.46 |
| Number of Cores | 4 | 6 | 1536 | 480 |
| GFLOPs per core | 28 | 13.88 | 2.116 | 2.92 |
| Device peak GFLOPs | 112 | 83.28 | 3250.176 | 1401.6 |
| L1-cache size (KB) | 64 | 64 | 64 | 64 |
| L2-cache size (KB) | 256 | 256 | 512 | 512 |
| L3-cache size (MB) | 8 | 12 | - | - |
| Memory type | DDR3-1600 | DDR3-1066 | GDDR5 | GDDR5 |
| Memory clock rate (GHz) | 1.6 | 1.066 | 6.008 | 3.8 |
| Memory bus width (bit) | 128 (dual) | 192 (tripple) | 256 | 320 |
| Memory peak bandwidth (GB/s) | 25.6 | 25.6 | 192.256 | 152 |

*azine*, Vol. 46, No.3, Mar, 2005.
[12] F. Black and M. Scholes, “The Pricing of Options and Corporate Liabilities,” In the *Journal of Political Economy*, Vol. 81, No. 3 (May - Jun., 1973), pp. 637-654, The University of Chicago Press, 1973.
[13] B. Lu, “Monte Carlo Simulations and Option Pricing,” Pennsylvania State University, July, 2011.

**Acknowledgments** This project is partially funded by the Department of the New Energy and Industrial Technology Development Organization (NEDO).