2010 International Symposium on Nonlinear Theory and its Applications
NOLTA2010, Krakow, Poland, September 5-8, 2010

# Standard C++ Compiling to GPU with Lambda Functions

Ádám Rák† and Gergely Feldhoffer†‡ and Gergely Balázs Soós‡ and György Cserey†∗

†Faculty of Information Technology, Pázmány Péter Catholic University,
Budapest, Práter u. 50/a. 1083, Hungary
‡StreamNovation Ltd. Budapest, Práter u. 50/a. 1083, Hungary
∗ Infobionic and Neurobiological Plasticity Research Group,
Hungarian Academy of Sciences, Pázmány Péter Catholic University, Semmelweis University,
Budapest Práter u. 50/a, H-1083, Hungary
Email: [rakad,flugi,soos,cserey]@itk.ppke.hu

**Abstract**—In this paper, a new method of a compiler application to GPU is introduced. In this method, a hybrid executable is generated from the C++ lambda function based code. Our compiler plugin creates GPU accelerated subroutines from code using our library. A C++ runtime library is designed embedding the generated GPU code into the original project.

## 1. Introduction

New generation hardwares contains more and more processors and the trends show that these numbers will intensely increase in the future. The question is how could we program these systems and may we port earlier codes on them? There is a huge need for this today as well as in the forthcoming period. Our new approach of the automation of software development may change the future techniques of computing science.

Exploiting the advantages of the new architectures needs algorithm porting which practically means the complete redesign of the algorithms. New parallel architectures can be reached by "specialized" languages (CUDA, OpenCL, Verilog, VHDL, etc.), for successful implementation, programmers must know the fine details of the architecture. After a twenty years long evolution, efficient compiling for CPU does not need detailed knowledge about the architecture, the compiler can do most of the optimalizations. Can we develop as efficient GPU (or other parallel architecture) compilers as the CPU ones? Will it be a two decade long development period again or can we make it in less time?

The specification of a problem describes a relationship from the input to the output. The most explicit and precise specification can be a working platform independent reference implementation which actually transforms the input from the output. Consequently, we can see the (mostly) platform independent implementation, as a specification of the problem.

Parallelization must preserve the behaviour in the aspect of specification to give the equivalent results, and should modify the behaviour concerning the method of the implementation. Automated hardware utilization has to separate the source code (specification) and optimization techniques on parallel architectures.

There are different trends and technical standards emerging. Without the claim of completeness, the most significant contributions are the following: OpenMP [1] - supports multi-platform shared-memory parallel programming in C/C++ and Fortran, practically it uses pragmas for existing codes. OpenCL [2] - is an open, standard C-language extention for the parallel programming of heterogeneous systems, also handling memory hierarchy. Threading Building Blocks of Intel [3] - is a useful optimized block library for shared memory CPUs, which does not support automation. One of the automation supported solution providers is the PGI Accelerator Compiler [4] of The Portland Group Inc. but it does not support C++. There are problem-software or language specific implementations on many-core architectures, one of them is a GPU boosted software platform under Matlab, called AccelerEyes' Jacket [5]. Overviewing the growing area, there are successful partialy solutions, but there is no universal product and still there are a lot of open problems.

Our aim is machine learning boosted OpenCL parallelization of any standard C++ source code by separating programming and parallelization meta-programming. This presentation shows that the basic technological problems (OpenCL source code generation, host code generation and insertion) are manageable: a C++ library is introduced, which can be compiled with every C++0x standard [6] compatible compiler, and produces CPU code. Our compiler plugin and C++ library creates GPU accelerated executables. This approach is methodically one step after the Intel Thread Building Blocks, because the parallelization schemes and memory access patterns are still fixed and provided by our library, but the building blocks themselves become completely user defined in the form of lambda functions.

This paper is organized as follows. After the introduction, in section 2 a general overview of the architecture of the new generation GPUs is given. The lambda functions in the new standard C++ are depicted in section 3. In section 4 we introduce our Minotaurus project which is a gcc based

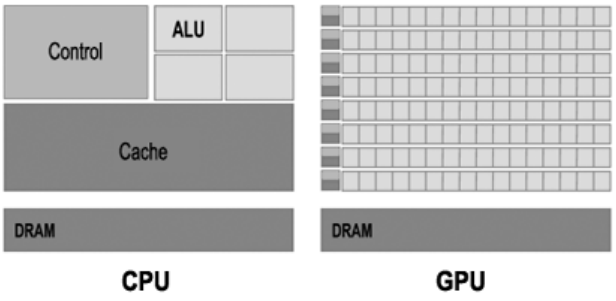



Figure 1: In the case of a GPU, most of the parts of a normal CPU are sacrificed to placethe maximum amount of processing units on the chip. In most cases, one core is completely reduced to a simple 32bit FPU / ALU pair, and many cores use the same execution control units on the chip.

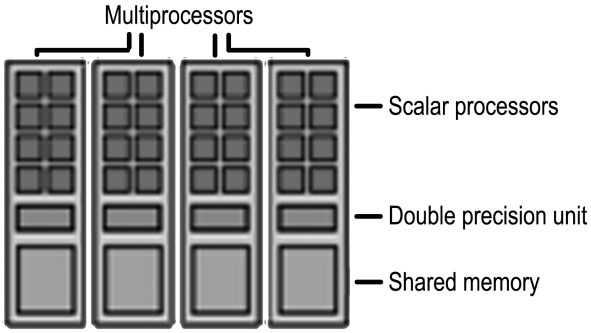



Figure 2: NVidia GPU architecture usually contains 30 Streaming Multiprocessors (SM), where each SM contains 8 scalar processors, 1 double precision unit, 2 special function units, 16K spared memory and 64K registers.

C++ compiler plugin. Results and working demonstrations are presented in section 5.

## 2. GPU architecture

Complex real-time 3D rendering needs considerable computing power, orders of magnitude greater than what one CPU can provide. But fortunately the algorithms are all data-parallel, which means that the same code must be executed on all the threads, just the processed data is different. These requirements gave rise to the massively SIMD parallel GPU architectures nowadays. Most of the parts of a normal CPU are sacrifised to place the maximum amount of processing units on the chip. In most cases one core is completely reduced to a simple 32bit FPU / ALU pair, and many cores use the same execution control units on the chip. The pipelines are generally very deep, further allowing more optimization of the architecture. While CPUs need serious trickery, both in hardware (branch prediction, instruction reordering) and sometimes in software too (compilers) to deal with deep pipelines, GPUs do not need this because rendering specific algorithms utilize massively huge amount of threads, much more than the number of cores, which makes it very easy to fill the pipelines. This is possible because every thread runs independently on different data, so there is no dependency between them, so on every core, on every pipeline stage a different thread can be executed. The scheduling of threads is done in the hardware to reduce the overhead.

OpenCL provides us an abstraction of the massively parallel hardwares, where both the computing resources (cores) and the memory is hierarchial. This approach was introduced by the hardware manufacturers and it seems that the multicore industry is heading this way. It is suspected [7] that currently this is the optimal trade-off between programmability and performance, where the highest performance is represented by the FPGAs (Field Programmable Gate Array) where everything is parallel, and the maximal programmability by the single core CPU with a single thread running. An important feature is that the memory can be accessed in 1D, 2D or 3D topography, accelerated by the 2D aware hardware caching, and the virtual indexing of the threads can also follow this scheme.

## 3. Lambda functions in C++

The use of ”lambda” originates from functional programming and lambda calculus, where a lambda abstraction defines an unnamed function. In the new standard of C++ (known also as C++0x) the syntactic element of lambda function is introduced to improve functor usability in templates. The lambda function is an inline expression of a functor object. It is nameless, only a few syntactic units can be given: the captured variables, the parameters, the type of the returning value, and the function body. The created functor will have the captured variables as members, and the constructor will assign the values. The operator() will be created with the parameters and the given function body. The function body is limited to use local variables, the parameters, and the captured variables. There is a convenient way to capture all of the local stack variables in the context as well.

Lambda functions are designed to be used where functors are passed but there is no need to reuse the functor class anywhere else, and building a whole class in order to fulfill syntactic requirements for only one use is circuitous.

## 4. Minotaurus project

The flowchart in Figure 3 shows the main components of the compiler using our plugin called Minotaurus. As usually, the gcc compiler has three parts, a frontend, a middle end and the backend. Minotaurus connects to the middle end using the inner representation of a gcc compiler besides the GPU application output, generating an OpenCL

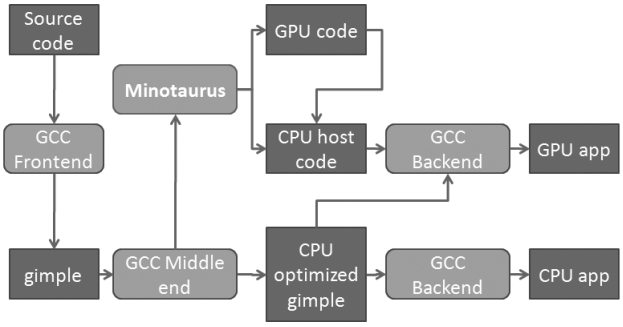



Figure 3: This flowchart shows the main components of the compiler using our plugin called Minotaurus. As usually, the gcc compiler has three parts, a frontend, a middle end and the backend. Minotaurus connects to the middle end using the inner representation of a gcc compiler besides the GPU application output, generating an OpenCL code based GPU-accelerated application output.

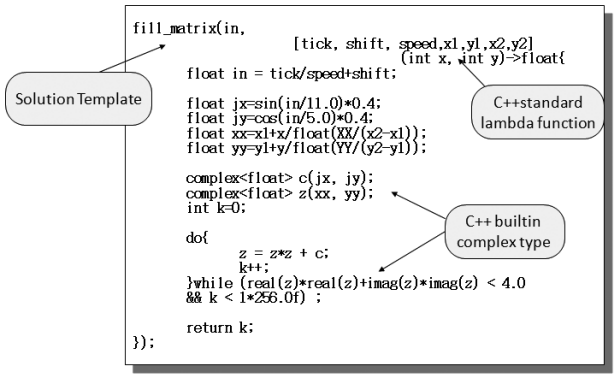



Figure 4: This C++ code demonstrates the usage of the lambda function in our system. The fill_matrix() function works as a solution template. The solution template contains hardware specific parallelization schemes and the memory access patterns. The lambda function is defined as a parameter of the function. The implementation of the algorithm is coded by the lambda function. In this given case, it generates a julia-set demonstrating the exploitation of the C++ advantages.

code based GPU-accelerated application output.

Simplified problem: the programmer specifies the code parts that can be run efficiently on GPU in lambda functions. Minotaurus compiles the lambda function to extract the data and control flow, and synthesize the OpenCL source code which is semantically equivalent to the lambda function. The programmer picks a template function to express the pattern of use (scan primitive for example), and gives the input data. The template function contains the host code which feeds the GPU kernel function. This is where we are standing right now.

### 4.1. Standard C++ code input

With Minotaurus, it is possible to compile CPU only executable and CPU-GPU mixed executable as well, using only standard C++ language elements in the common source code. This is useful if the debugging process is more complicated with GPU codes, which is usually far more complicated indeed. There are small differences between the resulting executables concerning floating point precision for example, but the theoretic correctness of the implemented method can be checked.

### 4.2. Using lambda functions to specify kernels

The fundamental benefit of using lambda functions for compiling to CPU-GPU mixed executable is the clear separation of function and parameter data. For the compilation of the general code, the compiler must explore the full data dependency of the given function to transport the required data to the GPU platform before code execution. This data dependency can be hard to follow because of reading global variables, pointers to globals, etc. Lambda functions are closed in this term, besides local variables, only the captured values and the parameters are accepted inside the function. These variables are given explicitly so any template function can handle the memory transfer to the GPU, so the data dependency of the GPU targeted code can be satisfied.

### 4.3. Automatic code generation

The function body of the lambda function may contain elements of C++, such as complex<> type, or references. Minotaurus can convert these elements to a semantically equivalent OpenCL code, using pointers instead of references for example. Lambda functions will be converted to OpenCL functions.

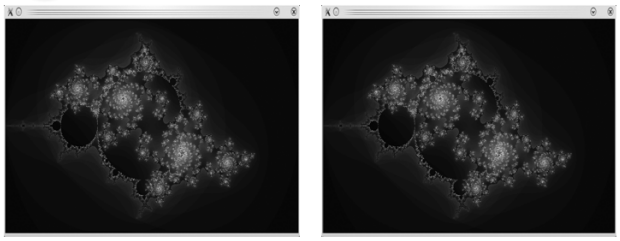
The host code is also generated, the memory transfer can be handled based on the lambda functions' members. The converted lambda functions are called from generated kernel functions based on the parameter set of the lambda function. The host code copies the actual data to the kernel functions, enqueues the kernel, and reads GPU memory to the returning variables.

### 4.4. Extendable technology to other languages

Since Minotaurus works in the inner representation of the compiler, most of the functionality does not depend on the input language.

### 4.5. Towards automatic GPU code generation

Some of the hard work is done on Minotaurus right now, but there is plenty of work ahead. Automatic data dependency exploration of any code segment is required for general CPU-GPU hybrid compilation. Functional dependency and guaranteed traces of control are needed in order



CPU GPU

Figure 5: These images show that the different CPU and GPU accelerated application output generates the same image result, but the GPU version has a significant 25-times performance speed-up.

to select a section of the code which can be compiled to GPU function (entry point of the generated function and the returning points - it is trivial in lambda functions).

### 4.6. GPU code generation is functional

We can now compile C/C++ code to OpenCL functions and host code, so we are able to create a hybrid executable from purely C++ code. The performance gain is heavily task dependent, it can be even 80x speed-up. This is notable since the conversion is purely mechanical, no additional tweaking is done with OpenCL local variables, and other sophisticated techniques yet.

## 5. Results and demonstrations

Figure 4. demonstrates the usage of the lambda function in our system. The fill_matrix() function works as a solution template. The solution template contains hardware specific parallelization schemes and the memory access patterns. The lambda function is defined as a parameter of the function. The implementation of the algorithm is coded by the lambda function. In this given case, it generates a julia-set demonstrating the exploitation of the C++ advantages (see Figure 5.).

The GPU accelerated version reached up to 25x performance gain on the same source code, utilizing the parallel GP-GPU technology (NVIDIA GTX 280) compared to the OpenMP Intel i7 4 cores implementation. This approach provides C++ support in the kernel code and shows a proof of concept for automatic GPU code (OpenCL) generation.


## Acknowledgement

The Operational Program for Economic Competitiveness (GVOP KMA), the support of NVIDIA Professor Partnership Program and the Bolyai János Research Scholarship is gratefully acknowledged. The authors are also grateful to Professor Tamás Roska for discussions, his suggestions and his never ending patience.


## References


[1] L. Dagum, R. Menon, and S. Inc, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[2] A. Munshi, "The OpenCL specification version 1.0," *Khronos OpenCL Working Group*, 2009.

[3] J. Reinders, "Intel threading building blocks," 2007.

[4] M. Wolfe, "Implementing the PGI Accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 43–50, ACM, 2010.

[5] AccelerEyes, "Jacket: a GPU engine for MATLAB," 2009.

[6] P. Becker, "Working draft, standard for programming language C++," *ISO/IEC, Tech. Rep*, vol. 2798, 2009.

[7] K. Hawick, A. Leist, and D. Playne, "Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software," tech. rep., Technical Report CSTN-091, Computer Science, Massey University, 2009.