Loop2GPU: Transforming Loops to OpenCL Kernels as a LLVM Pass

Semih Okur, Cosmin Radoi
Computer Science Department
University of Illinois at Urbana Champaign
{cos, okur2}@illinois.edu

May 14, 2012

Abstract

Lately, programmers have started to take advantage of the GPU capabilities of their systems. Still, programming for the GPU can be very hard. We are trying to hide some of this complexity from the programmer by making the compiler automatically transform embarrassingly parallel loops to GPU kernels. To this end, we have implemented a compiler pass that transforms simple loops to OpenCL kernels.

1 Introduction

In the quest for better performance, GPGPUs are becoming an increasingly popular solution. But taking advantage of the GPGPUs involves using, and hence learning first, either a specialized subset of C99 or rather complex language binding libraries. We explore the possibility of unloading this burden from the user to the compiler. As such, the user could write normal-looking code that would then be transformed by the compiler to use OpenCL. A first step in this direction would be replacing small, expensive computations with OpenCL kernels. This could be done at the LLVM IR level, completely shielding the user from the extra complexity. As the OpenCL language is a rather restricted subset of C99, the transformation is not straightforward.

We have implemented a compiler pass that converts pre-selected loops to OpenCL kernel code in LLVM-IR form. The original loops are replaced with calls to the OpenCL kernels.

2 Related Work

The closest work to ours is done by Leung et al.[1]. They present an extension to JikesRVM that enables it to execute suitable code pieces on GPU. The parallelization is done at the level of Java bytecode. The bytecode of suitable loops is transformed to a special IR used by RapidMind, a framework for easing GPGPU programming. Then, RapidMind generates executable GPU code. In the empirical evaluations, ray casting benchmark gives speedups of up to 13x. In general, their customized JikesRVM detects the loops that can be compatible with GPU parallelization and transforms these loops to GPU code. Our implementation is conceptually similar but we modify LLVM IR instead of Java bytecode and we use the LLVM PTX backend instead of RapidMind. Also, our transformation is part of a normal compilation pass, not a JIT one. Finally, for the purpose of this class project, we have not implemented more advanced features like automatic appropriate-loop selection.

Lee et al. [2] present a compiler framework for automatic source to source translation of OpenMP programs to CUDA-based programs. They implemented the translation steps for the Cetus compiler infrastructure that allows source-to-source transformations for C programs. Our implementation differs from this study because it provides the automatic translation based on intermediate representation code.

3 Overview of Loop2GPU

OpenCL is a framework that allows developers to write GPGPU programs. We call a kernel program the code that executes on the GPU. In our case, kernels can be thought of as the body of loops. The host program is the code that executes on the CPU. The host calls the kernel program at runtime.

Loop2GPU processes the non-optimized IR code of a C/C++ program. As a result of the pass, it generates a modified IR code of the host C/C++ program and a new Parallel Thread Execution (PTX) file having the OpenCL kernel. Loop2GPU extends `LoopPass`, so it executes for every loop in the program. We simply generate a new IR file from the body of the loop and replace the loop with the kernel calling statements.

There are two high-level steps in the Loop2GPU. The first one is generating the kernel PTX file from the loop body and the second one is modifying the host C/C++ program.

In the first step, Loop2GPU generates the OpenCL kernel mainly from the body of the loop. The pass merges the IR code generated from the loop with a skeleton of an OpenCL kernel. Then, the pass calls a system command that compiles the resulting IR code to PTX format.

In the second step, Loop2GPU replaces the loop with a set of statements that wire the kernel in the host program. The added statements manage the initialization and communication between GPU and CPU, the execution of the kernel, and the deallocation of OpenCL resources.

```
int main(int argc, char** argv)
{
    int size= 1024000;
    float data[size];      // original data set given to device
    float results[size];

    for(int i=0; i< size; i++)
    {
      results[i]= data[i]*data[i];
    }
    return 0;
}
```

Figure 1: C Program

4 Implementation Details

Loop2GPU extends the functionality of a loop pass. The pass only requires natural loop information and requires that loop preheaders be inserted into the CFG. Also, `loop-simplify` pass is very useful to increase the coverage of our pass.

4.1 Generating the OpenCL Kernel as a PTX file

The OpenCL framework provides two ways of loading a kernel. The commonly used way is to load the kernel in source code format at runtime. The framework then compiles and loads the kernel. Still, this solution does not work for us because it would engender generating source level code from IR code. We assume we don't have access to the source level of the original C++ program and generating C99-style OpenCL code form IR would be awkward.

The second way to load a kernel is to pass it as a PTX file. Although this alternative is more efficient, it is less used because it requires the programmer to have access to a compiler capable of generating PTX code. This capability has started to be added to compilers but, at it is still in its infancy, it is not very reliable. This also proved to be one of the most time-consuming parts of the project. It was hard because of several reasons. First, as previously mentioned, the LLVM PTX backend is not mature. Second, there are differences between platforms regarding the PTX format accepted by the host code. At first, we tried to run the experiments on OS X. We have spent a great deal of time exploring different possibilities of generating the PTX file. Although everything seemed correct, the kernel was still not readable by the host. We eventually contacted Justin Holewinski, the lead developer for the of the open-source PTX back-end of LLVM. After consulting with him and running some tests, the conclusion was that the NVIDIA OpenCL stack on OS X uses a different, non-standard, PTX file format. As we had already spent enough time on this track, we switched to using Ubuntu. We went through similar setup steps and, overall, everything worked much more smoothly.

```
 1 __kernel void kernel(
 2    __global float* input,
 3    __global float* output)
 4 {
 5    // global_id will be used an an iteration value like in the loops.
 6    int i = get_global_id(0);
 7
 8    // here, body of the loop starts
 9    output[i] = input[i] * input[i];
10 }
```

Figure 2: Kernel file

4.2 Transforming the loop body to an OpenCL kernel

Loop2GPU takes the loop body and transforms it to an OpenCL kernel method in IR. Each invocation of the method executes one iteration of the loop. Figure 2 shows the structure of the file containing the kernel in source code format (for readability). The body of the loop is inserted after line 8. The loop index is replaced with the value of `get_global_id(0)`, which generates an id for the GPU kernel. Because only the lines after 8 will change, we use a template file to generate the initialization IR code.

First, our pass reads the generated IR code of this template file. Second, it clones all instructions in the body of the loop. It copies the instructions not only from the main `for.body` block, but also recursively from the successor blocks until it encounters the `for.inc` branch. The cloning is not straightforward because the cloned branch instructions need to be rewired. Third, Loop2GPU fixes the references of the iteration value and the variables that are used in the body of the loop. For this purpose, Loop2GPU maps the old values to the kernel-initialized new ones by using a `ValueToValueMapTy`. Then, the LLVM `RemapInstruction` method is used to automatically replace all the old references with the correct new ones. For now, the implementation only allows two outside variables to be used in the kernel. Still, the implementation is flexible enough to allow easy adaptation to any number of data variables.

4.3 Modifying the IR code of the original C/C++ program

In this step, Loop2GPU transforms the original C/C++ code such that the sequential loop is replaced by the kernel computation. Because the size of host code for the GPU kernel computation is very large, we have again chosen a "template“ approach. We have generated an IR file containing a skeleton of the code needed to wire the kernel into the host program. Loop2GPU clones part of the skeleton into the original program and adapts the instructions where necessary.

First, Loop2GPU clones the loading instructions for dependent libraries and global variables. Then, it transfers the aliases and global variable initializers. Loop2GPU remembers a map from the existing instructions to the cloned ones.

The map is needed at a later stage for updating the references in the instructions.

To reduce the chances of name clashes and simplify the transformation, we have extracted the OpenCL related instructions into a separate method. Loop2GPU clones the GPU method and adapts it.

Loop2GPU identifies the variables that are used in the loop. For this, it traverses all the instructions in all the blocks of the loop, except for `for.inc` and `for.cond`. Loop2GPU checks whether these instructions use a value that is both initialized outside the loop and is not the index value. These values constitute the parameters for both `gpu` method in the host program and `kernel` method in the kernel program. As a last step, before removing the loop, Loop2GPU creates a call instruction that invokes the `gpu` method with the appropriate parameters.

The final step is the removal of the original `for` loop. Even if it sounds simple, this step proved to be error-prone and time consuming. First, we had to carefully interleave the cloning and erasing steps in order to avoid errors. Second, we had to erase the circular dependencies between the `for.end`, `for.inc`, `for.cond`, and `for.body` blocks. The method of removing the IR elements, `eraseFromParent` does not work in the presence of circular references. Hence, Loop2GPU first drops the references of each block by by using the `dropAllReferences` method. Then, it safely removes the blocks. Finally, it removes the loop from the pass manager.

5 Testing

So far, we have only tested Loop2GPU on a set of very simple C/C++ programs. Currently, Loop2GPU is able to transform loops which process only two variables that are defined outside the loop.

References

[1] A. Leung, O. Lhoták, and G. Lashari, “Automatic parallelization for graphics processing units,” in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ ’09. New York, NY, USA: ACM, 2009, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1596655.1596670

[2] S. Lee, S.-J. Min, and R. Eigenmann, “Openmp to gpgpu: a compiler framework for automatic translation and optimization,” in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP ’09. New York, NY, USA: ACM, 2009, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504194