Memory Acces Management with LLVM in OpenCL

Mohammed Yassine Jazouani
jazouani@gmail.com
CEA, Grenoble

Supervised by:
Ylies Falcone, Brice Videau
and Jean-François Mehaut

Abstract

Nowadays, more and more computing devices tend to opt for multi computational components that would work in parallel and include multicore CPUs and GPUs. Tools are needed so that it would be easier and safer to use such technology.

This work addresses the most important problem faced in parallel programming: Memory Management. Indeed, each memory access inside the buffers needs to be checked.

This paper presents how memory management can be done. Using passes written thanks to LLVM applied to OpenCL code will first allow us to pass more information at compile time, concerning the device components and kernel buffers, then will allow us to apply tests concerning the memory accesses and see if these are made correctly.

For example, we can see if there is is any aliasing with the kernel buffers or to see if there are out-of-bound memory access.

To do this work, we have used:

- OpenCL 1.2
- LLVM 3.2
- SPIR 1.2

1 Introduction

Parallel programming introduces shared memory management problem. Indeed, if two devices share one common memory, it is possible that these devices access to a forbidden memory slot or can access to the same memory slot without knowing it and can generate what we call memory data races. Having a tool capable to give enough information concerning the memory to the developer is mandatory to avoid any of these kinds of problems.

LLVM allows to create passes that will get all information concerning a code. There already exist tools that give the possibility to look for invalid memory accesses, memory data races or to improve the OpenCL code to avoid such problems:

- AddressSanitizer's [1] purpose is to detect:
    1. Out-of-bounds accesses to heap, stack and globals
    2. Double-free, invalid free
    3. Memory leaks
- MemorySanitizer [2] is a detector of uninitialized reads.
- ThreadSanitizer [3] is a tool that detects data races.
- OCLoptimizer [4] is used to optimize OpenCL code thanks to LLVM passes during execution time.
- Symbolic OpenCl code testing [5] is used to detect data races. Basically, it runs a program symbolically, which helps keeping track of memory accesses.

However, these technologies are only applied on the code during runtime, and not in compile time. Indeed:

- During compile time, only syntax, typechecking and compiler errors can be revealed. After this step, we only know that the program is well written and that it is possible to run the program compiled.
- During run time, only division by zero, deferencing a null pointer or running out of memory errors are detected. After this step, the program is supposed to finish without crashing.

Doing all the checking during compile time will improve the performance of a code, and more specifically, allow OpenCL to have its own memory management. Indeed, the main idea of this work is to start from an OpenCl program, find a way to use LLVM's tools and passes to get as many memory information as possible during compile time and provide them to the developer.

This paper is organized as follows:

- We will try to have an overview of LLVM and OpenCL which are the two technologies on top of which this work is built.
- Then, the work done will be described as precisely as possible.

- Finally conclude and see if any future work can be done to improve what has been done up til now.

2 Low Level Virtual Machine: Compiler Infrastructure

To understand why we have chosen to use LLVM's technology, it is important to understand what is LLVM, what is it used for and how we can use LLVM to solve our issue [6].

2.1 Definition And Purpose

Low Level Virtual Machine (LLVM) is a framework with many tools aimed to help the developer to compile, optimize and test code. LLVM is implemented in C++, is OpenSource and language independent and its primary purposes are to provide a low level code representation in Single Static Assignment (SSA). [7]

Globally, these are three steps to optimize, analyse and test a code using the LLVM:

1. Code a program in the favourite programming language
2. Use the compiler clang followed by the option '-c -emit-llvm' to generate bytecode
3. Optimizations and analysis are applied on the bytecode using the command "opt"
4. Once all optimizations and analysis are done, generate machine code, and then an executable using the command "llc" [8].

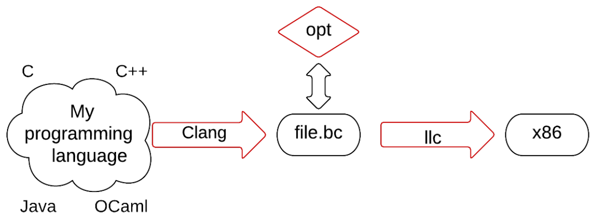

Figure 1: Global view of LLVM use

To understand how LLVM is the solution to our problem, it is important to first introduce the notion of "LLVM pass".

2.2 LLVM Pass

LLVM is the collection of libraries that are already implemented, ready-to-use by the developer that allow code analyses and optimizations. Each analyse or optimization is call a pass.

Concrete examples of pass already implemented by LLVM are:

- Memory allocation:This optimization consists in mapping memory slots into variables. This is helpful because it prevents the compiler to do constant memory access.
- Constant propagation:This optimization consists in substituing the values of known constants in expressions directly during compilation.
- Common subexpression elimination:This optimization consists in looking for common expressions which produce the same result and then compute this common expression only once in the code.
- Many other passes can be found in the website of LLVM [9]

A pass can be run on a complet code or only on a portion. This leads to the distinction of various types of pass. In our case, we will focuse on ModulePass (ran on the whole code) and FunctionPass (ran in functions) [10].

3 Open Computing Language: Open Standard For Parallel Programming

To be able to write testing structures concerning OpenCL, a small explanation concerning OpenCL, the memory sharing representation and programming is advised [11].

3.1 Purpose And Definition

Open Computing Language (OpenCL) is a framework dedicated to writting C programs that will be executed across multiple heterogeneous platforms (mainly composed of CPUs and GPUs). These platforms are represented as a number of **compute devices**. Each compute device will execute program, that is called **kernels**.

- One kernel can be executed in different compute devices
- One compute device is composed of many processing elements

OpenCL programs are compiled at runtime. As said previously, the kernels are written in C, but to be more precise, it is a C version adapted to the device model.

3.2 Structure Of OpenCL

There are four different types of memory systems supported by devices:

1. Global memory which corresponds to each device RAM
2. Constant memory which is a cached global memory
3. Shared local memory which can be accessed by processing elements of the same compute device.
4. Private memory accessible within one and only one processing element.

In an OpenCL program, many initialisations must be done before actually running a kernel on compute devices.

- We must create a context
- Create a command queue in which we will store the kernels that need to be applied for each compute device.
- Allocate the buffer memory objects that will act as memory for compute device.
- Build the compute program
- Build the compute program executable
- Create kernel arguments (such as size, type, flags ...)
- Enqueue command in the command queue

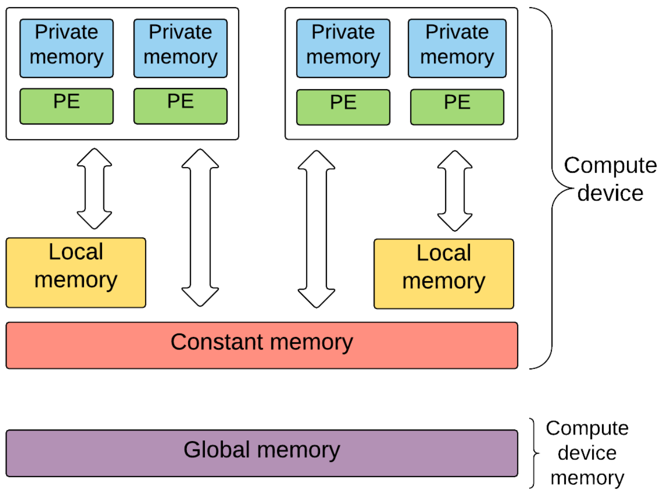

Figure 2: Example of out-of-bound memory access

In our case, we must be able to change the way that OpenCL builds the program. We must be able to find a way to take the OpenCL code, generate a bytecode file that will be modified directly by LLVM. Then, the resulting bytecode is going to be given to OpenCL and execute it.

For that purpose, when the building an executable comes, the bytecode generated by LLVM after passes will be used to build the final program.

3.3 From Runtime to Compile Time

For our work, we will need information concerning the kernels (such as size, access) to be able to perform test during memory access inside kernel functions. This information are only know during runtime. There for, we will need to find a way to communicate this information.

As the runtime of OpenCL is based on such a model, kernels sizes and memory information are known by it. Therefore, it is quite easy to make the passing of information automatic to concerned kernels. So the basic idea behing this is that the passing information make the link between runtime and compile time.

4 Memory Management Done By LLVM On OpenCL

As a reminder, the goal was to create a tool that would give us as many information as possible concerning the buffers usage in an OpenCL program. Mainly, here are the steps to get this information:

1. First, we compile the OpenCL thanks to CLANG and LLVM to get a bytecode. In this step, we needed an intermediate tool that would help this generation. Here inters SPIR. SPIR (or Standard Portable Intermediate Representation) is the first intermediate language for representing parallel compute and graphics. Thanks to this tool, the generation of the bytecode from the OpenCL using LLVM is possible.
2. The generated bytecode is used in LLVM where a pass is applied to gather all type of memory management information (read, write).
3. Then, thanks to additional information, the pass will apply test concerning all memory access
4. Print this information to the developer in a structured way.

To make the collection of information about memory usage possible,further material concerning the buffer used is needed. The first one is their size. Indeed, with the size given, it is possible to check for aliasing or out-of-bound memory access. To get this information, two solutions are offered to us:

1. Force the developer to give as arguments the sizes of used buffers in each function using them.
2. Create a LLVM pass that would change the signature of each function using the buffers dynamically by adding the sizes as arguments. Then, the bytecode generated after this pass is going to be passed to OpenCL so that the sizes needed are incorporated to the code. And all this would be made in compile time.

The second option makes it all easier for the developer. First of all, he doesn't need to know that buffer sizes must be given. Second, if in the future, complementary information are needed for some reasons, no need to change all the structure of the functions. A LLVM pass is run to add other arguments for the needed information. The fact that the developer doesn't have to do more work highlights again the power of LLVM. A very important remark is that all this arguments adding and information passing is done in compile time. Let us see in deep how each step in implemented.

4.1 Adding Arguments In The Bytecode Using LLVM

In this step, the bytecode is generated from the OpenCL file using SPIR and LLVM. The goal is to modify this bytecode such that we have as many arguments as needed for additional information. In this case, let us consider the sizes of buffer used. So the first step is going to parse the code, and at each function declaration, we have to look for how many buffers are present as arguments. Then, we will add as many arguments. For that purpos,LLVM adds arguments of type Integer which name are "size_" followed by the name of the buffer.
It is important to notice that in LLVM, as we are looping inside the code, it is impossible to modify directly the function that we are manipulating. Therefore, the technique used is simple: We create a clone function that will have the same name, the same body. The difference between the old function and the clone is the number of arguments.

Remark. *As we loop on the whole code, our pass is a ModulePass.*

The pass is structured as following:

1. First, we collect all function declaration inside the bytecode.
2. Then, we loop on the arguments of each function and look for usage of buffers. To do so, LLVM can give us the type of values. So we first look for declaration of pointers, and then we look for the ones which address space is equal to 1. Let us assume that the number of arguments that must be added is N.
3. Now, we get the type of the old function, and to this one we will add N arguments of type Integer, representing the sizes we need to add.
4. Then, we have to create the new clone function type to which we need to attach the body of the old function. The name of the clone function is the same as the old one.
5. Now we need to take the arguments names of the old function and map them to the arguments of the new function. Don't forget that we also need to rename the freshly N new added arguments. As a convention, we decided that, if we use a buffer name "x", then the added argument's name would be "size_x".
6. The two last steps are to, first clone the old function into the new one. This will attach the new function to the current module. And then, we will have to delete the old function from the current module.

4.2 Inserting Complementary Information

Now that the arguments are added inside the function's signture, there is another information that needs to be added inside the bytecode: Metadata.

Indeed, when OpenCL receives the freshly changed function, it will look inside the metadata to look for additional information. Basically, there are 5 types of information:

1. What is the address space of each argument of each function: 0 if private, 1 if global.
2. What is the qualification of each argument: If any argument is declared as a constant or not.
3. What is the type of each argument of each function.
4. What is the type qualification of each argument of each function.
5. What is the base type of each argument of each function.

So actually, after adding the size argument, more metadata must be added. To do so, we will just create a new namedMDNode in which we insert the previous medata and just add metada concerning the added argument (sizes of buffers). Technically, if we added N arguments representing the size, we add N metada in each type of information.

In our case, if we added N arguments representing the sizes, we will add:

- Their address space the number 0
- Their access qualification is none
- Their type is int
- Their type qualification is none
- Their base type is int

4.3 Testing All Memory Access

Once all requiered information (argument and metadata) is inserted, it is time to create tests. It has been decided that the most easy and mainstream test is aliasing. Indeed, knowing if two kernel buffers alias each other is easy just by having the address of the buffers and their sizes. Then, it is possible to know if address access are out of bound.

The way to perform test during compile-time is going to be done thanks to IRBuilder. Actually, we are adding instructions containing the tests we want to perform inside the bytecode. To do so, we use a IRBuilder that allows us creating if/then/else instructions, also known as IcmpInst. IRBuilder is a LLVM class that allows the user to create nearly anything possible.

Aliasing

Aliasing it the fact that a data location in memory can be accessed through different symbolic names in the program. Let us have an example:

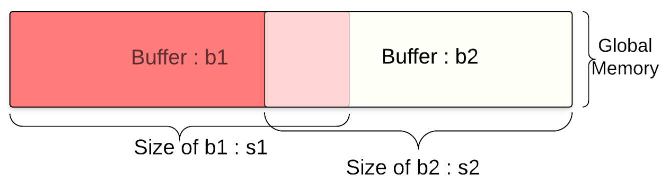

Figure 3: Example of memory aliasing

In this case, we can see that the base pointer of buffer b2 is located inside the buffer b1. The major problem in this case is the following: if any one reads in the aliasing location (represented in pink in 4) and that someone else writes in it, this leads to major memory error. The one who reads before a write doesn't have the update version.

The algorithm used so detect aliasing used in this work is the following:

```
Result: Return true if alias; else return false
List listOfKernels; List listOfSizes;
Kernel currentK = listOfKernels[0];
Size currentS = listOfSizes[0]; i = 1;
while $i \neq listOfKernels.size$ do
    if $currentK <$ $listOfKernels[i]$ then
        if $(currentK + currentS)$
        $<$ $listOfKernels[i]$ then
            No Alias; return false;
        else
            Alias; return true;
        end
    else
        if $listOfKernels[i] <$ $currentK$ then
            if $(listOfKernels[i] + listOfSizes[i])$
            $<$ $currentK$ then
                No Alias; return false;
            else
                Alias; return true;
            end
        else
    end
end
```

Algorithm 1: Detecting aliases

So the main idea is to do computation on kernel buffer's addresses and sizes.

Out Of Bound

Out-of-bound memory access happens when a load or store instruction is executed and that the program tries to access to a memory space that either doesn't exist or to a memory space that is forbidden. To explain the algorithm used, let us base ourselves on the figure

Knowing if there is any out-of-bound memory access needs to be done the following way:

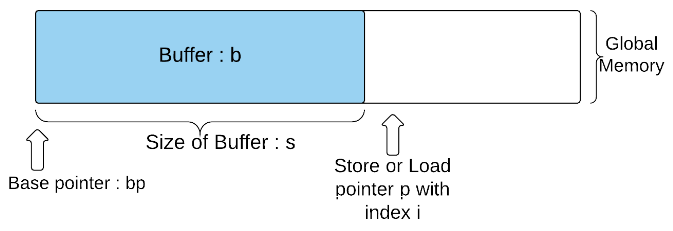

Figure 4: Example of out-of-bound memory access

– Get the list of kernel buffers and sizes of a function
– Get the list of load instructions and store instructions of a function
– For each of these instructions, we retrieve the pointer p and the index i then extract the base pointer bp from it. Then, we look in our list of defined kernels this base pointer.
  1. If the base pointer computed doesn't correspond to any kernel address pointer of the kernel list, then the memory access is totally false.
  2. If the sum of the pointer base and the index is greater than the kernel's address and its size, than we are performing an out-of-bound memory access (as in our figure 4
  3. Else, the memory access can be done.

Remark. *After adding argument and test instructions, the bytecode has been modified. Therefore, it is mandatory to return true at the end of the pass to notify to other passes the modification.*

5 Conclusion

In this paper we presented a way to do memory management on OpenCL code directly at compile time thanks to SPIR. The program has been tested on simple OpenCL code. All transformations are done in such a way that the used isn't even aware of them. He has the illusion that he's using a simple OpenCL program. During the work, assumptions had to be made and we took care of basic coding.

5.1 Future Work

There are many improvements that can be made:

– During the work, we made the following assumption: inside the kernel function that needs to be modified (adding argument and tests), if there is any call to other functions using kernel buffers, we will need to change this call instruction by adding again the argument directly inside the call.
– Another test that can be added is flag testing. Indeed, the kernel buffers have different access rights. They can be in write only, read only or read/write. This information can also be found during runtime.

As for the size, we can test all access (load or store instruction) to the kernel buffer, and see if they don't violate the access right to each kernel.

– We can also add a test concerning concurrent access of a part of a local memory. At this point, the notion of DMA (direct memory access) race is introduced.

5.2 Application

This tool has been explicitly created to overcome memory management problem. Indeed, we get enough information to be aware of aliasing or out-of-bound memory access. The next big step would be detecting DMA races. Thanks to the possibility to create testing structures directly inside the bytecode, this can be done and help OpenCL programmers write more secure and more powerful code.

6 Acknowledgments

First, I would like to thank my supervisor Ylies Falcone for proposing me this work and guiding me the whole time. A great thank you to Jean-François Mehaut who gave me good advices during the work. I would like to show my gratitude to Brice Videau, who helped me a lot in the realization of this project and gave me precious remarks. Without his guidance, this work wouldn't be what it is.

I would also like to sincerely thank the INRIA Corse team for sharing their huge knowledge of LLVM programming. A special thank you to Fabian Grubber for his insights and very useful help during coding passes.

References

[1] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[2] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, San Francisco, CA, USA, 2015.

[3] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[4] Jorge F Fabeiro, Diego Andrade, and Basilio B Fraguela. Ocloptimizer: An iterative optimization tool for opencl. *Procedia Computer Science*, 18:1322–1331, 2013.

[5] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. Symbolic testing of opencl code. In *Hardware and Software: Verification and Testing*, pages 203–218. Springer, 2012.

[6] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[7] Peter Lee, Frank Pfenning, and André Platzer. Static single assignment.

[8] Chris Lattner and Vikram Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[9] The LLVM Compiler Infrastructure. `http://llvm.org/`. [Online; accessed 19-May-2015].

[10] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] John E Stone, David Gohara, and Guochun Shi. Opencl: Aparallel programming standard for heterogeneous computing systems.