NVIDIA's Experience with Open64

Mike Murphy
NVIDIA
mmurphy@nvidia.com

Abstract

NVIDIA uses Open64 as part of its CUDA toolchain for general purpose computing using GPUs. Open64 was chosen for the strength of its optimizations, and its usage has been a success, though there have been some difficulties. This paper will give an overview of its usage, the modifications that were made, and some thoughts about future usage.

1. Background

NVIDIA has traditionally been a leader in making graphics chips (GPUs). The GPU is specialized for graphics rendering, which involves arithmetic-intensive, highly parallel computations. GPUs are well designed for data parallel (SIMD) computations, where the same program is executed on many data elements in parallel, and where the arithmetic costs dominate the memory costs. Due to the specialization, the GPU's have very high performance for such applications. Several years ago we realized that the parallel processing capabilities of graphics chips could potentially be useful in parallel non-graphics applications as well. The difficulty in unlocking the computation capabilities of a GPU for other applications lies in providing a simple way to program a data parallel machine. NVIDIA's solution is called CUDA. CUDA stands for Compute Unified Device Architecture. CUDA is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API. When programmed through CUDA, the GPU is viewed as a *compute device* capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU, or *host*: In other words, data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device. CUDA provides a few simple extensions to C/C++ that enables users to specify what part of their program they want to run in parallel on a GPU. The CUDA compiler driver splits an application into two parts, one which is run on the host CPU, and the other which is run on the GPU. The CPU code uses the default host compiler, while the GPU code uses a new toolchain that includes Open64.

2. Why Open64?

NVIDIA already had a well-tuned low-level compiler for graphics codes, called OCG (Optimized Code Generator). This handles register allocation, scheduling, and peephole optimizations. This compiler is built into the graphics driver for doing just-in-time compilation of graphic codes. Because graphics codes tend to be relatively simple compared to general purpose C code (control flow has only recently been added to DirectX), and due to the compile-time limitations of needing to compile at runtime, it was decided that CUDA needed a high-level optimizer to handle the compute codes. The optimizer would do all the traditional global optimizations and pipe the output to OCG. OCG would then generate efficient code for the graphics processor. There were three options for doing this: write a new proprietary optimizer, use GCC, or use Open64. Writing a new optimizer was ruled out because it would require too many man years to develop. So the choice was between using GCC or Open64. Because of Open64's reputation for optimization (and with my encouragement), we chose Open64.

3. Scope of Work

The CUDA compiler driver (nvcc) uses a preprocessor (cudafe) that splits the application into CPU and GPU parts, creating a preprocessed C file for the GPU input. The GPU input is then processed by our Open64 compiler (called nvopencc), which emits an assembly language that we created called PTX (Parallel Thread Execution). PTX provides a virtual machine model that multiple tools can target, and which is independent of the underlying processor. Some important features of PTX are that it has:

- unlimited virtual registers of different sizes,
- explicit memory segments for different levels of sharing between threads,
- no stack or heap,
- strongly typed instructions,
- vector support for memory accesses,
- predication for branching,
- C-like syntax for calls.

The PTX is then passed to OCG, which allocates registers and schedules the instructions according to the particular chip that is being used.

CUDA .cu program

⇩

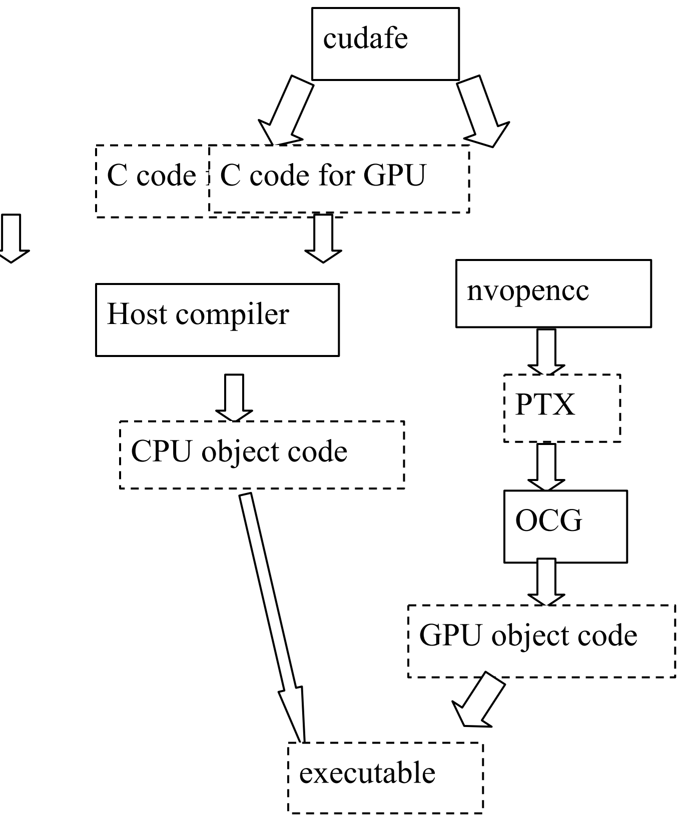

Figure 1. nvcc toolchain

We use a subset of Open64. One simplification is that the input is always C. CUDA code can be written in C++, but that is then translated into C. We perform inlining, but do not currently do any cross-file inter-procedural analysis (IPA). There is no cache to model, and software pipelining does not make sense, so we do not currently use LNO (Loop Nest Optimizer). Finally, because we utilize NVIDIA's own code generator for register allocation and scheduling, we are able to use a very simple code generator in Open64, which basically does CGEXP (Open64's instruction selection) and then a few target-specific optimizations before CGEMIT (code emission). So the nvopencc toolchain consists of the gcc-based front end, the inliner, and the backend. The backend is basically VHO (Very High Optimizer), WOPT (Whirl OPTimizer), and a small CG (Code Generator), with targ_info describing the PTX target.

4. Issues We Faced

The first job obviously was to port the compiler to our PTX target. Most of this was straightforward changes to targ_info and the target-specific files. There were some issues with the unlimited number of registers and the lack of a stack, and the need to know which memory segment is being referenced.

A bigger problem was that CUDA is supported on many platforms, in particular, 32 and 64bit Windows, 32 and 64bit Linux (tested on multiple distributions), and Mac OS. We started our port with the PathScale sources, which worked on Linux, but we needed to get it to work on Windows as well. Tensilica and STMicroelectronics had independently worked on Windows ports, but the main source repository did not have their changes, and this is an example of an area where better sharing of source changes would make it easier to port Open64 to new areas. We were able to get hold of the changes that Tensilica made for running on Windows, which was very helpful. We decided to utilize a cygwin variant called mingw, which allows us to build using cygwin (which provides Linux-like tools for development on windows) but run on systems that do not have cygwin installed. So the developers use gcc to build, but anyone can run the resulting executables. We also decided to avoid the problems of DLLs and shared libraries by building all of our backend into a single executable. Because we do not have "optional" pieces (we always compile optimized, for reasons that will be explained later), there is no advantage to using shared libraries.

To get good performance on a GPU, one needs to try and keep data in registers, avoiding memory accesses. There is no stack or fast memory to use. Therefore we want to optimize everything into pregs or virtual registers as much as possible. We also do not have a flat memory space, so we need to know what memory space is being referenced by pointers. Therefore we decided to inline everything by default. This enables us to optimize more code into registers and track memory references, at the cost of larger code size. It also means that we do not currently support recursion, which would not be efficient with the current architecture anyways.

Our applications typically do a lot of work with small structs, which by default were often allocated to memory and copied from memory. That works fine when you have a stack and fast memcpy, but is slow on a GPU. VHO already had some ability to expand structure copies into copies of the individual fields, so we enhanced that to handle more cases, which enables us to often keep the struct in registers (if address is taken then we fall back to the slower memory accesses).

Counteracting the above desire to split structs into multiple registers is the desire to use vector memory loads and stores to access multiple chunks of memory at once. To take advantage of this ability, we added a pass in the Open64 code generator to merge adjacent memory accesses into vector memory accesses. This pass tries to merge loads and stores into vectors as long as they do not have any intervening redefinitions, and will even re-align objects to get the best coalescing (since vectors require a larger natural alignment). This vectorizing ability helps both structs and array accesses.

Another enhancement to the code generator was to add a pass that transforms 32bit instructions to 16bit instructions when there will be no loss of precision. For example, if a user multiplies two shorts:

```
    short a, b, c;
    c = a * b;
```

The C rules promote this to a multiplication of "int" size, meaning a 32bit multiply. But some of our chips can do 16bit multiply faster than a 32bit multiply, and in this case since the result will be 16bits, it is safe to just do a 16bit multiply. We considered doing this as part of the bitwise-dead-code-elimination pass of WOPT, but WHIRL does not have I2/U2 variants of operations, and pregs seemed to confuse bdce, plus this was specific to our target. So we decided to do this as a separate pass in CG. This was done by analyzing 16bit load/store/converts and then propagating the info forwards and backwards through the TNs (Temporary Names, which are instruction operands in CG); then we unmark any TNs that are used in places that cannot be 16bit, and then change the remaining TNs and associated instructions to be 16bit.

Register pressure is a big issue for our target, and one way of limiting register pressure is to make sure we use the smallest-size register that is needed. This showed up particularly for 64bit compiles, where sometimes the default code would do 32bit operations in 64bit registers. For us, it is better to keep the data in 32bit registers, and to do that we had to be more careful about inserting and preserving 32<->64bit CVTs in the WHIRL.

To reduce register pressure we also added some cross-basic-block rematerialization to CG, to make it easier for the time-constrained register allocator in OCG.

5. Results

Open64 is a key component of NVIDIA's CUDA toolchain, and we are pleased with the quality and robustness of the code. We are in the process of releasing version 2.0 of the software, and are continuing to add features, fix bugs, and improve performance. We have seen impressive performance speedups on some parallel apps with CUDA, and Open64 has been an integral part of that success (e.g. Prof Wen-mei Hwu's group at University of Illinois demonstrated a 400x speedup using CUDA on a MRI application).

It is difficult to truly quantify the effect of Open64 on the performance of CUDA. One simple datapoint is to compare the performance of the default compile (which is optimized) with a run of Open64 at –O0. For a sample binomialoptions financial application that we have, comparing to a baseline of 100 options per second when run on a dual-core cpu, if we use Open64 at –O0 we only get 315 options per second, but with full optimization we get 4231 options per second, for a speedup of 13x over non-optimized code. On another financial application called blackscholes, we get a speedup of 32x over non-optimized code. However, before we get too excited about these numbers, what this really illustrates is just the importance of optimization for this architecture, particularly optimizations to avoid memory accesses. Further analysis reveals that 90% of the above improvement is just due to keeping things in registers. When every thread has to do a slow memory access, our performance degrades quite a bit, so there is a large benefit from simply putting local variables in registers.

6. Future Work

The above work was primarily done by myself over a period of two years, and enabled the successful release of an initial CUDA toolkit. Recently we have added several people to the group who are learning Open64 and will be contributing to our future efforts. Open64 is already being used for one interesting new project which will be announced at a later date.

We also hope to merge our target-independent changes back into the public Open64 sources. There is a lot of interesting work being done by different companies with Open64, but unfortunately the code changes are not always being merged into a common code base. It would benefit us all if there was closer cooperation and sharing of ideas. It would also help if there was better documentation of Open64, its optimizations, and how to modify the code.

For the current CUDA toolchain, one of the biggest remaining issues is register pressure. GPUs have a tradeoff between parallelism and registers, as they do not have a fixed register limit but instead divide the registers between threads. For example (this is not our real architecture but is simplified for understanding), if a chip has 100 registers it can either use 10 threads that take 10 registers each, or 5 threads that take 20 registers each. So the more registers that are needed, the less parallelism we can get. But if we have to spill a lot of registers, then that also hurts performance due to the slow memory accesses. So we sometimes hit a "performance cliff" due to register pressure.

The register pressure problem is exacerbated by our split of the register allocator from the open64 optimizer. Some of WOPT's optimizations create more register pressure (e.g. hoisting can lengthen a live-range). We have tried to tune some of WOPT's optimizations (e.g. we do not do aggressive code motion) to help with register pressure, but that has had mixed results. It would be helpful if there was a way for WOPT to take register pressure into account while doing its optimizations.

We are also looking at the possible benefits of using other parts of the Open64 toolchain like LNO, IPA, and feedback, as well as enhancing the compiler with new user and architectural features. In summary, NVIDIA is continuing to invest in the Open64 technology, and is hoping that the open-source code will continue to improve.

Acknowledgments

Jayant Kolhe and Ian Buck were instrumental in initiating and managing the CUDA compiler work. Bastiaan Aarts helped with the porting work, and this paper was improved by comments from Vinod Grover, Inderaj Bains, and Douglas Saylor. More information about CUDA, as well as information about ftp access to a copy of our Open64 sources, is available at www.nvidia.com/**cuda.**