

WASTE, *Why Another Simple Trivial Emulator for CUDA?*

Kenneth Domino

November 7, 2010

Introduction

CUDA C++ is a parallel programming language for NVIDIA GPU's. Using the CUDA Toolkit (http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html), developers can write programs that take advantage of the large parallel computing potential of the GPU, speeding up their programs several orders of magnitude. CUDA programs are executed on the GPU, which works as a coprocessor with the CPU, having its own memory and instruction set. But, the question is, after the developer invested the time to parallelize his program, can the CUDA program run on a PC without an NVIDIA GPU? Does the developer have to redo all his software?

In short, the answer to this question is currently "yes", because the program contains instructions that must be run on the NVIDIA GPU. However a solution will be available in the near future. Portland Group International is developing a compiler that translates the program into Intel x86 machine code, which can be run on the CPU without the NVIDIA GPU. But, this compiler won't be available for several months, and it will not be free. MCUDA is a similar technology, but it is a research project, not a commercial product[3].

If the developer is looking to simply test his code without regard to speed, then the developer has another alternative. The CUDA program can be executed using an emulator for the NVIDIA GPU, a special software program that mimics the actions of the GPU on the CPU. The CUDA program executes unaltered. The problem with this solution is that is very slow. While some parallelism can be achieved by taking advantage of the multiple cores in the CPU, the program runs much slower in comparison to running it on the GPU.

NVIDIA used to provide an emulator in its CUDA Software Development Kit, but that is no longer the case. Unfortunately, NVIDIA eliminated the emulator from version 3.0 of the CUDA Toolkit. Why? One reason is that PGI is going to be providing their CUDA C++ x86 compiler. Another reason is that the emulator required too much effort to support. NVIDIA makes no profit from the emulator, because it is free. NVIDIA makes its profits through sales of hardware.

Two other CUDA emulators exist: Ocelot[2] and Barra[1]. However, neither emulator runs on Windows, and it is unknown when either will be ported to Windows, if ever. While these could be ported to Windows, it is often easier to write code from scratch, rather than try to modify an existing program.

WASTE, an emulator for CUDA programs, is the result of one month of programming. The purpose of the program is to provide a means to develop and test algorithms on a PC that does not have an NVIDIA GPU.

WASTE is built and run on Windows only. This code is not portable to other systems because it makes a number of assumptions: the host machine is little endian; DLL injection is performed using Windows system calls and x86 Intel assembly code; call stack trace back uses x86 Intel assembly code; fundamental data types in CUDA are represented using non-standard fundamental data types in C++. However, these assumptions are fine because WASTE is not intended to be ported to any other environment. Many design decisions were made in order to simplify the coding of the emulator.

WASTE does not run quickly. While the host portion of a program will run quickly, WASTE interprets the PTX code for CUDA kernel portion of the program. Most programs will run, but will be several orders of magnitude slower on the emulator in comparison to running them natively on a NVIDIA GPU. The main reason for the slow runtime is that WASTE interprets each PTX instruction, which involves the extraction of the instruction from the AST representation of the kernel code, and symbol table manipulations, which are used to represent the values of registers and memory. In comparison, Ocelot translates PTX assembly code to LLVM, then into native code on the host machine. Barra is slightly different in that it interprets the machine code that is produced by the PTXAS assembler. It turns out that the PTXAS assembler is more like a compiler, and performs a great number of optimizations which speed up the original PTX code.

WASTE does not do a lot of invalid instruction checking, especially so in “release” mode. If you write hand-crafted PTX programs, they may not work at all because WASTE is targeted to run code produced by the NVCC compiler, which uses a small subset of the PTX instruction set. That said, WASTE tries to emulate more than this subset. Much of the CUDA Runtime API is supported, as well as the CUDA Driver API. The CUDA Runtime API is a layer that makes calls to the CUDA Driver API. Calls to the CUDA Driver API are intercepted using DLL injection and API hooking. If you write your programs using the CUDA driver directly, you will need to check your assembly code by first using the PTXAS tool.

WASTE does, however, have several nice features. First, it works on most code generated by the NVCC compiler. Many of the sample programs in the SDK work. Second, WASTE does not depend on a lot of other software libraries. If you are so inclined, you can build the emulator from the latest sources using Microsoft Visual Studio Essential C++, so it does not require you to buy additional software. Third, it performs DLL injection, which means that you do not need to rebuild your program and link it with WASTE. You can and debug run both native GPU and emulated GPU side by side for comparison.

Downloading WASTE

There are two ways to get WASTE. You can either download one of the pre-built installers, or you can download the latest sources and build the executable yourself.

Install from MSI setup file

Download the MSI installation file from <http://code.google.com/p/cuda-waste/downloads/list>. Execute MSI file. It will then install the program on your PC, in the location you specify. The setup program will install WASTE by default in "c:\Program Files\waste\waste". The program executable is "waste.exe".

Install Source

The source for WASTE is open source, and it is under Subversion control. To build WASTE, you will need Microsoft Visual Studio C++ 2010 Express or Professional.

You will also need to get the sources directly from the Subversion repository. For Subversion, it is recommended that you install Cygwin (<http://www.cygwin.com>). If installing Cygwin, use the "Select packages" dialog box to subversion. Select "subversion: A version control sytem" under the Devel category.

After installing subversion, start a bash prompt, and type:

```
svn checkout http://cuda-waste.googlecode.com/svn/trunk/cuda-waste
```

You will also need several other packages to build WASTE:

The Antlr parser generator: <http://wwwantlr.org/>

The NVIDIA CUDA GPU Toolkit:

http://developer.NVIDIA.com/object/cuda_3_2_toolkit_rc.html

Building

Open the Microsoft Visual Studio project file for WASTE, cuda-waste/waste.sln, then select Build. Some parts require separate builds. The tests in cuda-waste/test/hw must be build with Microsoft Visual Studio 2008. If you are modifying the grammar in cuda-waste/ptxp, execute make in the directory.

If you are running exclusively on a computer that does not have an NVIDIA GPU, then you may not be able to install the drivers for CUDA. (These drivers are required even in emulation mode because the

DLL injection/API hooking method used by WASTE needs the DLL's.) In that case, you must build and install the nvcuda DLL project and install the DLL in a directory in your executable path.

Running WASTE

To run WASTE, type “*waste program.exe*”, where *program.exe* is your CUDA program. WASTE will inject code into your program to perform emulation. There are a number of options you can use to change the behavior of WASTE.

USAGE: WASTE [OPTION] ... USER-PROGRAM(.EXE)

Options:

-e Set emulator mode.

By default, WASTE runs programs in emulator mode, so this option is not very useful. This option is added for the sake of completeness.

-ne Set non-emulator mode.

If you do not want to run WASTE in emulator mode, but still want to check memory usage, then use this option.

-d device Set the name of the device to emulate.

By default, the emulator will choose `compute_20` for the device to run. Use this option to select another device if it was compiled into the program using NVCC and the `-arch` option.

-t=NUMBER Trace CUDA memory API calls, emulator, etc, at a given level of noisiness.

Use this option to have the emulator print out debugging information, and for tracing memory API calls. The higher the number, the more debugging information is printed.

-s=NUMBER, --padding-size=NUMBER Set size of padding for buffer allocations.

When allocating device memory, WASTE will pad the buffers with 32 bytes on either side of the buffer. The padding will be initialized to the value `0xDE`. If there are memory overwrite issues with the program, WASTE may detect these if the padding has changed.

-b=CHAR, --padding-byte=CHAR Set byte of padding.

Use this option to set the padding value. See the “-s” option.

-q, --quit-on-error Quit program on error detection.

By default, WASTE will try to detect and flag certain errors in the CUDA API. This is done because some programmers do not check the return values from the API. This option causes WASTE to quit when an error has been detected.

-k, --skip-on-error Skip over CUDA call when the usage is invalid.

Some CUDA API calls are checked for valid arguments prior to actually calling the API function. This option causes WASTE to skip the call if the arguments to the function are invalid.

-n, --non-standard-ptr Allow computed pointers.

Some CUDA API functions that pass pointers to functions can actually be computed addresses. For example, if the programmer allocates a large buffer using `cudaMalloc`, he can compute a pointer to a point within the buffer and pass that to `cudaMemset`. By default, WASTE does not allow this. This option allows this kind of pointer usage.

-x Start debugger.

Debugging WASTE on a program is often required to understand where and what kind of error occurred in WASTE. Because the WASTE executable is a separate process from the user program, there must be a mechanism to start a debugger on the user program after DLL injection but before API hooking. This option performs this task.

Requirements

WASTE, like all programs, is the result of many different compromises. As previously mentioned one goal was to have this program written as quickly as possible.

For several weeks prior to starting WASTE, there was an effort to port Ocelot to Windows Cygwin. While it was possible to get it to compile, an emulation of a simple “Hello World” program would not work. In addition, there were many dependencies in the build on Linux tools: make, autoconf, external libraries, etc. While it could be ported to build using the Microsoft compiler, there was no script to build Ocelot. Extending it to work on Windows would be very slow.

Prior to starting WASTE, a CUDA memory debugging program was written (<http://code.google.com/p/cuda-memory-debug/>), which would trace proper use of CUDA memory allocation, usage, and deallocation. This program included code for DLL injection and API hooking, two important features required in WASTE.

- 1) WASTE must work on Windows. All other emulators are Linux tools, and do not work on Windows. Even if these programs were modified to compile and link on Windows, changes would be needed to get either emulator to work with Windows executables.
- 2) DLL injection and API hooking must be the primary means of calling WASTE of CUDA program. The user should not be forced to rebuild his program in order to run the program with emulation. A side-by-side comparison of the original and the emulated program must be possible. Barra and Ocelot, two other emulators that exist, do not do DLL injection. Instead, they require the user to relink the program with a library that replaces the CUDA API.
- 3) Linking of WASTE with a user program must be possible. While a user should not have to rebuild his program with WASTE, it should be possible to link WASTE into the program so he does not have to consciously call WASTE every time he wants an emulated version of the program. In addition, linking WASTE to the program allows WASTE to be more easily debugged. Barra and Ocelot are linked emulators.
- 4) WASTE must be build using the Microsoft compiler. NVIDIA provides a compiler suite for CUDA C programs. NVCC, the CUDA C compiler, is a wrapper program that calls several different programs, including the Microsoft Compiler, CL. There have been attempts to substitute GCC for CL in NVCC, but they have failed because the code generated by the CUDA C front-end translator generates C code that is incompatible with GCC.
- 5) WASTE must provide tracing of the CUDA memory API.
- 6) WASTE must be able to run on a machine that does not have an NVIDIA GPU installed. While the NVIDIA CUDA GPU Toolkit can be installed on any machine, running that program even only with WASTE is not possible. When loading the user CUDA program, the NVIDIA GPU driver DLL

nvcuda.dll will be needed, even if WASTE is used. WASTE will provide a stub DLL in order to run the program with the emulator.

- 7) WASTE does not need to intercept all of the CUDA API, and the PTX instruction set.
- 8) WASTE must execute PTX assembly language code. It does not need to execute machine code at this point. The NVCC compiler generates code in two phases. In the first phase, it generates PTX assembly language code (-arch), and machine code (-code). The PTX assembly code is used by the JIT for CUDA, and allows the CUDA program to run on a variety of GPU's. The machine code forces the program to run on a specific GPU, and cannot be run on a different GPU.
- 9) WASTE should use the Antlr parser generator. Antlr is a widely-used parser generator, easy to use, and has a debugger, Antlrworks, to test the grammar in a standalone manner.
- 10) WASTE must provide command-line options of features to select. Since WASTE is a stand-alone program that loads and executes user programs, it must have a way to invoke the debugger just after DLL injection, but before API hooking.

How WASTE works

Kernel launch sequence

There are two CUDA API's a user can call in a CUDA program: the CUDA Driver API and the CUDA Runtime API. While some programmers use the CUDA Driver API, most prefer to use the CUDA Runtime API instead. The CUDA Runtime API, along with the syntax available in the CUDA C++ compiler for writing and calling kernels, provides an easier way to write CUDA programs.

When a user writes a CUDA Runtime API program, the compiler will translate the C-like program into PTX code, and insert calls to the CUDA Runtime API to register and call the kernel. The API calls are not generally noticed by the user, but WASTE needs to intercept these calls to perform the emulation.

We now look at the sequence of CUDA API calls for an example (Figure 1). This simple CUDA program creates an integer variable; initialize the variable to -1; copies the value to device global memory; calls the kernel *fun*, which assigns the absolute value of the variable back to the variable within a single thread; copies the value from device global memory; and prints the integer to the standard output device. This program uses the CUDA Runtime API, specifically *cudaMalloc*, *cudaMemcpy*, *cudaThreadSynchronize*, *cudaGetLastError*, and the chevron-syntax kernel call. The CUDA C++ compiler translates the `__global__` (kernel) code into PTX assembly language, the chevron syntax for the kernel launch into three CUDA Runtime API calls, and sets up the kernel in PTX with the runtime (Figure 2).

Figure 1 — ABS Example

```
#include <iostream>

__global__ void fun(int * mem)
{
    int d = abs(*mem);
    *mem = d;
}

int main()
{
    int h = -1;
    int * d;
    cudaMalloc(&d, sizeof(int));
    cudaMemcpy(d, &h, sizeof(int), cudaMemcpyHostToDevice);
    fun<<<1,1>>>>(d);
    cudaThreadSynchronize();
    int rv = cudaGetLastError();
    cudaMemcpy(&h, d, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "Result = " << h << "\n";
    return 0;
}
```

Figure 2 — CUDA API calls for the ABS Example

```
1)  cudaRegisterFatBinary(fatCubin = 0x00b01000 __fatDeviceText)

2)  cudaRegisterFunction(fatCubinHandle = 0x00000000, hostFun = 0x00a7109b "  ", deviceFun = 0x0141520c "_Z3funPi", deviceName = 0x00ae5200
    "_Z3funPi", thread_limit = -1, tid=0, bid=0, bDim=0, gDim=0, wSize=0)

3)  main()

4)  cudaMalloc(size = 4)

5)  cudaMemcpy(..., ..., count = 4, kind = cudaMemcpyHostToDevice)

6)  cudaConfigureCall(gridDim = {x=1 y=1 z=1 }, blockDim = {x=1 y=1 z=1 }, sharedMem = 0, stream = 0x00000000)

7)  cudaSetupArgument(arg = 0x001cfd60, size = 4, offset = 0)

8)  cudaLaunch(entry = 0x013a109b "  ")

9)  cudaThreadSynchronize()

10) cudaGetLastError()

11) cudaMemcpy(..., ..., count = 4, kind = cudaMemcpyDeviceToHost)

12) cudaUnregisterFatBinary(...)
```

In addition to the calls to *cudaMalloc*, *cudaMemcpy*, *cudaThreadSynchronize*, and *cudaGetLastError*, there are several other calls to the CUDA Runtime API: *cudaRegisterFatBinary*, *cudaRegisterFunction*, *cudaConfigureCall*, *cudaSetupArgument*, and *cudaUnregisterFatBinary*. These additional functions are calls from code generated by the CUDA compiler.

cudaRegisterFatBinary is used to register the PTX code with the CUDA runtime, which is generated by the CUDA C++ compiler. The parameter, *__fatDeviceText*, is a pointer to a structure (Figure 3) that contains information related to the PTX code, the entry points, symbols, and debugging information. The API function extracts this information from the *__fatDeviceText* pointer. In addition, the API parses the PTX code, which is why it some users notice a lag time in the start up of the program. The PTX assembly code for the ABS Example consists of five PTX instructions (Figure 4).

cudaRegisterFunction is used to register the PTX entry point with a particular host function. The *hostFun* parameter is used by the program to note which function it wants to call.

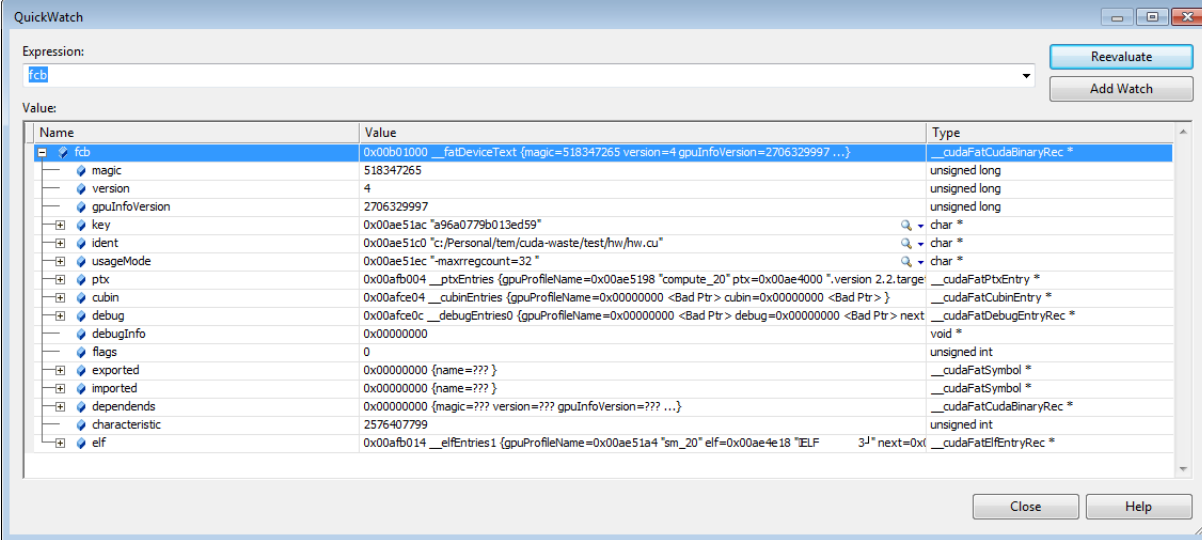
CudaConfigureCall is used to specify the grid and block sizes for the kernel launch. In this example, the grid size is a single block, and the block size is a single thread.

cudaSetupArgument is used to pass parameters to the kernel.

cudaLaunch is used to call the kernel. The parameter for the function is the entry point.

cudaUnregisterFatBinary is used to clean up all the data structures for the PTX code that the CUDA Runtime API uses.

Figure 3—The `__cudaFatCudaBinary` Structure



Name	Value	Type
fcb	0x00b01000 __fatDeviceText {magic=518347265 version=4 gpuInfoVersion=2706329997 ...}	__cudaFatCudaBinaryRec *
magic	518347265	unsigned long
version	4	unsigned long
gpuInfoVersion	2706329997	unsigned long
key	0x00ae51ac "a96a0779b013ed59"	char *
ident	0x00ae51c0 "c:/Personal/Item/cuda-waste/test/hw/hw.cu"	char *
usageMode	0x00ae51ec "maxregcount=32"	char *
ptx	0x00afb004 __ptxEntries {gpuProfileName=0x00ae5198 "compute_20" ptx=0x00ae4000 ".version 2.2.target"	__cudaFatPtxEntry *
cubin	0x00afce04 __cubinEntries {gpuProfileName=0x00000000 <Bad Ptr> cubin=0x00000000 <Bad Ptr> }	__cudaFatCubinEntry *
debug	0x00afce0c __debugEntries0 {gpuProfileName=0x00000000 <Bad Ptr> debug=0x00000000 <Bad Ptr> next	__cudaFatDebugEntryRec *
debugInfo	0x00000000	void *
flags	0	unsigned int
exported	0x00000000 {name=???}	__cudaFatSymbol *
imported	0x00000000 {name=???}	__cudaFatSymbol *
depends	0x00000000 {magic=??? version=??? gpuInfoVersion=??? ...}	__cudaFatCudaBinaryRec *
characteristic	2576407799	unsigned int
elf	0x00afb014 __elfEntries1 {gpuProfileName=0x00ae51a4 "sm_20" elf=0x00ae4e18 "ELF 32" next=0x1	__cudaFatElfEntryRec *

Figure 4—The generated PTX code for the ABS Example

```
.version 2.2
.target sm_20
.entry _Z3funPi (
    .param .u32 __cudaparm__Z3funPi_mem)
{
    .reg .u32 %r<9>;
    ld.param.u32    %r1, [__cudaparm__Z3funPi_mem];
    ldu.global.s32    %r2, [%r1+0];
    abs.s32          %r3, %r2;
    st.global.s32    [%r1+0], %r3;
    exit;
} // _Z3funPi
```

DLL injection

WASTE emulates the kernel code of a CUDA program by intercepting the CUDA Driver and Runtime API. To do this, WASTE performs DLL injection and API hooking in a multi-step boot process. The first step in this process is DLL injection, which is the process of inserting foreign code into another program. The purpose of the WASTE program is to (1) start the user CUDA program in suspended mode; (2) inject the WASTE DLL emulator into the user process, which pass options to the user program; and, (3) start the user program with the WASTE emulator.

In the first step of DLL injection, WASTE starts the user program in suspended mode. During the startup of the CUDA program, calls to the CUDA runtime API are made under the covers without the user's knowledge. In order capture these calls, the program must be created. But, the calls to the CUDA runtime API must be avoided until the point when WASTE is able to capture the calls.

In the second step of DLL injection, WASTE creates a block of code in the suspended CUDA program. We call this code block the DLL Injection Code Block (DICB). Before the user program can execute, the DICB will be executed. WASTE writes Intel x86 machine code instructions to load the WASTE DLL, hook the CUDA runtime API, and set options for the WASTE emulator. A thread is created in the user program to execute the DICB. The WASTE program waits for this thread to finish. DLL injection is now complete.

Finally, WASTE now starts the main thread of the user program. This executes the user program with the WASTE emulator.

The following source code from WASTE loads the WASTE DLL.

```
// Add code to load wrapper library.
{
    AddBytes(code, 0x9c); // pushfd
    AddBytes(code, 0x60); // pushad

    // Inject cuda-memory-debug wrapper library target (load the library into this program).
    AddBytes(code, 0x68, 0x00, 0x00, 0x00, 0x00); // push "wrapper.dll"
    JmpAbsoluteAddress(code, size-4, pszCMD); // patch with actual string address.

    AddBytes(code, 0xE8, 0x00, 0x00, 0x00, 0x00); // call LoadLibraryA
    JmpRelativeAddressBased(code, size-4, &LoadLibraryA, codePtr, 0);
    // patch with actual function address.
}
```

API hooking

The user's CUDA program contains many calls to the CUDA runtime API. These calls must be intercepted by the WASTE emulator in order to simulate the execution of an NVIDIA GPU. During the execution of the DICB, WASTE sets up hooks in the entire user program to intercept these calls, with the call to WrapCuda. This code has an extra call to LoadLibraryA of the wrapper DLL because the module handle must be used to get the address of WrapCuda.

```
{
    LPVOID pszSetFunc = VirtualAllocEx(hProcess, NULL, strlen(str_padding_size) + 1,
    MEM_COMMIT, PAGE_READWRITE);
```

```

    BOOL rv_wpw2 = WriteProcessMemory(hProcess, pszSetFunc,
(LPVOID) str_wrap_cuda, strlen(str_wrap_cuda) + 1, &written);
    AddBytes(code, 0x68, 0x00, 0x00, 0x00, 0x00); // push "wrapper.dll"
    JmpAbsoluteAddress(code, size-4, pszCMD);
    AddBytes(code, 0xE8, 0x00, 0x00, 0x00, 0x00); // call LoadLibraryA
    JmpRelativeAddressBased(code, size-4, &LoadLibraryA, codePtr, 0);
    AddBytes(code, 0x68, 0x00, 0x00, 0x00, 0x00); // push str_wrap_cuda
    JmpAbsoluteAddress(code, size-4, pszSetFunc);
    AddBytes(code, 0x50); // push eax
    AddBytes(code, 0xE8, 0x00, 0x00, 0x00, 0x00); // call GetProcAddress
    JmpRelativeAddressBased(code, size-4, &GetProcAddress, codePtr, 0);
    AddBytes(code, 0xFF, 0xD0); // call eax
}

```

WrapCuda itself looks to wrap the NVIDIA CUDA driver API contained in NVCUDA.DLL and the CUDA runtime API.

API hooking in WASTE is done via *Import Address Table (IAT) patching*. All modules are searched for the each API procedure to patch. When it is found in the IAT, WASTE will substitute the entry with the address of the wrapper function.

The PTX Parser

When a CUDA C program is compiled, the CUDA C compiler will translate the C code into a mixture of x86 machine code and PTX assembly language code. The PTX code, which corresponds to kernel source code, is further translated by the PTXAS assembler into the GPU machine code.

When the CUDA program executes, the *cudaRegisterFatBinary* API is called during start up, even before *main* is called. In *cudaRegisterFatBinary*, all PTX and machine code is registered with the CUDA runtime. WASTE captures the PTX code, and calls a parser to create an internal representation of the PTX assembly code suitable for emulation.

The PTX parser in WASTE produces an abstract syntax tree (AST) that represents the parse of the PTX. WASTE uses only the PTX assembly language code for emulation. It performs this analysis regardless whether all the kernel and device functions are actually called. The PTX code must be valid assembly code.

The parser for PTX is specified as an Antlr grammar, in the file `ptxg/Ptx.g`. Antlr (antlr.org, <http://en.wikipedia.org/wiki/ANTLR>) is a general-purpose parser generator for language recognition. One reason for choosing Antlr for the parser was because it can be debugged using Antlrworks (<http://antlr.org/works/index.html>). The grammar for PTX is LL(k) with over 600 terminal and non-terminal productions. The grammar specifies not only valid PTX syntax, but contains rules for the construction of the AST. The start rule for the grammar is the non-terminal symbol “prog”.

Let’s take as example the rule for the ABS instruction. There are two forms for the instruction: integer and floating point. The integer form syntax is described in the PTX manual as:

```

abs.type d, a;
.type = { .s16, .s32, .s64 };

```

The floating point form syntax is:

```
abs{.ftz}.f32 d, a;  
abs.f64 d, a;
```

Since the grammar allows backtracking, it is also possible to define two different non-terminals with a common handle prefix. However, since the grammar is LL, it is probably best to define a common non-terminal, and use alternative sub-rules for the two forms, so it can be compatible with other LL parser generators.

In Antlr, there are two ways to specify the AST construction: operators and rewrite rules. The PTX grammar does a little of both. However, Antlr restricts the use of the tree construction methods two different ways. First, tree construction can only be done by one technique per production, i.e., via operators or via rewrite rules, but not both. Second, Antlr does not allow an entire sub-tree to be labeled then referenced in rewrite rules. I.e., it is not possible to reference the type of the ABS instruction this way:

```
i_abs  
: i=KI_ABS  
  t=(  
    (  
      K_S16 | K_S32 | K_S64  
    ) |  
    (  
      K_FTZ? K_F32  
    ) |  
    (  
      K_F64  
    )  
  )  
  o=(  
    opr_register  
    T_COMMA  
    opr_register_or_constant  
  )  
  -> ^($i ^(TREE_TYPE $t) $o)  
;
```

In order to get around these deficiencies in Antlr, many auxiliary rules were added. The rules for the ABS instruction are:

```
i_abs  
:  
  i=KI_ABS  
  t=i_abs_type  
  o=i_abs_opr  
  -> $i ^(TREE_TYPE $t) $o  
;  
  
i_abs_type
```

```

:
(
  (
    K_S16 | K_S32 | K_S64
  ) |
  (
    K_FTZ? K_F32
  ) |
  (
    K_F64
  )
)
;

i_abs_opr
:
(
  opr_register
  T_COMMA!
  opr_register_or_constant
)
;

```

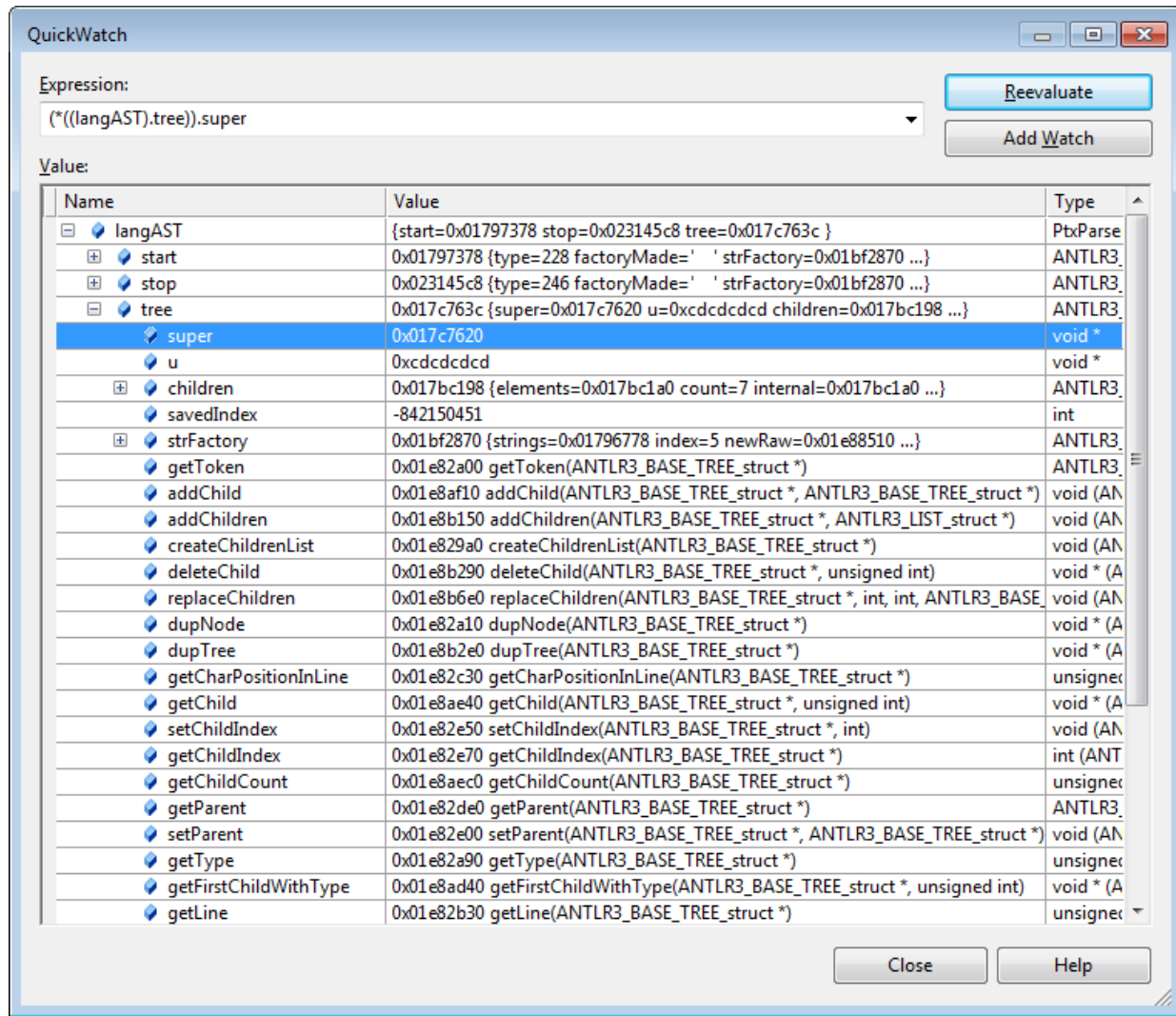
For symbols without an explicit tree name, the AST is just the token itself.

AST – the intermediate representation for WASTE

While Antlr produces a general purpose AST, in this particular instance it was not the best choice.

One problem is debugging the Antrl C runtime data structures. The data structures for the tree are struct ANTLR3_BASE_TREE_struct. This structure contains members that contain pointers to functions for C++ class emulation, a vector of children of the node, a pointer to a super class, etc. Unfortunately, most of the fields are void *. When debugged in Microsoft Visual C++ (Figure 5), there are no fields that directly yield the type and text representation of the node. Typecasting must be done. While it is possible to display typecast data for one or two fields, it is impractical for a large number of them. It makes debugger harder to do.

Figure 5—Debugging the Antlr AST in Microsoft Visual Studio



A second problem is the poor quality of the Antlr C runtime documentation. While Antlr itself is fairly well documented, the C API is not. There is no documentation of how to “subclass” the `ANTLR3_BASE_TREE_struct` structure in order to add application-specific information to the AST, which is how one would add application-specific information with the Java runtime. (The field “u” of the structure is provided to allow user-specific data, but it was discovered too late in the writing of WASTE.) Wading through old forum messages (see <http://www.antlr.org/pipermail/antlr-interest/>) is not informative because many of the messages are several years old, which do not apply to the current version of the product. The Doxygen documentation for the C runtime (http://www.antlr.org/api/C/struct_a_n_t_l_r_3__b_a_s_e__t_r_e_e_struct.html#7ef255261f0dfda0d26ad8fa73bd9c2b) does not describe at all the purpose of the functions in the structure, how to use them. The Doxygen documentation itself contains stale links if you try to follow up on the purpose and use of the functions (e.g., http://www.antlr.org/api/C/antlr3commontreeadaptor_8c-

[source.html#l00427](#)). The *getText* function returns a string that is copied into an allocated buffer each time it is called, but there is no documentation indicating that one needs to free the structure, or how to free it.

```
class TREE
{
    private:
        // A node in the tree has children.
        std::vector<TREE*> children;

        // Each node has a type.
        int type;

        // Every node has a text representation. The text
        // representation is just for the node itself, not for the
        // forest.
        char * text;

        // A node can represent a symbol, e.g., a register. This field is an
        // optimization, which points to the symbol data structure.
        SYMBOL * symbol;

        // The TREE data structure is a general tree. Every node has a
        // parent.
        TREE * parent;
    public:
        TREE();
        int GetType();
        char * GetText();
        TREE * GetChild(int index);
        int GetChildCount();
        void SetText(char * text);
        void SetType(int type);
        void AddChild(TREE * child);
        SYMBOL * GetSymbol();
        void SetSymbol(SYMBOL * symbol);
        TREE * GetParent();
        void SetParent(TREE * parent);
};
```

To avoid these problems, the TREE class was created to represent the AST for WASTE during emulation.

There are roughly 285 tree node types for tokens in the grammar, and another 24 or so tree node types for nodes that are not tokens in the grammar. For example, tree nodes that correspond to tokens in the grammar are:

"KI_ABS"	a node containing the name "abs", of type KI_ABS, used in ABS instructions;
"K_CC"	a node containing the name ".cc", of type K_CC, used in ADD.CC instructions;
"K_S32"	a node containing the name ".s32", of type K_S32, used in many instructions.

Examples of tree nodes that do not correspond to tokens in the grammar are:

- “TREE_OPR” a node containing no name, of type KI_ABS, used in instructions for each operand;
- “TREE_INST” a node containing no name, of type TREE_INST, used for each instruction;
- “TREE_ENTRY” a node containing no name, of type TREE_ENTRY, used to start a kernel.

Each instruction is encoded with an internal node TREE_INST. The number and types of children nodes depends on the instruction. TREE_INST nodes always represent a single instruction in the PTX code. The children of the TREE_INST node represent parts of the instruction. The first child of the TREE_INST node is always the instruction name. TREE_TYPE nodes are used to represent a type. The remainder of the children of the TREE_INST node are usually TREE_OPR nodes, which are used to represent the operands of the instruction. Since the form of the sub-tree depends on the instruction, the emulator pays close attention to finding and validating the information contained in the sub-tree.

In order to get a good picture of how what the tree looks like for a real program, the AST for a simple example is now explained (see Figure 1a). This program performs the absolute value of -1. The program launches one block, with one thread in the block. When the program is compiled with the `-gencode=arch=compute_20,code=\"sm_20,compute_20\"` and the `-keep` options, the compiler generates a PTX file with five PTX instructions (Figure 4). When parsed using the Ptx.g grammar, the parser will produce an AST (Figure 1c). In particular, note the sub-tree for the ABS instruction. The next child for the ABS instruction is the TREE_TYPE node. For the ABS instruction, the type is “.S32”.

The AST for the PTX code for the example is shown in Figure 6.

Figure 6—The AST for the ABS Example

```

nil
.version
  2.2
.target
  sm_20
TREE_ENTRY
  _Z3funPi
  TREE_PARAM_LIST
  TREE_PARAM
    __cudaparm__Z3funPi_mem
  TREE_TYPE
    .u32
  TREE_PERF
  TREE_BLOCK
  TREE_VAR
  TREE_SPACE
  .reg
  TREE_TYPE
    .u32
  %r
  TREE_PAR_REGISTER
  9
  TREE_INST
  ld
  TREE_TYPE
    .param
    .u32
  TREE_OPR
    %r1
  TREE_OPR
    __cudaparm__Z3funPi_mem

```

```

TREE_INST
ldu
TREE_TYPE
.global
.s32
TREE_OPR
%r2
TREE_OPR
%r1
+
TREE_CONSTANT_EXPR
0
TREE_INST
abs
TREE_TYPE
.s32
TREE_OPR
%r7
TREE_OPR
%r2
TREE_INST
st
TREE_TYPE
.global
.s32
TREE_OPR
%r1
+
TREE_CONSTANT_EXPR
0
TREE_OPR
%r7
TREE_INST
exit
null

```

The output of the AST was obtained from a stand-alone version of the parser, ptxp/Doit.java.

One of the advantages of the Antlr is the ability to view the AST using Antrworks, using “java -jar antlrworks-1.4.jar”. For example, for a simple PTX file containing show in the left-hand side panel, the AST for the input is shown in the right-hand side panel. In this example, the instruction “abs.s32 %r7, %r2;” has an AST representation starting at the node labeled TREE_INST (Figure 7).

Figure 7—Antlrworks debugging ptxp.g

ANTLRWorks 1.4

File Edit Find Go To Grammar Refactor Generate Run Window Help

C:\Personal\tem\cuda-waste\ptxp\ptxp.g

ptxp.g

```
prog
{
    .version 2.2;
    .target sm_20;
    .entry _Z3funPi (
        .param .u32 __cudaparm__Z3funPi_mem)
    {
        .reg .u32 %r<9>;
        ld.param.u32 %r1, [__cudaparm__Z3funPi_mem];
        ldu.global.s32 %r2, [%r1+0];
        abs.s32 %r7, %r2;
        st.global.s32 [%r1+0], %r7;
        exit;
    } // _Z3funPi
}
```

Break on: ☐ All ☐ Location ☒ Consume ☐ LT ☐ Exception

Break on Terminate

Input

AST

Stack

Syntax Diagram Interpreter Console Debugger

656 rules 358:5 Writable 88M of 259M

String table

All strings in WASTE are entered into a table of strings. This table is used to minimize duplicate strings throughout the program. The string table is implemented using `std::set<>`.

```
class STRING_TABLE
{
public:
    STRING_TABLE() {}

    ~STRING_TABLE()
    {
        for (std::set<char *, ltstr>::iterator it = this->table.begin();
             it != this->table.end(); ++it)
        {
            free(*it);
        }
        this->table.clear();
    }

    char * Entry(char * text)
    {
        char * result = 0;
        std::set<char *, ltstr>::iterator it = this->table.find(text);
        if (it == this->table.end())
        {
            char * the_text = strdup(text);
            this->table.insert(the_text);
            result = the_text;
        }
        else
        {
            result = *it;
        }
        return result;
    }

private:
    struct ltstr
    {
        bool operator()(const char* s1, const char* s2) const
        {
            return strcmp(s1, s2) < 0;
        }
    };

    std::set<char *, ltstr> table;
};
```

Symbol table

The symbol table is probably the most important part of the emulator. All registers, labels, user-defined and CUDA built-in variables that appear in the PTX assembly language code are managed using the symbol table. Each symbol has a name, size, type, a flag to indicate if it is an array, the size of the array if an array, and a pointer to a dynamically allocated buffer for the value. Symbols are stored in a per-thread basis usually, except when declared as shared or global.

The symbol table is actually a tree structure of individual SYMBOL_TABLE objects. Each SYMBOL_TABLE object contains symbols at a common scope and thread in the CUDA PTX function. All SYMBOL's in SYMBOL_TABLE are stored in a std::map<>, which maps a name to the symbol.

```
class SYMBOL;

class SYMBOL_TABLE
{
private:
    struct ltstr
    {
        bool operator()(const char* s1, const char* s2) const
        {
            return strcmp(s1, s2) < 0;
        }
    };

public:
    std::map<char *, SYMBOL *, ltstr> symbols;
    SYMBOL_TABLE * parent_block_symbol_table;
    SYMBOL_TABLE();
    ~SYMBOL_TABLE();
    void Dump();
    SYMBOL * FindSymbol(char * name);
    void EnterSymbol(SYMBOL * sym);
    SYMBOL_TABLE(const SYMBOL_TABLE & original);
        void CachePvalues();
    void CheckCachedPvalues();
};

class SYMBOL
{
public:
    char * name;
    void * pvalue;
        void * cache;//for debugging.
    size_t size;
    char * typestring;
    int type;
    bool array;
    size_t index_max;
    int storage_class;
    EMULATOR * emulator;
    ~SYMBOL();
};
```

Read, dispatch, execute loop

When a CUDA kernel is called via `cudaError_t EMULATOR::_cudaLaunch(const char *hostfun)` is called, which is the top-level routine for emulating PTX. WASTE finds the corresponding PTX AST for *hostfun*, then starts executing the PTX instructions for the kernel function.

First, WASTE creates a symbol table structure with some predefined, global, const, and texture variables. Then, for each block in the grid, WASTE executes the PTX:

```
for (int bidx = 0; bidx < conf.gridDim.x; ++bidx)
{
    for (int bidy = 0; bidy < conf.gridDim.y; ++bidy)
    {
        for (int bidz = 0; bidz < conf.gridDim.z; ++bidz)
        {
            ExecuteSingleBlock(block_symbol_table, do_thread_synch, code, bidx, bidy, bidz);
        }
    }
}
```

For each block, WASTE creates a symbol table for shared variables, then creates threads for each block. Each thread has a symbol table that contains registers, locals, aligned, and parameter variables.

```
for (int tidx = 0; tidx < conf.blockDim.x; ++tidx)
{
    for (int tidy = 0; tidy < conf.blockDim.y; ++tidy)
    {
        for (int tidz = 0; tidz < conf.blockDim.z; ++tidz)
        {
            SYMBOL_TABLE * root = PushSymbolTable(block_symbol_table);
            dim3 tid(tidx, tidy, tidz);
            CreateSymbol(root, "%tid", "dim3", K_V4, &tid, sizeof(tid), K_LOCAL);
            if (do_thread_synch)
            {
                int sc[] = { K_REG, K_LOCAL, K_ALIGN, K_PARAM, 0 };
                SetupVariables(root, code, sc);
            }
            THREAD * thread = new THREAD(this, code, 0, root);
            wait_queue.push(thread);
        }
    }
}
```

After creating threads, threads are executed, two concurrent threads at a time. Within each thread, WASTE does a “read, dispatch, execute” loop (in `void THREAD::Execute()`) to execute the PTX in the block of code.

```
void THREAD::Execute()
{
    // set up symbol table environment.
    int pc = this->pc;

    // Execute.
```

```

pc = emulator->FindFirstInst(block, pc);
if (pc < 0)
{
    this->finished = true;
    return;
}
for (;;)
{
    TREE * inst = this->emulator->GetInst(block, pc);
    if (this->emulator->trace_level > 3)
        this->Dump("before", pc, inst);

    // if debug, check if pvalues in each symbol was changed. It
    // should have not!
    if (this->emulator->trace_level > 0)
        this->root->CachePvalues();

    int next = this->Dispatch(inst);

    if (this->emulator->trace_level > 0)
        this->root->CheckCachedPvalues();

    if (next > 0)
        pc = next;
    else if (next == -KI_EXIT)
    {
        this->finished = true;
        return;
    }
    else if (next == -KI_BAR)
    {
        // Set state of this thread to wait, and pack up current program counter.
        this->wait = true;
        this->pc = pc + 1;
        return;
    }
    else
        pc++;

    pc = this->emulator->FindFirstInst(block, pc);

    if (this->emulator->trace_level > 2)
        this->Dump("after", pc, inst);
}
}

```

The code to execute an instruction is generally fairly straight forward. Each instruction is first checked for validity. Then, after verifying the information for the instruction, the state of the PTX machine (contained in the symbol table) is updated.

For example, for the ABS instruction, *int THREAD::DoAbs(TREE * inst)* is called (Figure 8). The internal node for the AST for the instruction is variable *inst*. The first child of the tree can be a predicate. Since

the value of the predicate is checked in `void THREAD::Execute()` and is true, function *DoAbs* ignores this part of the AST. Next, the function checks that the next child is a `KI_ABS` node. If not, then there was an error in the dispatch code. Next, the function checks the existence of the source and destination operands, and extracts the type of the operation, e.g., integer, float, etc. If the AST for the instruction in any way does not appear valid, execute will halt. For example `ABS.FTZ` is not implemented yet, so there is a check for this condition. Next, the destination operand is fetched. It must be a register, which is represented using a `T_WORD` node. The name of the register is stored in the text field of the node. The symbol is fetched from the symbol table (`sdst = this->root->FindSymbol(dst->GetText());`). The source operand can be either a constant expression, or a symbol. The source operand is then found. Finally, the absolute value of the source is assigned to the destination, based on the type of the instruction.

Figure 8—Function DoAbs of WASTE

```
int THREAD::DoAbs(TREE * inst)
{
    int start = 0;
    if (inst->GetChild(start)->GetType() == TREE_PRED)
        start++;
    assert(inst->GetChild( start)->GetType() == KI_ABS);
    start++;
    TREE * ttype = 0;
    TREE * odst = 0;
    TREE * osrc1 = 0;
    for (; ++start)
    {
        TREE * t = inst->GetChild(start);
        if (t == 0)
            break;
        int gt = t->GetType();
        if (gt == TREE_TYPE)
            ttype = t;
        else if (gt == TREE_OPR)
        {
            if (odst == 0)
            {
                odst = t;
            } else if (osrc1 == 0)
            {
                osrc1 = t;
            } else assert(false);
        } else assert(false);
    }
    assert(ttype != 0);
    assert(odst != 0);
    assert(osrc1 != 0);
    bool ftz = false;
    for (int i = 0; ; ++i)
    {
        TREE * t = ttype->GetChild(i);
        if (t == 0)
            break;
        int gt = t->GetType();
        if (gt == K_S16 || gt == K_S32 || gt == K_S64)
            ttype = t;
        else if (gt == K_F32 || gt == K_F64)
            ttype = t;
        else if (gt == K_FTZ)
            ftz = true;
    }
}
```

```

    else assert(false);
}
assert(ttype != 0);
this->emulator->unimplemented(ftz, "ABS.ftz not implemented.");

int type = ttype->GetType();
TREE * dst = odst->GetChild(0);
TREE * src1 = osrc1->GetChild(0);

SYMBOL * sdst = 0;
if (dst->GetType() == T_WORD)
{
    sdst = this->root->FindSymbol(dst->GetText());
} else assert(false);

TYPES::Types value1;
char * dummy;
TYPES::Types * d = (TYPES::Types*)sdst->pvalue;
TYPES::Types * s1 = &value1;

if (src1->GetType() == TREE_CONSTANT_EXPR)
{
    CONSTANT c = this->emulator->Eval(type, src1->GetChild(0));
    switch (type)
    {
        case K_S16:
            s1->s16 = c.value.s16;
            break;
        case K_S32:
            s1->s32 = c.value.s32;
            break;
        case K_S64:
            s1->s64 = c.value.s64;
            break;
        case K_F32:
            s1->f32 = c.value.f32;
            break;
        case K_F64:
            s1->f64 = c.value.f64;
            break;
        default:
            assert(false);
    }
} else if (src1->GetType() == T_WORD)
{
    SYMBOL * ssrc1 = this->root->FindSymbol(src1->GetText());
    assert(ssrc1 != 0);
    assert(ssrc1->size == this->emulator->Sizeof(type));
    TYPES::Types * psrc1_value = (TYPES::Types*)ssrc1->pvalue;
    switch (type)
    {
        case K_S16:
            s1->s16 = psrc1_value->s16;
            break;
        case K_S32:
            s1->s32 = psrc1_value->s32;
            break;
        case K_S64:
            s1->s64 = psrc1_value->s64;
            break;
        case K_F32:
            s1->f32 = psrc1_value->f32;
            break;
    }
}

```

```

        case K_F64:
            s1->f64 = psrc1_value->f64;
            break;
        default:
            assert(false);
    }
} else assert(false);

switch (type)
{
    case K_S16:
        d->s16 = abs(s1->s16);
        break;
    case K_S32:
        d->s32 = abs(s1->s32);
        break;
    case K_S64:
        d->s64 = abs(s1->s64);
        break;
    case K_F32:
        d->f32 = abs(s1->f32);
        break;
    case K_F64:
        d->f64 = abs(s1->f64);
        break;
    default:
        assert(false);
}
return 0;
}

```

Conclusions

WASTE is a simple emulator for PTX. The purpose of WASTE is to provide a simple means of testing CUDA code on a PC that does not contain an NVIDIA GPU. While it works on many programs, the run times for emulation are several orders of magnitude times slower than the equivalent of a serial implementation on a CPU. That said, WASTE does provide additional tracing of memory usage that is not normally found in the CUDA Toolkit.

Many problems encountered in writing WASTE:

- 1) The most prevalent problem was understanding the syntax and the semantics for each instruction. The documentation of the PTX instruction set is not clear, and contains many errors.
- 2) Hooking of the CUDA Driver API is incomplete. The function `cuGetExportTable` creates data structures used by `cuMemAlloc`, and is not described in any documentation.
- 3) Floating point rounding is not explained in any of the documentation.
- 4) The V2 functions in the CUDA DLL's are not explained.

Modules, Classes, and Methods of WASTE

WASTE

This module is an executable that performs DLL injection and API hooking into the CUDA Runtime and Driver API. It performs a CreateProcess Windows call to spawn the program. The spawned process is first placed in suspended mode. X86 machine code is created and injected into the process. This code initializes Wrapper, the main hooking API for the emulator.

Functions

<code>int main(int argc, char * argv[])</code>	Function main is the driver for the WASTE executable, which performs DLL injection and option setting of the user program.
<code>void AddBytes(byte * code, byte a)</code> <code>void AddBytes(byte * code, byte a, byte b)</code> <code>void AddBytes(byte * code, byte a, byte b, byte c)</code> <code>void AddBytes(byte * code, byte a, byte b, byte c, byte d)</code> <code>void AddBytes(byte * code, byte a, byte b, byte c, byte d, byte e)</code>	Functions to insert x86 machine code into a buffer.

Wrapper

This module contains the code to do API hooking for the CUDA API, packaged as a DLL, and used by program WASTE. The modules Emulator and Ptxp are linked into this DLL.

Classes

<code>_CUDA</code>	Wrapper functions for the CUDA Driver API, e.g., <code>culniit</code> .
<code>_CUDA_RUNTIME</code>	Wrapper functions for the CUDA Runtime API, e.g., <code>cudaMalloc</code> .
<code>CALL_STACK_INFO</code>	Contains functions for call stack tracing. This functionality is used for debug output.
<code>CUDA_WRAPPER</code>	This class contains the code that sets up API hooking for the CUDA libraries, memory tracing functions, and option handling.
<code>LOCK_MANAGER</code>	The lock manager is used for locks in the multithread application.
<code>PROCESS_MANAGER</code>	The process manager gets information about the processes.
<code>MEMDBG</code>	WASTE contains this class to trace its own memory usage.

Emulator

This module contains the code for the emulator, packaged as a static library, and is linked into the wrapper.dll.

Classes

<code>CONSTANT</code>	This class represents constants within the CUDA
-----------------------	---

	PTX assembly language, and has a function to evaluate constant expressions.
EMULATOR	This large class is the façade for the WASTE emulator.
STRING_TABLE	WASTE stores all “char*” strings in a table.
SYMBOL	The interpretation of CUDA PTX code essentially modifies a machine state, which is represented with SYMBOL and SYMBOL_TABLE. Class SYMBOL represents one register or variable in the PTX code.
SYMBOL_TABLE	The SYMBOL_TABLE class, which contains SYMBOL class objects, represents the state of the GPU.
THREAD	This class represents on thread in a PTX block. It contains a SYMBOL_TABLE class object for the state of the tread.
TREE	This class is used to represent the abstract syntax tree of a PTX program.
TYPES	This class is used to convert values from one type to another in PTX code.

Ptxp

This module contains the code for parsing PTX code, packaged as a static library, and is linked into the wrapper.dll.

Functions

TREE * convert(pANTLR3_BASE_TREE node)	This function converts the Antlr tree into the WASTE-internal representation (TREE).
TREE * parse(char * ptx_module)	This function parses the PTX source code and returns the AST. It calls the convert function.

References

1. Collange, S., Daumas, M., Defour, D. and Parello, D., Barra: A Parallel Functional Simulator for GPGPU. in *18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, (2010), IEEE, 351-360.
2. Damos, G. The design and implementation ocelot’s dynamic binary translator from ptx to multi-core x86, Georgia Tech, 2009.
3. Stratton, J., Stone, S. and Hwu, W. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. *Languages and Compilers for Parallel Computing*. 16-30.