

A Java Library for the Generation and Scheduling of PTX Assembly

Christos Kartsaklis
christos.kartsaklis@ichec.ie
Irish Centre for High-End Computing
(ICHEC)

Gilles Civario
gilles.civario@ichec.ie
Irish Centre for High-End Computing
(ICHEC)

Abstract

This paper discusses an ongoing progress regarding the development of a Java-based library for rapid kernel prototyping in NVIDIA PTX and PTX instruction scheduling. It is aimed at developers seeking total control of emitted PTX, highly parametric emission of, and tunable instruction reordering. It is primarily used for code development at ICHEC but is also hoped that NVIDIA GPU community will also find it beneficial.

1 Introduction

Tuning computing kernels entails the manipulation of certain parameters, such as the depth of unrolling, the stages in pipelining, the order and stride of memory accesses, etc. Achieving this task directly from within the source of a high-level language, such as C (in terms of `#pragmas` and `#defines`), is often laborious or insufficient.

This is often addressed in terms of code generation. Typically, a front end takes as input a set of kernel-tuning parameters and generates the desired output (e.g. C generating C). Programming at the assembly level is more intricate, especially when fine tuning and a large number of instructions are involved. BAGEL [1], for instance, expresses assembly at a high level (hereafter referred to as HLA), programs it and reorders it in C, and outputs “proper” assembly in an order that a built-in algorithm decides.

PTX (Parallel Thread Execution) is a Virtual Machine developed by NVIDIA and is accompanied by an ISA (Instructions Set Architecture) that exposes the Tesla hardware in a highly non implementation-specific manner. The `nvcc` compiler translates CUDA C/C++ to PTX assembly, while `ptxas` turns that into implementation-specific machine code.

We are currently developing a library for PTX code generation and scheduling, called JASM, to acceler-

ate our own GPU porting efforts; in the process of doing so we are gradually enhancing the library with compiler-level code transformations. We believe that since NVIDIA GPGPU computing is well embraced, with manufacturers, such as Bull¹, offering high-end solutions, such a tool will highly assist the fine-tuning process.

2 Current Feature Set

A summary of the features implemented so far is given below.

- Object-Oriented and highly-reflective ISA-independent instruction and register file modelling.
- Modular instruction reordering logic built atop of a constraints solver for basic blocks currently.
- Stackable, pre-defined or user-specified, dependencies analysis and cost function modules.
- Static Single Assignment (SSA) support for instruction that do not have *INOUT* arguments.
- Software register renaming built on top of SSA.
- High-level constructs oriented around software pipelining.
- Pointer-aliasing and multi memory-space infrastructure.

Architecture-independently, instruction arguments are expressed in terms of an instruction descriptor (register, label, immediate, etc); instructions, then, have their own descriptor that is expressed in terms of IN, OUT and IN-OUT, argument descriptors. Register files are, in OO terms, register factories and are classifiable as space- and type-limited (in PTX, for instance, the register file

¹<http://www.bull.com/hpc/nvidia-tesla.html>

is space-limited where from type-specific registers are drawn from).

We find that employing a common algorithm for instruction reordering, such as list-scheduling, is undesirable for the mere reason that it complicates/restricts the addition of future dependencies. Hence, a generic reordering logic has been built atop of an existing constraints solver [2], where dependencies, that ultimately affect instruction motion, are introduced modularly by the programmer. Pre-defined modules, such as register data hazards, exist.

The solver performs full search space exploration given a user-specified or pre-defined cost function. For instance, the GT200 SMP² cores are single-issue, and thus reordering in order to co-dispatch instructions for execution in the same cycle is unimportant. However, as double-precision instructions are executed by a single unit (residing at the SMP), when multiple threads from the same half-warp issue such instructions, they get serialised and thus not executed in a SIMD fashion.

We have implemented SSA (Single Static Assignment) versioning for the arguments of the instructions and a transformation for software-based register renaming in basic blocks. This we find practical as PTX is plentiful in register space (i.e. spilling logic is often not needed) and also because, as code grows in length, renaming opportunities are hard to spot manually.

A certain inconvenience with software pipelining is the number of “steps” involved, as this determines the shape of the prologue/epilogue and the number of iterations in the body of the loop; modifying the number of steps³ requires a major rewrite of the code. We have implemented a high-level construct, called the *HLA Snippet* that reflects (to the programmer) these concerns in a very transparent manner. Essentially, the programmer can design the rest of the code as a function of the snippet.

Pointer aliasing refers to the situation where two or more pointers have the same value; mixed loads and stores against addresses that cannot be proven to be exclusive, leads to serialisation of the code. We also recognise that, accesses may be naturally disjoint because they address data residing in physically different memories; this is the case with PTX and the various memory spaces (global, shared, etc.).

In our work, we assume that the programmer knows the aliased status of accesses and that she is rather concerned with efficiently hinting the reordering logic. The construct we have crafted, to achieve this task, is called an *Addressable Memory Region* (AMR). This is essentially a tree structure that achieves aliased status determination in a single scan of the tree.

²Streaming Multi-Processors (MPs).

³By simplifying or introducing more dependencies.

Emphasis will now be given to the HLA Snippet and AMR constructs.

2.1 HLA Snippets

An *HLA Snippet* is a reflective template of how a multi-instruction functionality is implemented, what dependencies lie therein, and what the register requirements are. To elaborate, consider the PTX source below that implements a modulo operation ($\%r12 = \%r4 \% 3$) in software (emitted by `nvcc`); assume that it is part of a loop that needs software-pipelining:

```
01-a: mov.s32          %r5, -1431655765;
01-b: mov.s32          %r6, 0;
02-a: setp.lt.s32     %p1, %r4, %r6;
02-b: abs.s32         %r7, %r4;
03:   mul.hi.u32      %r8, %r7, %r5;
04:   shr.s32         %r9, %r8, 1;
05:   @%p1 sub.s32    %r9, %r6, %r9;
06:   mov.s32         %r10, %r9;
07:   mul.lo.s32     %r11, %r10, 3;
08:   sub.s32        %r12, %r4, %r11;
```

The snippet will have 8 instructions sets, with all, apart from the first and the second, being single-instruction sets. This information is available to the programmer so she can decide how many software pipeline stages are necessary. Accessing this value is very practical as it affects the increments against the loop index variable in addition to the number of iterations predating the pipelined loop in case the loop length is not ideal. What the snippet also exposes is the need for 8 registers (one is a predicate), excluding the `%r4` and `%r12` that would appear as *IN* and *OUT* arguments, respectively, to the snippet. Thus, the body of the pipelined loop requires $8 \times 8 = 64$ registers.

Part of the reflective capabilities of the snippet is to propose alternative snippets that implement the same functionality with different requirements. For instance, the modulo operation can be implemented in terms of the `rem` instruction. One would utilise this snippet when outside a loop in order to reduce register pressure; this is often necessary in PTX if the kernel is desired to be scalable to multiple threads.

2.2 Memory Access Ordering

Programmers use compiler `#pragmas` (e.g. `disjoint` and `aliased`) to hint the compiler of aliasing issues. In JASM, we address this matter in a more elegant way, called the *Addressable Memory Region* (AMR). Memory IO instructions can be associated with an AMR to signify an aliasing issue.

An AMR occurs as a node in a tree structure with a single operation defined for it – *split(k)*. A split of a node *N*

by k creates k children N_1, \dots, N_k such that each instruction associated with a node N_i is considered disjoint with those associated with the siblings of N_i . It can be shown that given a node N in such a tree, an instruction that is associated with a node N is guaranteed to be disjoint with any instruction known to the tree, apart from those that are associated with nodes that are present in the path from the root node to N , inclusive of both.

For instance, consider the following operation where all memory accesses are global-based:

```
__global__ void foo(
    int *aA, int aN) {
    for (int lI=1 ; lI<aN ; lI+=2) {
#pragma unroll 2
        for(int lJ=lI ; lJ<lI+4 ; lJ++)
            aA[lJ] = aA[lJ] + aA[lJ-1];
    }
}
```

The innermost loop is shown below and has been obtained from `nvcc 2.1` (branching has been omitted for clarity); the base of `aA` is held in `%rd2`:

```
// 01: load from aA[lJ-1];
ld.global.s32    %r10, [%rd2+-4];
// 02: load from aA[lJ];
ld.global.s32    %r11, [%rd2+0];
add.s32          %r12, %r10, %r11;
// 03: store in aA[lJ];
st.global.s32    [%rd2+0], %r12;
// 04: load from aA[lJ+1];
ld.global.s32    %r13, [%rd2+4];
add.s32          %r14, %r13, %r12;
// 05: store in aA[lJ+1];
st.global.s32    [%rd2+4], %r14;
```

The concern here is with the third load (04), which we know that can be safely pushed to the top of the instruction stream, and the first store (03) that can be scheduled later. The aliased accesses are also visible (2nd load with 1st store, and last load with last store).

To model this information in terms of AMRs, we declare a root AMR, say N_* and *split*(3) it into $N_{*.1}$, $N_{*.2}$ and $N_{*.3}$, to correspond to locations $\&aA[lJ-1]$, $\&aA[lJ]$ and $\&aA[lJ+1]$. As siblings are, by definition, disjoint regions, we associate the instructions as follows: $01 \in N_{*.1}$; $02, 03 \in N_{*.2}$; and $04, 05 \in N_{*.3}$.

With this information, the reordering logic knows that, as long as other dependencies (e.g. data register hazards) are satisfiable, the instructions from each AMR node can be reordered with respect to instructions stemming from other nodes. That is, the load in 04 can precede the store in 03 because they belong to sibling nodes $N_{*.3}$ and $N_{*.2}$, respectively.

3 In-Development Feature Set

In this section, we discuss a number of features that are currently under development:

- Structuring of HLA in a CFG (Control Flow Graph); toggling between branching and predicated execution.
- Reverse compilation of PTX source to HLA and cross-compilation of HLA to CUDA C.

There is a compromise between conditional and predicated execution. While conditional execution stalls divergent threads, but makes space for ready-to-run threads from other warps, predicated execution “occupies” divergent threads, but prohibits other threads from doing “useful” work. The right choice is obtained from the profiling of various configurations. We are currently implementing predication of whole branches in the CFG aiming to offer an HLA construct that performs the transformation with minimal code modification effort. Most of the work regards the predication of CFG branches that entail nested branching; this requires extra instructions for predicate arithmetics.

We are working on a translator for `nvcc`-emitted PTX to HLA and HLA to CUDA C for reasons of performance analysis and portability, respectively. This effort has guided the CFG structure greatly as keeping it minimal is paramount for obtaining a CFG from PTX without many complications. With respect to practicality, the predication toggling function finds great applicability in CUDA C cross-compilation, where the programmer is limited to branching only.

4 Conclusions

The purpose of this paper has been to share with the NVIDIA GPGPU community our efforts in providing a complementary (to `nvcc`) tool for kernel tuning and development at the PTX level. Although a number of basic features are already available, including the ability to emit reordered PTX code, we continue to improve functionality. Finally, a C++ branch of the tool is under consideration once development has been stabilised.

References

- [1] Peter Boyle. BAGEL. <http://www.ph.ed.ac.uk/~paboyle/bagel/Bagel.html>, 2005.
- [2] Naoyuki Tamura. Cream. <http://bach.istc.kobe-u.ac.jp/cream/>, 2003.