USING INLINE PTX ASSEMBLY IN CUDA

Application Note

DOCUMENT CHANGE HISTORY

DA-05713-001_v01

Version	Date	Authors	Description of Change
01	February 14, 2011	CUDA	Initial release

TABLE OF CONTENTS

USING INLINE PTX ASSEMBLY IN CUDA

The NVIDIA® CUDA™ programming environment provides a parallel thread execution (PTX) instruction set architecture (ISA) for using the GPU as a data-parallel computing device. For more information on the PTX ISA, refer to the latest version of the PTX ISA reference document *ptx_isa_[version].pdf* in the CUDA Toolkit doc folder.

This application note describes how to inline PTX assembly language statements into CUDA code.

ASSEMBLER (ASM) STATEMENTS

Assembler statements, asm(), provide a way to insert arbitrary PTX code into your CUDA program. A simple example is:

```
asm("membar.gl;");
```

This inserts a PTX membar.gl into your generated PTX code at the point of the asm() statement.

Parameters

An asm() statement becomes more complicated, and more useful, when we pass values in and out of the asm. The basic syntax is as follows:

```
asm("template-string" : "constraint"(output) : "constraint"(input));
```

where you can have multiple input or output operands separated by commas. The template string contains PTX instructions with references to the operands. Multiple PTX instructions can be given by separating them with semicolons.

A simple example is as follows:

```
asm("add.s32 %0, %1, %2;" : "=r"(i) : "r"(j), "r"(k));
```

Each %n in the template string is an index into the following list of operands, in text order. So %0 refers to the first operand, %1 to the second operand, and so on. Since the output operands are always listed ahead of the input operands, they are assigned the smallest indices. This example is conceptually equivalent to the following:

```
add.s32 i, j, k;
```

Note that the numbered references in the string can be in arbitrary order. The following is equivalent to the above example:

```
asm("add.s32 %0, %2, %1;" : "=r"(i) : "r"(k), "r"(j));
```

You can also repeat a reference, e.g.:

```
asm("add.s32 %0, %1, %1;" : "=r"(i) : "r"(k));
```

is conceptually

```
add.s32 i, k, k;
```

If there is no input operand, you can drop the final colon, e.g.:

```
asm("mov.s32 %0, 2;" : "=r"(i));
```

If there is no output operand, the colon separators are adjacent, e.g.:

```
asm("mov.s32 r1, %0;" :: "r"(i));
```

If you want the % in a ptx instruction, then you should escape it with double %%, e.g.:

```
asm("mov.u32 %0 %%clock;" : "=r"(x));
```

The above was simplified to explain the ordering of the string % references. In reality, the operand values are passed via whatever mechanism the constraint specifies. The full list of constraints will be explained later, but the "r" constraint refers to a 32bit integer register. So the earlier example asm() statement:

```
asm("add.s32 %0, %1, %2;" : "=r"(i) : "r"(j), "r"(k));
```

produces the following code sequence in the output generated by the compiler:

```
ld.s32 r1, [j];
ld.s32 r2, [k];
add.s32 r3, r1, r2;
st.s32 [i], r3;
```

This is where the distinction between input and output operands becomes important. The input operands are loaded into registers before the asm() statement, then the result register is stored to the output operand. The "=" modifier in "=r" specifies that the register is written to. There is also available a "+" modifier that specifies the register is both read and written, e.g.:

```
asm("add.s32 %0, %0, %1;" : "+r"(i) : "r" (j));
```

Multiple instructions can be combined into a single asm() statement; basically, anything legal can be put into the asm string. Multiple instructions can be split across multiple lines by making use of C/C++'s implicit string concatenation. Both C++ style line end comments "//" and classical C-style comments "/**/" can be interspersed with these strings. To generate readable output in the PTX intermediate file it is best practice to terminate each instruction string except the last one with "\n\t".

For example, a cube routine could be written as:

```
__device__ int cube (int x)
{
  int y;
  asm(".reg .u32 t1;\n\t"              // temp reg t1
      " mul.lo.u32 t1, %1, %1;\n\t"    // t1 = x * x
      " mul.lo.u32 %0, t1, %1;"        // y = t1 * x
      : "=r"(y) : "r" (x));
  return y;
}
```

Constraints

There is a separate constraint letter for each PTX register type:

```
"h" = .u16 reg
"r" = .u32 reg
"l" = .u64 reg
"f" = .f32 reg
"d" = .f64 reg
```

For example:

```
asm("cvt.f32.s64 %0, %1;" : "=f"(x) : "l"(y));
```

will generate:

```
ld.s64 rd1, [y];
cvt.f32.s64 f1, rd1;
st.f32 [x], f1;
```

Note that there are some constraints supported in earlier versions of the compiler like "m", "s", and "n" that are not guaranteed to work across all versions and thus should not be used.

PITFALLS

Although asm() statements are very flexible and powerful, you may encounter some pitfalls—these are listed in this section.

Namespace Conflicts

If the cube function (described before) is called and inlined multiple times in the code, it generates an error about duplicate definitions of the temp register t1. To avoid this error you need to:

▶ not inline the cube function, or,

▶ nest the t1 use inside {} so that it has a separate scope for each invocation, e.g.:

```
__device__ int cube (int x)
{
  int y;
  asm("{\n\t"                        // use braces for local scope
      " reg .u32 t1;\n\t"            // temp reg t1,
      " mul.lo.u32 t1, %1, %1;\n\t"  // t1 = x * x
      " mul.lo.u32 %0, t1, %1;\n\t"  // y = t1 * x
      "}"
      : "=r"(y) : "r" (x));
  return y;
}
```

Note that you can similarly use braces for local labels inside the asm() statement.

Memory Space Conflicts

Since asm() statements have no way of knowing what memory space a register is in, the user must make sure that the appropriate PTX instruction is used. For sm_20 and greater, any pointer argument to an asm() statement is passed as a generic address.

Incorrect Optimization

The compiler assumes that an asm() statement has no side effects except to change the output operands. To ensure that the asm is not deleted or moved, you should use the volatile keyword, e.g.:

```
asm volatile ("mov.u32 %0 %%clock;" : "=r"(x));
```

Normally any memory that is written to will be specified as an out operand, but if there is a hidden side effect on user memory, or if you want to stop any memory optimizations around the asm() statement, you can add a "memory" clobbers specification after a 3rd colon, e.g.:

```
asm volatile ("mov.u32 %0 %%clock;" : "=r"(x) :: "memory");
```

Incorrect PTX

The compiler front end does not parse the asm() statement template string and does not know what it means or even whether it is valid PTX input. So if there are any errors in the string it will not show up until ptxas. For example, if you pass a value with an "r" constraint but use it in an add.f64 you will get a parse error from ptxas.

Refer to the document *nvcc.pdf* for further compiler related details.

ERROR CHECKING

The following are some of the error checks that the compiler will do on inline PTX asm:

▶ Multiple constraint letters for a single asm operand are not allowed, e.g.:

```
asm("add.s32 %0, %1, %2;" : "=r"(i) : "rf"(j), "r"(k));
```

error: an asm operand may specify only one constraint letter in a __device__/__global__ function

▶ Only scalar variables are allowed as asm operands. Specifically aggregates like 'struct' type variables are not allowed, e.g.:

```
int4 i4;
asm("add.s32 %0, %1, %2;" : "=r"(i4) : "r"(j), "r"(k));
```

error: an asm operand must have scalar type

▶ The type and size implied by a PTX asm constraint must match that of the associated operand. Example where size does not match:

For 'char' type variable "ci",

```
asm("add.s32 %0,%1,%2;":"=r"(ci):"r"(j),"r"(k));
```

error: asm operand type size(1) does not match type/size implied by constraint 'r'

In order to use 'char' type variables "ci", "cj", and "ck" in the above asm statement, code segment similar to the following may be used,

```
int temp = ci;
asm("add.s32 %0,%1,%2;":"=r"(temp):"r"((int)cj),"r"((int)ck));
ci = temp;
```

Another example where type does not match:

For 'float' type variable "fi",

```
asm("add.s32 %0,%1,%2;":"=r"(fi):"r"(j),"r"(k));
```

error: asm operand type size(4) does not match type/size implied by constraint 'r'