Inline PTX Assembly in CUDA

Release 12.8

NVIDIA Corporation

Feb 27, 2025

Contents

Inline PTX Assembly in CUDA

The reference guide for inlining PTX (parallel thread execution) assembly statements into 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.

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

Chapter 1. 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.

1.1. 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 "nt".

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;
}
```

If an output operand is conditionally updated by the asm instructions, then the "+" modifier should be used. There is an implicit use of the output operand in such a case. For example,

```
__device__ int cond (int x)
{
  int y = 0;
  asm("{\n\t"
      " .reg .pred %p;\n\t"
      " setp.eq.s32 %p, %1, 34;\n\t" // x == 34?
      " @%p mov.s32 %0, 1;\n\t"      // set y to 1 if true
      "}"                            // conceptually y = (x==34)?1:y
      : "+r"(y) : "r" (x));
  return y;
}
```

1.2. Constraints

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

```
"h" = .u16 reg
"r" = .u32 reg
"l" = .u64 reg
"q" = .u128 reg
"f" = .f32 reg
"d" = .f64 reg
```

Example:

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

generates:

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

Note that the constraint `"q"` is only supported on platforms that support `__int128`.

The constraint `"n"` may be used for immediate integer operands with a known value. Example:

```
asm("add.u32 %0, %0, %1;" : "=r"(x) : "n"(42));
```

generates:

```
add.u32 r1, r1, 42;
```

The constraint `"C"` can be used for operand of type 'array of const char', where the array contents are known at compile time. It is intended to allow customization of PTX instruction modes based on compile time computation (see examples). Here is the specification for the `"C"` constraint:

```
'C'(constant-expression)
```

The *constant-expression* is evaluated during compilation and shall generate the address of a variable V, where:

- V has *static storage duration*.
- V has type 'array of const char'.

- V is *constant-initialized*.
- If V is a static class member, then V's *initializing declaration* is the declaration within the class.

During translation, the compiler will replace a reference to the operand within the *Assembler Template* with the contents of V's initializer, except for the last trailing zero. No constraint modifiers are allowed for this constraint. This constraint can only be used in device code.

(terms in *italics* are C++ standard terms and/or terms from the GNU inline asm specification).

Here's an example of the use of C constraint to generate different PTX instruction modes based on compile time computation:

```
constexpr int mode_rz = 0;
constexpr int mode_rn = 1;

template <int mode>
struct helper;

template<> struct helper<mode_rz> {
    static constexpr const char mode[] = ".rz";
};

template<> struct helper<mode_rn> {
    static constexpr const char mode[] = ".rn";
};

template <int rounding_mode>
__device__ float compute_add(float a, float b) {
    float result;
    asm ("add.f32%1 %0,%2,%3;" : "=f"(result)
                               : "C"(helper<rounding_mode>::mode),
                                 "f"(a), "f"(b));
    return result;
}

__global__ void kern(float *result, float a, float b) {
    *result++ = compute_add<mode_rn>(a,b); // generates add.f32.rn
    *result   = compute_add<mode_rz>(a,b); // generates add.f32.rz
}
```

Other examples (compile in C++17 or later dialect):

```
struct S1 {
static constexpr char buf1[] = "Jumped";
static constexpr char buf2[] = {'O', 'v', 'e', 'r', 0};
};

template <const char *p1, const char *p2, const char *p3>
__device__ void doit() {
asm volatile ("%0 %1 %2" : : "C"(p1), "C"(p2), "C"(p3));
}

struct S2 {
static const char buf[];
};
const char S2::buf[] = "this";

const char buf3[] = "Jumped";
```

(continues on next page)

(continued from previous page)

```
extern const char buf4[];

__global__ void foo() {
    static const char v1[] = "The";
    static constexpr char v2[] = "Quick";
    static const char v3[] = { 'B' , 'r' , 'o', 'w', 'n', 0 };
    static constexpr char v4[] = { 'F', 'o', 'x', 0 };

    //OK: generates 'The Quick Brown Fox Jumped Over' in PTX
    asm volatile ("%0 %1 %2 %3 %4 %5" : :  "C"(v1) , "C"(v2), "C"(v3),  "C"(v4), "C
↪"(S1::buf1), "C"(S1::buf2) );

    //OK: generates 'Brown Fox Jumped' in PTX
    doit<v3, v4, buf3>();


    //error cases

    const char n1[] = "hi";

    //error: argument to "C" constraint is not a constant expression
    asm volatile ("%0" :: "C"(n1));

    //error: S2::buf was not initialized at point of declaration
    asm volatile ("%0" :: "C"(S2::buf));

    //error: buf4 was not initialized
    asm volatile ("%0" :: "C"(buf4));
}
```

There is no constraint letter for 8-bit wide PTX registers. PTX instructions types accepting 8-bit wide types permit operands to be wider than the instruction-type size. Example:

```
__device__ void copy_u8(char* in, char* out) {
    int d;
    asm("ld.u8 %0, [%1];" : "=r"(d) : "l"(in) : "memory");
    *out = d;
}
```

generates:

```
ld.u8 r1, [rd1];
st.u8 [rd2], r1;
```

The behavior of using a constraint string that is not one of those specified above is undefined.

Chapter 2. Pitfalls

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

2.1. 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.

2.2. 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.

2.3. 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 during generation of PTX, 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 read or write on user memory (for example, indirect access of a memory location via an operand), or if you want to stop any memory optimizations around the asm() statement performed during generation of PTX, you can add a "memory" clobbers specification after a 3rd colon. For example:

```
asm volatile ("mov.u32 %0, %%clock;" : "=r"(x) :: "memory");
asm ("st.u32 [%0], %1;" :: "l"(p), "r"(x) : "memory");
```

2.4. 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. Similarly, operand modifiers are not supported. For example, in

```
asm("mov.u32 %0, %n1;" : "=r"(n) : "r"(1));
```

the 'n' modifier in "%n1" is not supported and will be passed to ptxas, where it can cause undefined behavior. Refer to the document nvcc.pdf for further compiler related details.

Chapter 3. Error Checking

The following are some of the error checks that the compiler will do on inlinePTXasm:

- 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'

Chapter 4. Notices

4.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

4.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

4.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright