

Motivation

Previously, we discussed how modern GPUs are equipped with enormous amounts of arithmetic processing power. In fact, the computational advantage graphics processors have over CPUs helped precipitate the initial interest in using graphics processors for general-purpose computing. With hundreds of arithmetic units on the GPU, often the bottleneck is not the arithmetic throughput of the chip but rather the memory bandwidth of the chip. There are so many ALUs on graphics processors that sometimes we just can't keep the input coming to them fast enough to sustain such high rates of computation. **So, it is worth investigating means by which we can reduce the amount of memory traffic required for a given problem.**

What is Constant Memory?

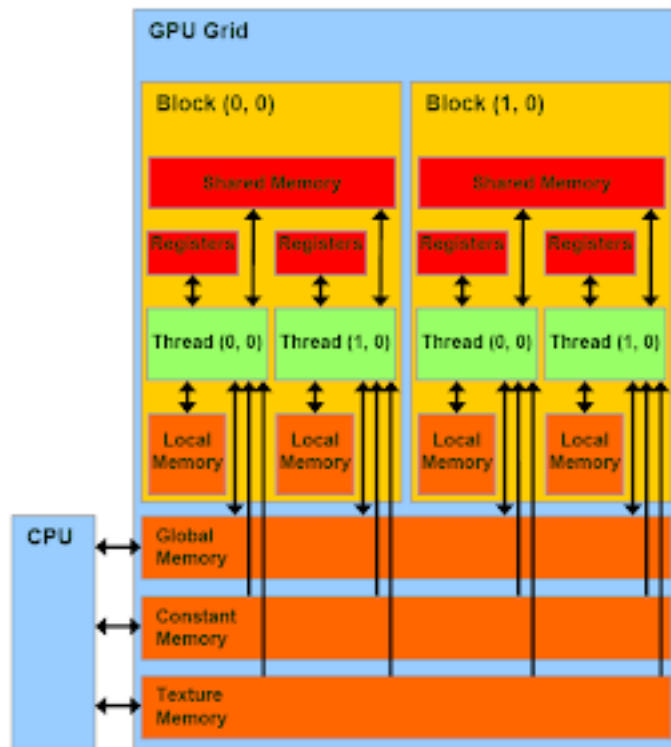
The CUDA language makes available another kind of memory known as ***constant memory***. As the name may indicate, **we use constant memory for data that will not change over the course of a kernel execution.**

Why Constant Memory?

NVIDIA hardware provides 64KB of constant memory that it treats differently than it treats standard global memory. **In some situations, using constant memory rather than global memory will reduce the required memory bandwidth.**

Where the Constant memory resides in GPU ?

Fig will shown you, where the constant memory resides in GPU



How does Constant memory speed up your CUDA code performance?

There is a total of 64 KB constant memory on a device. The constant memory space is cached. As a result, a read from constant memory costs one memory read from device memory only on a cache miss; otherwise, it just costs one read from the constant cache.

Advantage along with disadvantage

For all threads of a half warp, reading from the constant cache is as fast as reading from a register as long as all threads read the same address. Accesses to different addresses by threads within a half warp are serialized, so cost scales linearly with the number of different addresses read by all threads within a half warp.

The constant memory space resides in device memory and is cached in the constant cache mentioned in Sections F.3.1 and F.4.1 see *CUDA C Programming Guide* for details.

How does Constant memory works in CUDA?

Working of Constant memory is divided in Three steps which are as follows.

For devices of compute capability 1.x;

Step 1: A constant memory request for a warp is first split into two requests, one for each half-warp, that are issued independently.

Step 2: A request is then split into as many separate requests as there are different memory addresses in the initial request, decreasing throughput by a factor equal to the number of separate requests.

Final Step: The resulting requests are then serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise.

Alternatively, on devices of compute capability 2.x, programs use **the Load Uniform (LDU)** operation; see Section F.4.4 of the *CUDA C Programming Guide* for details.

How to use Constant memory in CUDA?

We talked about;

Where is constant memory?

- Data is stored in the device global memory
- Read data through multiprocessor constant cache
- 64KB constant memory and 8KB cache for each multiprocessor.

How about the Performance?

- Optimized when warp of threads read same location
- 4 bytes per cycle through broadcasting to warp of threads
- Serialized when warp of threads read in different locations
- Very slow when cache miss (read data from global memory)
- Access latency can range from one to hundreds clock cycles

Now we learn how to declare and use constant memory.

Declaration of constant memory

We declare constant memory using `__constant__` keyword. Constant memory always declared **in File scope (global variable)**. So, The mechanism for declaring memory constant is similar to the one we used of declaring a buffer as shared memory.

Example

```
__constant__ float cst_ptr [size];
```

Here we declare array of constant memory with name “**cst_ptr**” of size “**size**”.

Note:

It must be declared out of the main body and the kernel.

Way to use Constant memory

The instruction `cudaMemcpyToSymbol` must be used in order to copy the values to the kernel.
Syntax

```
cudaError_t cudaMemcpyToSymbol (const char * symbol, const void * src, size_t count
                                ,
                                size_t offset=0, enum cudaMemcpyKind )
```

Example:

```
//copy data from host to constant memory
cudaMemcpyToSymbol (cst_ptr, host_ptr, data_size );
```

Since we didn't explicitly write the `cudaMemcpyKind`, its default value is `cudaMemcpyHostToDevice`

Note:

The variables in constant memory are not necessary to be declared in the kernel invocation.

For example: “M” is the constant memory declared outside

```
__global__ void kernel (float *a, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N)
    {
        a[idx] = a[idx] * M;
    }
}
```

Complete Example of constant memory in cuda

Constant_memory_in_cuda.cu

```

//declare constant memory
__constant__ float cangle[360];

int main(int argc,char** argv)
{
    int size=3200;
    float* darray;
    float hangle[360];

    //allocate device memory
    cudaMalloc ((void**)&darray,sizeof(float)*size);

    //initialize allocated memory
    cudaMemset (darray,0,sizeof(float)*size);

    //initialize angle array on host
    for(int loop=0;loop<360;loop++)
        hangle[loop] = acos( -1.0f ) * loop / 180.0f;

    //copy host angle data to constant memory
    cudaMemcpyToSymbol ( cangle, hangle, sizeof(float)*360 );

    test_kernel <<< size/64 ,64 >>> (darray);

    //free device memory
    cudaFree(darray);
    return 0;
}

__global__ void test_kernel(float* darray)
{
    int index;

    //calculate each thread global index
    Index = blockIdx.x * blockDim.x + threadIdx.x;

    #pragma unroll 10
    for(int loop=0;loop<360;loop++)
        darray[index]= darray [index] + cangle [loop] ;
    return;
}

```

Difference between cudaMemcpy() and cudaMemcpyToSymbol()

`cudaMemcpyToSymbol()` is the special version of `cudaMemcpy()` when we copy from host memory to constant memory on the GPU. The only differences between `cudaMemcpyToSymbol()` and `cudaMemcpy()` using `cudaMemcpyHostToDevice` are that `cudaMemcpyToSymbol()` copies to constant memory and `cudaMemcpy()` copies to global memory.

Where to use and should not use Constant memory in CUDA?

As per as I guide;

Use Constant memory in the following cases

- When you know, your input data will not change during the execution
- When you know, your all thread will access data from same part of memory
As shown in above example; every thread in a block access same address space pointed out by “**cangle**” array.

Should not use Constant memory in the following cases

- When you know, your input data will be change during the execution
- When you know, your all thread will not access data from same part of memory.
- When your data is not read only. For example “output” memory space should not be constant.

In essence, you should not use constant memory when every thread in a block don't access same address space. For example; you have an array of 3,000 elements and you breaks this element to lunch sufficient number of threads in a block. So, each thread in a block will access different element of an array as counter part of the above example where each thread reads same data controlled by “**loop**” variable, therefore each thread will access same data.

Performance consideration of constant memory

Declaring memory as `__constant__` constrains our usage to be read-only. In taking on this constraint, we expect to get something in return. As I previously mentioned, reading from constant memory can conserve memory bandwidth when compared to reading the same data from global memory. **There are two reasons why reading from the 64KB of constant memory can save bandwidth over standard reads of global memory.**

A single read from constant memory can be broadcast to other “**nearby**” threads, effectively saving up to **15 reads**.

Constant memory is cached, so consecutive reads of the same address will not incur any additional memory traffic.

What do we mean by the word *nearby*?

To answer this question, we will need to explain the **concept of a warp**. For those readers who are more familiar with *Star Trek* than with weaving, a warp in this context has nothing to do with the speed of travel through space. In the world of weaving, a warp refers to the group of *threads* being woven together into fabric. **In the CUDA Architecture, a warp refers to a collection of 32 threads that are “woven together” and get executed in lockstep. At every line in your program, each thread in a warp executes the same instruction on different data [More? follow [this link](#)].**

When it comes to handling constant memory, **NVIDIA hardware can broadcast a single memory read to each half-warp**. A half-warp—not nearly as creatively named as a warp—is a group of 16 threads: half of a 32-thread warp. **If every thread in a half-warp requests data from the same address in constant memory, your GPU will generate only a single read request and subsequently broadcast the data to every thread. If you are reading a lot of data from constant memory, you will generate only 1/16 (roughly 6 percent) of the memory traffic as you would when using global memory.** But the savings don’t stop at a **94 percent reduction in bandwidth when reading constant memory! Because we have committed to leaving the memory unchanged, the hardware can aggressively cache the constant data on the GPU.**

So after the first read from an address in constant memory, other half-warps requesting the same address, and therefore hitting the constant cache, will generate no additional memory traffic.

After caching the data, every other thread avoids generating memory traffic as a result of one of the two constant memory benefits:

It receives the data in a half-warp broadcast.

It retrieves the data from the constant memory cache.

Unfortunately, there can potentially be a downside to performance when using constant memory. **The half-warp broadcast feature is in actuality a double-edged sword.** Although it can dramatically accelerate performance when all 16 threads are reading the same address, it actually slows performance to a crawl when all 16 threads read different addresses.

The Final words on Performance consideration of constant memory

The trade-off to allowing the broadcast of a single read to 16 threads is that the 16 threads are allowed to place only a single read request at a time. For example, if all 16 threads in a half-warp need different data from constant memory, the 16 different reads get serialized, effectively taking 16 times the amount of time to place the request. If they were reading from conventional global memory, the request could be issued at the same time. In this case, reading from constant memory would probably be slower than using global memory.

Summary of the Article

In this article we read about constant memory in context of CUDA programming. We started talking about Why (What is) constant memory and how to declare & use constant memory in CUDA and end our discussion with Performance consideration of constant memory in CUDA.