

Nvidia Cuda

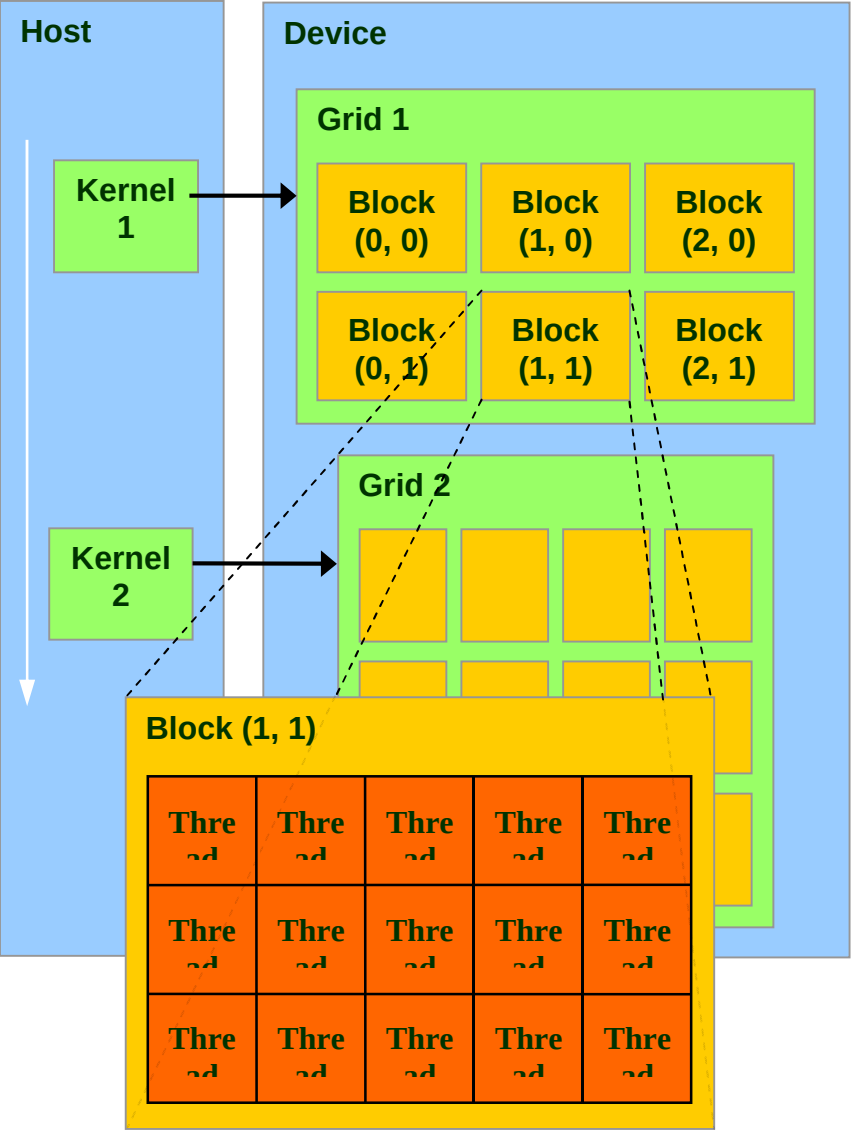
A Short Introduction

GPGPU Background Information

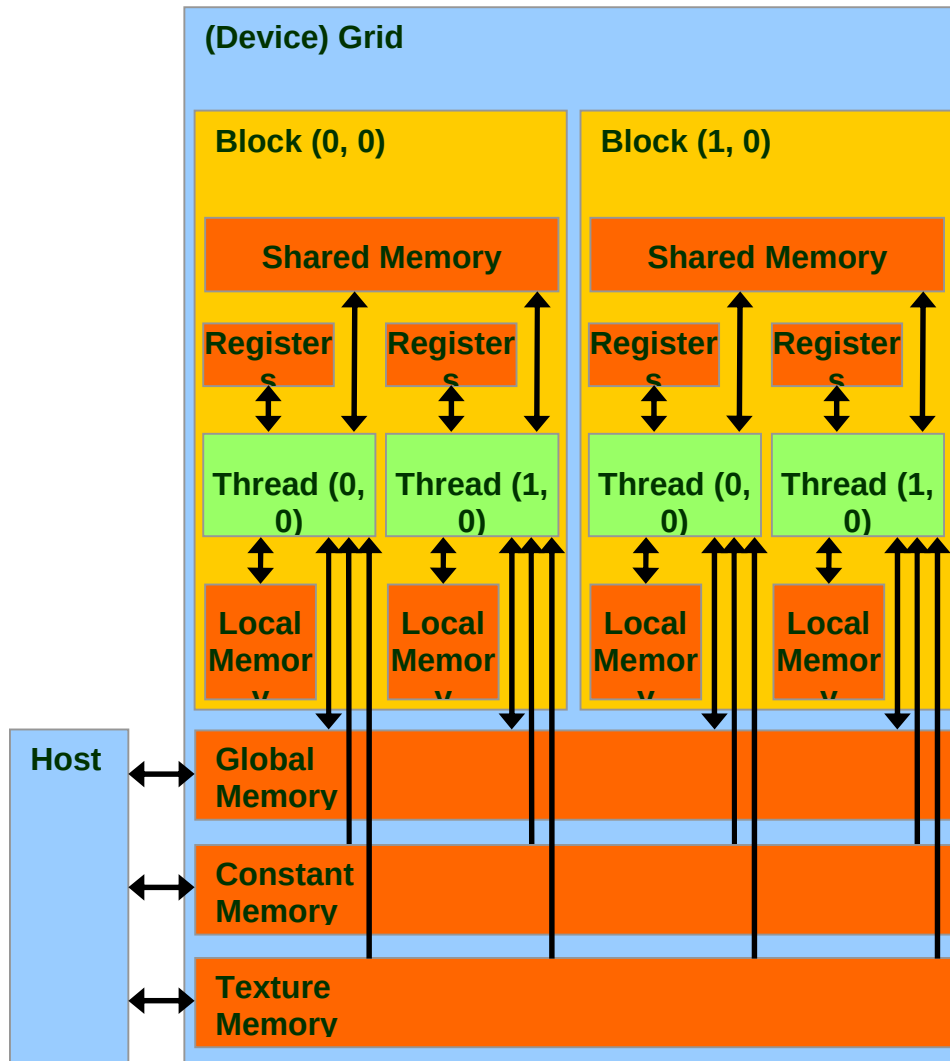
GPUs are traditionally used in graphics rendering for personal computers, but due to their massively parallel, multi-core architecture, are also suited to stream processing. Stream processing involves doing the same computation on a number of elements or vertices. This obviously limits the number of applications in which the GPU would actually be helpful, but in some instances such as Fourier Transforms and weather modeling, a decrease in computation time may be noted. APIs such as OpenGL and DirectX allowed programmers to tap into the computing power of the GPU but Nvidia's CUDA really brought the GPU's capabilities onto the market. CUDA is essentially a C language extension that provides the programmer with the low-level, deterministic and high-level API. In other words, CUDA lets any level of programmer harness the power of the GPU.

GPU Architecture:

- Maximum of 512 threads per block. Ideally blocks are 16x16 threads or larger in order to take advantage of the GPU's resources. Maximum dimensions(512 X 512 X 64) (3D)
- Warp size – 32 threads
- Grid size(max) – 65, 535 X 65, 535 X 1 (2D)
- During execution of a block(it will execute on a single multiprocessor) the block is split into warps which then are executed in a SIMD fashion and are time-sliced.



Memory Model:



Memory Details

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

- Register – single cycle
- Shared Memory –single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1...10s...100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1...10s...100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

Cuda Language Extensions:

Function Qualifiers

`__global__` : invoked from within host (CPU) code, cannot be called from device (GPU) code , executes on device

`__device__` : called from other GPU functions, cannot be called from host (CPU) code, executes on device

`__host__` : can only be executed by CPU, called from host

`__host__` and `__device__` qualifiers can be combined

Sample use: overloading operators

Compiler will generate both CPU and GPU code

Restrictions on these qualifiers are described in the Nvidia Programming Guide –see address below.

Variable Qualifiers (GPU code)

`__device__`

Stored in device memory (large, high latency, no cache)

Allocated with `cudaMalloc` (`__device__` qualifier implied)

Accessible by all threads

Lifetime: application

`__shared__` (hundreds of times faster than global memory)

Stored in on-chip shared memory (very low latency)

Allocated by execution configuration or at compile time

Accessible by all threads in the same thread block

Lifetime: kernel execution

Unqualified variables:

Scalars and built-in vector types are stored in registers

Arrays of more than 4 elements stored in device memory

Built in

All `__global__` and `__device__` functions have access to these automatically defined variables
`dim3 gridDim`; Dimensions of the grid in blocks (at most 2D)
`dim3 blockDim`; Dimensions of the block in threads
`dim3 blockIdx`; Block index within the grid
`dim3 threadIdx`; Thread index within the block

Kernels

Kernels are C functions with some restrictions
Can only access GPU memory
Must have void return type
No variable number of arguments (“varargs”)
Not recursive
No static variables

Launching kernels

See programming guide 4.2.3 for details.

Modified C function call syntax:

`kernel<<<dim3 grid, dim3 block, N, S>>>(...)`

`grid`: grid dimension

`block`: block dimension

`N`: number of bytes of dynamically allocated shared memory in addition to the statically allocated memory. Optional argument.

`S`: a type of cuda stream. Optional argument.

Using a CUDA Template

This may not be the most efficient way of using a template but it works.

1. The sample projects provided by Nvidia can be found in `>C:\Program Files (x86)\NVIDIA Corporation\NVIDIA CUDA SDK\projects`.
2. In this case we will start with the project ‘Template’
3. Make a copy of the folder and rename it.
4. In the folder rename the `.cu` files, the `.sln` file and the `.cpp` file by changing ‘template’ to your chosen name.
5. Open the `.sln` file. You will find that it still opens the template project.
6. Right click on the project and remove it from the solution and save.
7. Rename the `.vcproj` file in your working directory by changing the ‘template’ as before.
8. Now open the `.vcproj` file with WordPad and select Edit > Replace.
9. Replace all instances of ‘template’ with your chosen name, save and exit.

10. Now in Visual Studio right click on the solution on the right and select Add Existing Project.
11. Find the .vcproj file that you just altered and add it.
12. All your files should now be properly named and linked, but check to ensure that a build works. Don't forget to change win32 to x64.
13. In each of these linked files find and replace all instances of 'template' with your chosen name.
14. If the build works then you are ready to make any changes you wish!

Cuda and Fortran

- To call a Cuda function from FORTRAN, use the standard 'call function' syntax.
- Ensure that all the files are placed where they can be found, or are linked to during the compilation.
- The Cuda function must be named FUNCTION because FORTRAN works in caps.
- The variables passed from FORTRAN to Cuda are pointers. Therefore some problems may arise from row-wise memory allocation in C vs. column-wise memory allocation in FORTRAN. Also such things as f(g) where f is an array of some type and g is an array of indices poses a problem because the elements that should be passed are not in one solid memory block.

The Cuda Fortran Interface Compilation:

In order to compile the sample interface supplied to us by Nvidia(or any other Cuda-Fortran interface) here are the steps:

- Compile from the command line. Make sure you are in your working directory.
- First compile the Cuda code using a call like ' nvcc -c -O3 Cuda_function.cu'
- This creates a Cuda_function.obj file
- Then compile the Fortran code and link to the .obj file
- The call would be ' ifort -o program fortran_main.f90 Cuda_function.obj '
- Often the compiler won't be able to find certain libraries that it needs and that is simply solved by giving it the proper path in the above call.

A Note on Fortran:

The ifort compiler still does not recognize .f95 files. A way around that is simply to rename them .f90 .

ECE 498 AL1 : Programming Massively Parallel Processors

Very useful power-point presentations as well as lectures:

<http://courses.ece.uiuc.edu/ece498/al1/>

<http://www.nvidia.com/docs/IO/47904/VolumeI.pdf>

http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf