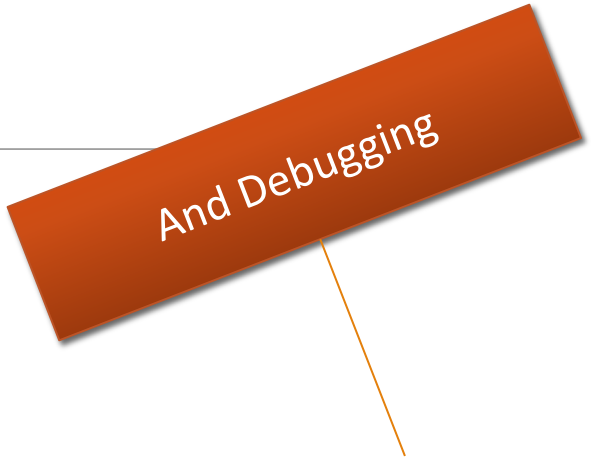
CUDA Programming on the Tegra Xavier

-BY KRISTOFFER ROBIN STOKKE

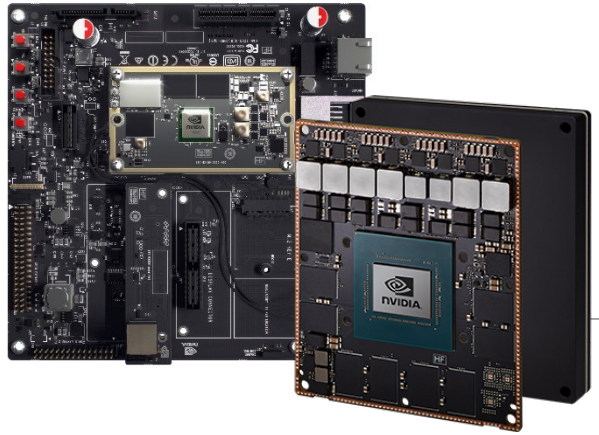


Keep This Under Your Pillow



- Volta Tuning Guide
  - https://docs.nvidia.com/cuda/archive/10.2/volta-tuning-guide/index.html
- CUDA Programmer's Guide
  - https://docs.nvidia.com/cuda/archive/10.2/cuda-c-programming-guide/index.html
- CUDA for Tegra
  - https://docs.nvidia.com/cuda/archive/10.2/cuda-for-tegra-appnote/index.html
- Tegra X1 Whitepaper
  - https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/
- Last but not least
  - CUDA-GDB
    - You will need it
  - NVPROF – if you care about performance
    - https://docs.nvidia.com/cuda/archive/10.2/profiler-users-guide/index.html

Tegra Xavier vs. Tegra X1



	Tegra Xavier		Tegra X1	
High Performance CPU	8 x Carmel	2 MB L2 + 128 kB $I, 64 kB $D	4 x Cortex-A57	2 MB L2 + 48 kB $I, 32 kB $D
Low Power CPU	None		4 x Cortex-A53	512 kB L2 + 32 kB $I/$D
Architecture	Volta		Maxwell	
Cores	512 Cores (8 Volta SM)		256 Cores (2 SMM, 128 Cores / SMM)	
Memory bitwidth	256-bit		64-bit	
L2 cache	512 kB		256 kB	
L1 cache	128 kB (shared memory + L1 RO cache)		• 64 kB (unique shared memory) • 64 kB read-only cache	

GPU Compute Capability

Compute Capability	Generation
1.x	Tesla
2.x	Fermi
3.x	Kepler (Tegra K1)
4.x	???
5.x	Maxwell (Tegra X1)
6.x	Pascal (Tegra X2)
7.x (**7.2**)	Volta (**Tegra Xavier**)
7.5	Turing
8.x	Ampere



You are here

- «The functional development of GPUs»
- For Maxwell:
  - Half-precision (16-bit) floating point
  - Dynamic parallelism
    - Kernels can launch kernels
- Newest CC
  - Tensor Core (neural network support)
- CUDA Toolkits
  - Compiler, doc, examples etc
  - nvcc –version
  - Already installed for you
- If interested:
  - https://en.wikipedia.org/wiki/CUDA

Parallelism in GPUs

- Massive.
- Volta SM – **Volta** Streaming Multiprocessor
  - Four Warp Schedulers (WS)
  - 4 x 16 CUDA cores
  - 4 x 8 LD/ST units, ~16k 32-bit registers
  - 4 x 4 Special Function Units (SFUs)
  - 8 Tensor Cores
  - 128 kB shared memory*
- At every clock cycle..
  - Each WS selects an eligible **warp..**
  - .. and dispatches two instructions
- All threads *should* follow, more or less, the same execution path

Group of 32 threads



Volta GPU Memories

- A *flexible*, multi-layered cache hierarchy
  - Improves memory bandwidth
  - WS selects ready (non-stalling warps)
  - Highly programmable

- 256-bit RAM interface
- 512kB, GPU-global L2 cache
  - Shared between all Volta SMs
- 128 kB, read-only L1 cache / shared memory
  - Shared texture / local l1 cache
  - Local to the SM
- SM-local L1 cache
  - Directly addressable through *shared memory*
  - Local to the SM
- Registers



GPU Memories (Continued)

- The CUDA toolkit documentation introduces the following memory spaces and naming conventions..
- Global memory loads: Loads from RAM, possibly through caches
- «Local memory»: Register spills, code, and other
  - Resides in **RAM** or «somewhere» in the cache hierarchy, hopefully in the right place
- «Shared memory»: RW L1 cache shared in a **thread block**
- «L1 RO cache»: Cache global, read-only memory loads

CUDA Programmer's Perspective

- Schedule **blocks** of threads (**execution configuration syntax**)
- WS schedules eligible 32-thread groups of blocks

```
__global__ void memcpy( uint32_t * src, uint32_t * dst) {
        ...
}

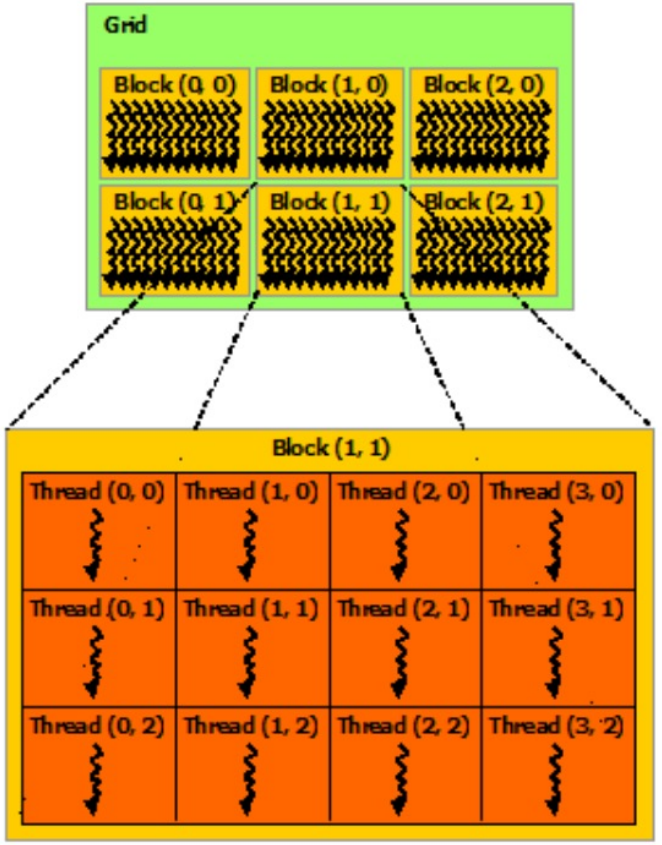
void main(void){
        dim3 block_dim(1024, 1, 1);
        dim3 grid_dim(1024, 1, 1);
        uint32_t *src, *dst;

        memcpy<<<grid_dim, block_dim, 0, 0>>>(src, dst);

        ...
}
```

Shared memory per block

Cuda stream (default 0)



From CUDA Programmer's guide.

CUDA Programmer's Perspective (Cont.)

- Special purpose registers
  - threadIdx.[x/y/z] -> **block index coords**
  - blockIdx.[x/y/z] -> **grid index coords**
  - blockDim.[x/y/z] -> **grid dimension sizes**
- In example:
  - blockDim.x = 1024, blockIdx.x \in [0, 1023]
  - Index into contiguous memory

```
__global__ void memcpy( uint32_t * src, uint32_t * dst) {

        int idx;

        idx = threadIdx.x + ( blockIdx.x * blockDim.x );

        dst[idx] = src[idx];

        return;
}
```

Synchronisation

- **ECS** kernel_symbol_name<<< gridDim, blkDim, shared, **stream** >>> (__VA_ARGS__)
- Kernel launches are **always asynchronous**
  - Executing thread immediately returns
- «Worst» sync: **cudaDeviceSynchronise()**
  - **Blocks** until *all pending GPU activity is done*
  - However good for debugging / testing purposes
- **Streams**
  - Streams created with **cudaStreamCreate()** -> + flags!
  - Run kernel launches and asynchronous memory copies in streams
  - Sync on streams with **cudaStreamSynchronize(stream)**

Other API Specific Details

- Two APIS
  - Driver API
  - Runtime API <- **use this** (https://docs.nvidia.com/cuda/archive/10.2/cuda-runtime-api/index.html)
- Other modules you should have a look at
  - Device management
  - Error handling
  - Memory management, unified addressing
- CUDA samples: deviceQuery
- CUDA Compiler: nvcc
  - Source files with CUDA code (***.cu**) are compiled as **.cpp** files
  - **nvcc** extracts CUDA code, passes rest to native c++ compiler

When Things Aren’t Going Your Way

- **Cuda-gdb**
  - Just like gdb
- Main advantage: captures error conditions for you
  - But this doesn’t mean you can get lazy
- **Always check error codes** and **break on anything != cudaSuccess**
  - Make a macro

GPU Performance Analysis

- CUPTI: GPU Hardware Performance Counters (HPCs)
- **Usage: nvprof –e <event counters> -m <metrics> <binary> <arguments>**
- Summary modes, counter collection modes....
  - Tells you about resource usage – time, memory, floating point performance, **elapsed cycles**
  - Takes time to profile – be patient or use **./c63enc –f 5** <- make sure to trigger ME & MC
- Check HPC availability with **nvprof –query-events –query-metrics**
  - Notice there are well above 100 HPCs to choose from..
  - ...which ones matter?
  - I will tell you! ☺ ☺ ☺

GPU Performance Analysis (Continued)

- Memory usage
  - **L1_global_load_hit, l1_local_{store/load}_hit, l1_shared_{store/load}_transactions, shared_efficiency**
- Instructions
  - **Inst_integer, inst_bit_convert, inst_control, inst_misc, inst_fp_{16/32/64}**
- Causes of stalling
  - Memory, instruction dependencies, sync...
- Other
  - **Elapsed_cycles_sm**
- These are for the TK1, but should be at least similar for TX1
- **Don't get confused** by HPCs such as **{gld/gst}_throughput**

Code Examples