\chapter{GPU With CUDA Architecture Updates}

\label{sec:gpu\_architecture}

GPU is specialized computer that provides fast image processing and construction for display. However, over a decade ago NVidia started to adjust the architecture for general applications that can be executed in parallel. The NVidia GPU consists of multiple clusters of core arrays, with each core array sharing cache memory and registers for execution.

This architecture allows us to run programs in the mode ``same instruction multiple data’’ (SMID), where, for each clock cycle, all active cores run the same operation but on different parts of the memory.

NVidia architecture may be divided to three types:

\begin{enumerate} \label{memorytypes}

\item Global memory is accessible for all streaming multiprocessors (clusters of core arrays), which make access relatively slow, with hundreds of cycles per read or write. However, to conceal the delay, execution control can switch between active cores while memory is loaded.

\item Cache is accessible separately for each multiprocessor, which is order of magnitude faster for access but is much slower than the global memory. It is advisable to use this for synchronizing data between threads of the same block or for overflow of data that cannot be stored in the registers.

\item Register files are very limited amounts of memory close to the processors. They are accessible in single clock cycles, and all instructions executions can be done there.

\item Constant memory is a small amount of memory separately accessible to each core array. Although this is read-only memory, when executing code from the device, the host can upload data before kernel execution. This memory can be rapidly accessed if all threads in warp access the same cell at the same clock cycle; otherwise, the memory is serialized, which slows the read operation because of the number of different cells that need to be accessed.

\end{enumerate}

The Compute Unified Device Architecture (CUDA) has two contexts of execution. The host is the CPU and all CUDA programs start with host execution. It controls all memory read to the GPU and from the GPU and launches the device (GPU) context procedures, which are called kernels. Each kernel has a two-level hierarchy:

\begin{enumerate}

\item Blocks: Each block runs on single multiprocessors and have access to the same cache memory. Each multiprocessor can run multiple blocks but the order of execution is not guaranteed. Although multiple blocks can communicate through the atomic actions, it is slow and mostly unadvised, because the core array must stop and switch execution to another block or stop completely if all blocks are waiting.

\item Threads: Each block can contain up to 1024 threads, which execute concurrently (32 of them each cycle). Threads can access the same shared memory array with few limitations. When threads have to read memory that needs multiple clock cycles to be accessed, the multiprocessor switches context and makes these threads dormant, which is a fast operation.

\end{enumerate}

All threads on each block have access to the same shared memory array (SHM), which is defined in the cache by the programmer. SHM access is divided into banks of either four bytes or, on some cards, eight bytes. Each thread can access only a single bank. If $n$ threads read or write to $n$ distinct banks, all can be serviced simultaneously, which multiplies the memory bandwidth by $n$. However, if those threads read or write to the same bank, a conflict occurs, and the chip will split the memory access into $n$ conflict-free serialized requests. SHM is placed in L1 cache memory on chip and is much faster to deliver requests (one clock cycle) than either the local or global memory, but an $n$-way bank conflict will reduce access speed by a factor $n$ and should be avoided. If multiple threads of the same warp read a single bank, multicasting occurs and all threads get memory at the same time. For multiple writes to the same bank, only a single, arbitrarily selected, thread writes.

\phantomsection

\section{Difference Between Generations of NVidia architecture}

\label{sec:gpu\_architecture\_generations}

\phantomsection

\paragraph{SM basic properties}

\label{par:sm-basic-properties}

The streaming multiprocessor (SM) is the center of the GPU and controls execution of the kernel. Over the years, NVidia has changed components included in the SM.

Each compute capability is identified with a SM model. This project is optimized and run on several capabilities.

% fig:nvidia-architectures-capabilities

\paragraph{Compute Capability 1.x, Tesla Architecture}

\label{par:cuda-tesla-architecture}

The NVidia architecture is divided into generations of processors (identified by the number before decimal point, major version). The first-generation architecture 1.x \cite{NVIDIA\_Programming\_guide42} had a maximum capacity of 768 threads per SM, with only 8 FP32 lanes and 2 SFUs for a single-precision unit and a double-precision unit. Because threads executed in groups of 32, called warps, each multiply or add command took four clock cycles to complete, and intrinsic functions such as $fdividef$ or $logf$ took 16 cycles to complete. Our project requires 256 parallel threads per block to represent spatial partitions \cite{Saboddoron20131215}, which limits us to maximum of three blocks on 1.0--1.1 and four blocks on 1.2--1.3. Another critical resource is the 32 bit registers per SM, with each having only 8--16 K. For a maximum capacity each thread would have to use 10 to 16 registers. We can thus expect at most a single block for 1.0--1.1 or two blocks for 1.2--1.3 (32 registers per thread), which significantly limits the parallelism of 1.x SMs relative to later generations. Shared memory is also a limiting factor; the architecture 1.x had 16 banks per warp, so reading shared memory took 2 cycles per warp, and bank conflicts had to be avoided between two halves of the warp. This model has been deprecated and is not supported by CUDA.

\paragraph{Compute Capability 2.x, Fermi Architecture}

\label{par:cuda-fermi-architecture}

The NVidia second-generation architecture was significantly modified relative to the first-generation architecture. Comparing \cref{tab:technical-comparison,tab:architectures-comparison} shows that SM 2.0 doubled the amount of registers per SM to 32 K and increased warps per SM to 48, thereby allowing more active threads. Only four \ac{sfu} are available for every 32 FP32 lanes, meaning that intrinsic operations take eight cycles, and single-precision multiply, add, and multiply-add take one clock cycle to execute. With four times the number of FP32 lanes and twice the number of registers, register pressure increased, which reduced kernel occupancy because too few registers were available for the maximum number of threads. For example, SM 2.0 can run 1536 threads simultaneously. If each requires 32 registers, maximum occupancy will be achieved when using 48 K, but SM 2.0 has only 32 K so maximum occupancy cannot rise above 66\%. The second-generation architecture also doubled the number of warp schedulers, while the occupancy defined the amount of warps held in SM registers at any given time. Each scheduler executed warp if sufficient FP32 lanes were available, which happens when warp needs to read from memory and is stalled. SM 2.1 increased the number of FP32 lanes to 48 and the number of \ac{sfu} units to eight, allowing also for two instructions to execute per warp, if they were independent. Although this configuration increased register pressure, it constitutes an overall improvement for arithmetic-intense programs and for intrinsic operations. This model is deprecated since CUDA 8 and is not supported.

\paragraph{Compute Capability 3.x, Kepler Architecture}

\label{par:cuda-kepler-architecture}

The third generation of CUDA-supported architecture (SM 3.x, called SMX) is shown on the left side of \cref{fig:nvidia-kepler-maxwell-comparison}. The FP32 lanes were increased to 192 (light green boxes), and the \ac{sfu} to 32 (forest-green boxes), doubling the number of warp schedulers to four (orange boxes on top), which improves the ratio of FP32 lanes per scheduler allowing more-arithmetic-intense programs. The maximum number of threads per SM was increased to 2048 and the number of registers was doubled to 64 K (dark-blue box under brown boxes). The register pressure was reduced, permitting 32 per thread even for maximum occupancy. The number of double-precision units per SM is only six (yellow boxes). Note that the figure shows the Quadro variant of the core described in \cref{par:cuda-quadro-tesla-geforce}. We see 64 of them, but Geforce has only eight per SMX. This discourages 64-bit operations; however, the two instructions per scheduler (brown boxes are dispatchers) were expanded to execute 64-bit instruction with 32-bit instructions if independent Kepler schedulers are assigned. Each SMX can read or write 128 bytes (olive boxes can read 4 bytes each). The 3.5 architecture added dynamic parallelism \cite{NVIDIA\_Programming\_guide75}, which allows kernel launching from kernels. This improves speed by avoiding copying kernel data and instructions from the host to the device for run-time decisions. The launched kernel, called a child, is guaranteed to execute and return before parent completion. The Kepler cache memory divided into L1 allocated per SMX and an L2 cache shared for all SMXs. L1 has 64 KB (cyan box near the bottom) and can be configured as 48-16, 32-32, or 16-48 KB between L1 (used for register spilling) and shared memory. The allocation per block must be reduced, or CUDA will reduce occupancy two ensure that each remain block has sufficient shared memory.

\phantomsection

\paragraph{Compute Capability 5.x and 6.1-6.2, Maxwell and Pascal Architecture}

\label{par:cuda-maxwell-pascal-architecture}

The Maxwell SM architecture (CC 5.x), designated SMM, was restructured to improve energy efficiency, \cite{NVIDIA\_Maxwell\_White\_Paper} and is shown on the right side of \cref{fig:nvidia-kepler-maxwell-comparison}. The colored boxes function identical to those in the Kepler SMX chart. The SMM is divided into four warp schedulers, each of which can dispatch one instruction per cycle (\cref{tab:architectures-comparison}). Each scheduler is physically connected to its 32 FP32 lanes, 8 SFUs, and single DP unit. This doubles the energy efficiency and also doubles the numbers of SMs while decreasing energy consumption and increasing die size from 294 to 398~mm$^2$. Note that, while the number of instructions per cycle executed was reduced from 2 to 1, the number of arithmetic logic units per SM also decreased, so more SM per card can be set (on comparable tiers of cards). With both architectures limited to 2048 threads per SM and 64 K registers per SM, the Maxwell architecture can run double the number of threads on a single card. Note, however, that the count of DP units per FP32 lanes lowered from 1/24 in Kepler to 1/32 (this is relevant for Geforce, \Cnameref{par:cuda-quadro-tesla-geforce}), making the use of double precision inadvisable. The Maxwell architecture also introduced a separation between the L1 cache (24 KB) and shared memory (64 KB in 5.0, 96 K from 5.2, \cref{tab:technical-comparison}). In contrast, the Kepler architecture unified 64 KB that could be configured between the shared and L1 caches. Pascal, CC 6.1-6.2, with a designated SMP, has mostly the same features as the CC 5.2, with an increase in the L1 cache from 24 to 48 KB. This allows a better ratio of cache hits to misses from global memory. The register spills are stored in the L2 cache in the Maxwell architecture, not the L1 cache as in the Kepler architecture, and this need to be considered when limiting number of registers per thread.

\begin{figure}[ht]

\setlength{\subfigcapmargin}{.1in}

\centering

\includegraphics[width=0.9\textwidth,keepaspectratio=true]{figs/nvidia-kepler-vs-maxwell-sm}

\caption{Comparison of Kepler (CC 3.x) and Maxwell (5.x) SM based on figures from \cite{NVIDIA\_Kepler\_White\_Paper} and \cite{NVIDIA\_Maxwell\_White\_Paper}. }

\label{fig:nvidia-kepler-maxwell-comparison}

\end{figure}

\paragraph{Quadro and Tesla series Differences from GeForce}

\label{par:cuda-quadro-tesla-geforce}

NVidia manufactured several variations for each CC version; these versions of SMs differ by the number of DP units on the SM. The numbers in the previous paragraphs described the GeForce model, which is available for this project. For Quadro and Tesla versions there are two FP32 lanes per DP unit in the Fermi and Maxwell architectures and three in the Kepler architecture (available in CC 3.7). The number is also two on Pascal 6.0; however, CC 6.0 has SM with 64 FP32 lanes and two warp schedulers, with double the number of SMs per FP32 lane and a limitation of 2048 thread per SM. Thus, CC 6.0 can run double the number of threads per lane if the register count is less than 16 per thread.

\section{NVidia Nsight}

\label{sec:nvidia-nsight}

The NVidia Nsight tool allows for both CUDA debugging and performance analysis \cite{NSight54}. Because our project aim is to more rapidly solve the cochlear equations, Nsight enables us to test multiple aspects of card workload and thus to find bottlenecks. We present here the tests used to optimize performance in this project.

\begin{enumerate}

\item[\namedlabel{itm:achieved-occupancy}{Achieved Occupancy}] describes the actual number of warps that are executed on the SM for the current kernel. The Kepler architecture can run 64 warps on each SM in granularity of blocks. If the kernel needs between 49 and 64 registers per thread and the block contains 256 threads, no more than 1024 will be able to run (4 blocks) per SM and Achieved Occupancy will be $\frac{1024}{2048}=50\%$. It is advisable to use a number of blocks that is larger than Achieved Occupancy times the number of SMs. We also divide it to prevent a small number of final blocks from being processed while most of the SM sits idle. This is called the tail-end effect.

\item[\namedlabel{itm:achieved-flops}{Achieved FLOPs}] describes the number of floating point operations per second executed on the device. It is a common method to evaluate performance and can be assembled from several independent sequential tasks that run on multi-core devices or any device with shared memory. It can also be adjusted for use on cores with different operation frequencies \cite{ISAT2017PART2}. Thus, this metric is useful for optimizing the time-to-solution process.

\item[\namedlabel{itm:achieved-iops}{Achieved IOPs}] indicates the number of integer operations per second executed on the card. Although similar to \ref{itm:achieved-flops}, this method is useful for differentiating between solution performance but does not permit one to derive the cause of bottlenecks.

\item[\namedlabel{itm:instructions-statistics}{Instruction Statistics}] is the basic measurement of the global work distribution scheduler and has several indicators. The instructions per cycle (IPC) is the measured average for warps across the SM. The IPC gives the number of dispatched instructions for the warp and is ideally identical to the executed IPC. However, some instructions have to be issued multiple times. Instructions per warp (IPW) gives the average number of executed IPWs or cycles. If the number of warps launched per SM varies greatly, the most likely cause is insufficient warps to get \ref{itm:achieved-occupancy} across all SMs, which is called the ``tail effect’’ and reduces the average time to solution across the device due to idle warp schedulers that wait for active warps to finish the kernel.

\item[\namedlabel{itm:issue-efficiency}{Issue Efficiency}] measures the ability of the device to allocate instructions for execution per cycle and is composed of several indicators: Active warps measured per SM and average number of allocated warps for the SM across the kernel. The issue efficiency cannot be higher than \ref{itm:achieved-occupancy} but, if significantly lower, there are not enough warps to exploit device parallelism. If SM vary greatly, some of SMs finish before others and sit idle, so the workload should be redistributed. The eligible warps are the average number of warps that can execute instruction each cycle, and this number cannot exceed the number of warp schedulers per SM. The warp issue efficiency is the percent of time that the scheduler can issue instructions for one of its warps. If too low, the graph of issue stall reasons indicates division of failures for schedulers to allocate instructions, such as memory dependency, or waiting for load or store units to be available to read or write to memory. Execution dependency and data for executed instruction are not yet available.

\item[\namedlabel{itm:memory-statistics}{Memory Statistics}] measures the number of requests and bytes across all memory modules, including shared, constant, local, and global memory as well as the L1 and L2 caches. This can indicate bottlenecks caused by stalls issues due to memory calls.

\item[\namedlabel{itm:source-level-experiments}{Source-Level Experiments}] offer several indicators to resolve single instructions, including the total number of times executed in the kernel, and the percent of threads in a warp that executed the instruction.

\end{enumerate}

\paragraph{Occupancy Calculator}

\label{sec:occupancy-calculator}

This NVidia tool consists of an Excel sheet that takes as input the CC, threads per block for kernel, registers per thread, and SHM and calculates the maximum number of blocks per SM. This tool gives the number of target registers to increase occupancy.