Nvidia K20 GPU: Accelerated Computing

Overview

CPU is optimized for the low latency access to cached data sets while the GPU is optimized for the data parallel throughput computation and has architecture tolerant of memory latency. CPU architecture must minimize latency within each thread. GPU architecture hides latency with computation from other thread wraps.

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Tesla K20X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic SKU reference</td>
<td>699-22081-0200-xxx</td>
</tr>
<tr>
<td>Chip</td>
<td>GK110</td>
</tr>
<tr>
<td>Package size GPU</td>
<td>45 mm x 45 mm 2397-pin S-FBGA</td>
</tr>
<tr>
<td>Processor clock</td>
<td>732 MHz</td>
</tr>
<tr>
<td>Memory clock</td>
<td>2.6 GHz</td>
</tr>
<tr>
<td>Memory size</td>
<td>6 GB</td>
</tr>
<tr>
<td>Memory I/O</td>
<td>384-bit GDDR5</td>
</tr>
<tr>
<td>Memory configuration</td>
<td>24 pieces of 64M x16 GDDR5 SDRAM</td>
</tr>
<tr>
<td>Display connectors</td>
<td>None</td>
</tr>
<tr>
<td>Power connectors</td>
<td>8-pin PCI Express power connector</td>
</tr>
<tr>
<td>Board power</td>
<td>235 W</td>
</tr>
<tr>
<td>Idle power</td>
<td>25 W</td>
</tr>
<tr>
<td>Thermal cooling solution</td>
<td>Passive heat sink</td>
</tr>
<tr>
<td>Mean time between failures (MTBF)</td>
<td>Uncontrolled environment: 128440 hours at 35 °C</td>
</tr>
<tr>
<td></td>
<td>Controlled environment: 208861 hours at 35 °C</td>
</tr>
</tbody>
</table>

GPU Architecture has two main components:

- **Global Memory:**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - 6GB per GPU
- **Streaming Multiprocessors (SMs):**
  - Perform the actual computations
  - 14
  - Each SM has its own: control units, registers, execution pipelines, caches
ams@login5:~> xtprocadmin --attrsgpu | grep "GPU"

<table>
<thead>
<tr>
<th>NID (HEX)</th>
<th>NODENAME</th>
<th>TYPE</th>
<th>GPUNUM</th>
<th>GPUMEM</th>
<th>GPUMHZ</th>
<th>GPUCORES</th>
<th>GPUSUBTYPE</th>
<th>GTYPE</th>
<th>GTHREADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>462</td>
<td>c7-0c2s7n0</td>
<td>compute</td>
<td>0</td>
<td>6144</td>
<td>732</td>
<td>14</td>
<td>Tesla_K20X</td>
<td>nVidia_Kepler</td>
<td>GPU</td>
</tr>
<tr>
<td>463</td>
<td>c7-0c2s7n1</td>
<td>compute</td>
<td>0</td>
<td>6144</td>
<td>732</td>
<td>14</td>
<td>Tesla_K20X</td>
<td>nVidia_Kepler</td>
<td>GPU</td>
</tr>
<tr>
<td>464</td>
<td>c7-0c2s7n2</td>
<td>compute</td>
<td>0</td>
<td>6144</td>
<td>732</td>
<td>14</td>
<td>Tesla_K20X</td>
<td>nVidia_Kepler</td>
<td>GPU</td>
</tr>
<tr>
<td>465</td>
<td>c7-0c2s7n3</td>
<td>compute</td>
<td>0</td>
<td>6144</td>
<td>732</td>
<td>14</td>
<td>Tesla_K20X</td>
<td>nVidia_Kepler</td>
<td>GPU</td>
</tr>
</tbody>
</table>

Simple Processing Flow:

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory

Additionally you can do MPI directly from a GPU memory to a remote node but you still need to initiate it from the CPU side, give MPI pointers. Also GPU can create the work for themselves.

Single Precision is important to notice: because more and more applications are becoming mixed precision applications, means they run the single precision most of the time and they drop down to the double precision only when they need to converge or they need accuracy.

Kepler Memory Subsystem-L1,L2,ECC

Kepler offers flexibility in configuring the allocation of shared memory and L1 cache by permitting 32KB /32 KB split between shared memory and L1 cache. 48KB Read_Only Data Cache is directly accessible to the SM for general load operations. Use of the read-only path can be managed automatically by the compiler or explicitly by the programmer. The L2 cache is the primary point of data unification between the SMX units, servicing all load, store and texture effects providing efficient, high speed data sharing across GPU.

SMX: (Streaming Multiprocessor)

SMX: 192 single-precision CUDA cores (and each core has fully pipelined floating-point and integer arithmetic logic units), 64 double-precision units, 32 special function units (SFU), and 32 load /store units. The SMX schedules threads in groups of 32 parallel threads called warps. Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Each thread can access to up to 255 registers.
Shuffle Instruction

Allows threads within a wrap to share data, any thread can read from any another thread (arbitrary indexed references). Shuffle offers a performance advantage over shared memory, in that a store-and-load operation is carried out in a single step. Shuffle also can reduce the amount of shared memory needed per thread block, since data exchanged at the wrap level never needs to be placed in a shared memory.

Hyper-Q:

The ability of multiple CPU cores to all issue work to the GPU so that GPU can stay as maximally loaded and busy as possible. Hyper-Q enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing GPU utilization and slashing CPU idle times. This feature increases the total number of connections between the host and the the Kepler GK110 GPU by allowing 32 simultaneous, hardware managed connections, compared to the single connection available with Fermi. Hyper-Q is a flexible solution that allows separate connections from multiple CUDA streams, from multiple Message Passing Interface (MPI) processes, or even from multiple threads within a process. GPUs today are getting so powerful, so much throughput, so many processors, it become difficult for CPU core to keep GPU busy: there are SM cores who are able to do job on behalf of CPU cores, or CPU has allocated parallel work to GPU. Before GPU were able to receive work from many processors but efficiently it could perform only one processors work at the time, and as a result of that the work was queued up, in another words the CPU was unable to keep it busy. Then with Hyper-Q we have the multiple queue work system with which we can receive the 32 processors cores at the same time. Each one of those queues is fully scheduled, synchronized and managed all by itself, all in in the hardware.

The Hyper-Q feature foe MPI is addressed with MPI (Cuda) Proxy:

Hyper-Q is a flexible solution that allows connections for both CUDA streams and Message Passing Interface (MPI) processes, or even threads from within a process.

While it has always been possible to issue multiple MPI processes to concurrently run on the GPU, these processes could become bottleneched by false dependencies, forcing the GPU to operate below peak efficiency. Hyper-Q removes false dependency bottlenecks and dramatically increases speed at which MPI processes can be moved from the system CPU(s) to the GPU for processing.
If you have a Proxy server running on CPU and all MPI ranks on this node funnel their request for the GPU to the Proxy server and the Proxy server schedules the work on the GPU. This means you can run more than one MPI process on the GPU at the same time, and with only Proxy Server accessing the GPU you have just one contact.

Dynamic Parallelism

Dynamic Parallelism is the ability to have the work created within the threads: for the GPU to create the work itself based on the data and the simulation results of previous loops and previous kernels. Without Dynamic Parallelism the CPU launches every kernel onto the GPU: we take a data parallel structure, we receive it from CPU, we accelerate it, we do the work, we take it back. Often algorithms have nested loops, where the subsequent loops are data dependent and that data dependency is not known until the run time. As a result of that whenever we take a data structure, we simulate it, we find the answer, we send the answer back to CPU and than CPU has to figure out what next to do in the algorithm, send more work for the GPU to accelerate, we gather the data and send the data back. It goes back and for the multiple times, and the structure which we see in our algorithm reflects that. So we didn't have the ability to create the work on GPU depending on data until Kepler's GPU.

With Kepler we can now on every single thread, and every single thread concurrently it can generate a work for itself depending on the answers of the data parallel structures, that data, the resulting answer can influence what the next data structure to spawn. So many algorithm have nested loops which are simply impossible to completely unroll without understanding the results of the previous loops, they are data dependent: recursive algorithms, nested algorithms. With Dynamic Parallelism any kernel can launch another kernel and can create the necessary streams, events, and dependencies needed to process additional work without the need for host CPU interaction. This simplified programming model is easier to create, optimize, and maintain. It also creates a programmer friendly environment by maintaining the same syntax for GPU launched workloads as traditional CPU kernel launches.

Parallel Computing Platform
3 Ways to accelerate Applications:

Some of the leading apps now accelerated by GPU’s is NAMD, Charmm …Here it is the latest catalogue

Disclaimer: all images are courtesy of NVIDIA.