GPUs for HPC

Stan Tomov
02/25/2009
Introduction

- **GPUs: excelling in graphics rendering**

  ![Diagram showing the process from Scene model to Final image through streams of data and graphics pipelined computation.]

  Repeated fast over and over: e.g. TV refresh rate is 30 fps; limit is 60 fps

  **Note**: the rendering could be done on the CPU but the market has sustained the demand for significantly higher performance than that of the CPU (GPUs have been doubling the performance every year). Major CPU manufacturers are currently trying to include GPU functionality back on the chip (e.g. AMD Fusion plans, Intel’s Larrabee, etc).

- **Evolution: from fixed function graphics pipeline**
  - To fully programmable graphics pipeline
  - To powerful stream processor
  - To even **more functionality** (multithreading, pointers, asynchronicity, levels of memory hierarchy, etc.)
  - And **programmability** (with CUDA no need to know graphics to program for GPUs; can use CUDA libraries to benefit from GPUs even without knowing CUDA)
GPUs for HPC

Excelling in Graphics Rendering

An operation that

- Requires enormous computational power
- Allows for high parallelism
- Stresses more on high bandwidth vs low latency (because of a deep graphics pipeline)

Powerful General Purpose GPU

Computational pattern, common with many applications (but still not all applications map well)

Hybrid GPU + CPU Computing

Split the computation to fully exploit the power that each of the hybrid components offers (GPUs + multicores)
## GPUs for HPC

HPC applications and **high speedups** reported using GPUs

(from NVIDIA CUDA Zone homepage)

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cubic Interpolation</td>
<td>327 x</td>
</tr>
<tr>
<td>Sliding-Window for Rapid Object Class Localization: A Parallel Technique</td>
<td>109 x</td>
</tr>
<tr>
<td>Jacket: GPU Engine for MATLAB</td>
<td>50 x</td>
</tr>
<tr>
<td>FIR and QR Decomposition on GPUs</td>
<td>35 x</td>
</tr>
<tr>
<td>Computational Fluid Dynamics (CFD) using GPUs</td>
<td>17 x</td>
</tr>
<tr>
<td>Graphic-Card Cluster for Astrophysics (GraCCA)</td>
<td>250 x</td>
</tr>
<tr>
<td>GpuCV: GPU-accelerated Computer vision library</td>
<td>100 x</td>
</tr>
<tr>
<td>Distributed Password Recovery</td>
<td>50 x</td>
</tr>
<tr>
<td>General Relativistic Evolution Code</td>
<td>26 x</td>
</tr>
<tr>
<td>Highly Optimized Object-oriented Molecular Dynamics: HOOMD</td>
<td>15 x</td>
</tr>
<tr>
<td>Quantum Chemistry Two-Electron Integral Evolution</td>
<td>130 x</td>
</tr>
<tr>
<td>A Fast Similarity Join Algorithm</td>
<td>100 x</td>
</tr>
<tr>
<td>Accelerating Density Functional Calculations with GPU</td>
<td>40 x</td>
</tr>
<tr>
<td>Molecular Dynamics of DNA and Liquids</td>
<td>18 x</td>
</tr>
<tr>
<td>Computational Chemistry Using GPUs</td>
<td>4.3 x</td>
</tr>
</tbody>
</table>
Current NVIDIA GPUs

<table>
<thead>
<tr>
<th></th>
<th>GeForce GTX 280</th>
<th>GeForce GTX 260</th>
<th>Tesla C1060</th>
<th>Tesla S1070</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form Factor</td>
<td>Dual slot card</td>
<td>Dual slot card</td>
<td>Dual slot card</td>
<td>Rackmount</td>
</tr>
<tr>
<td>TPCs</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>4x10</td>
</tr>
<tr>
<td>SMs</td>
<td>30</td>
<td>24</td>
<td>30</td>
<td>4x30</td>
</tr>
<tr>
<td>SPs</td>
<td>240</td>
<td>192</td>
<td>240</td>
<td>4x240</td>
</tr>
<tr>
<td>Graphics Freq.</td>
<td>602MHz</td>
<td>576MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor Freq.</td>
<td>1296MHz</td>
<td>1242MHz</td>
<td>1300MHz</td>
<td>1500MHz</td>
</tr>
<tr>
<td>Memory Freq.</td>
<td>1107MHz</td>
<td>999MHz</td>
<td>800MHz</td>
<td>800MHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>141.7GB/s</td>
<td>127.9GB/s</td>
<td>102.4GB/s</td>
<td>4x102.4GB/s</td>
</tr>
<tr>
<td>Memory Capacity</td>
<td>1GB</td>
<td>896MB</td>
<td>4GB</td>
<td>4x4GB</td>
</tr>
<tr>
<td>Power</td>
<td>236W TDP</td>
<td>183W TDP</td>
<td>160W &quot;Typical&quot;</td>
<td>700W &quot;Typical&quot;</td>
</tr>
<tr>
<td>SP GFLOP/s (wo/MUL)</td>
<td>622.1</td>
<td>476.9</td>
<td>624.0</td>
<td>4x720.0</td>
</tr>
<tr>
<td>SP GFLOP/s (w/MUL)</td>
<td>933.1</td>
<td>715.4</td>
<td>936.0</td>
<td>4x1080.0</td>
</tr>
<tr>
<td>DP GFLOP/s</td>
<td>77.8</td>
<td>59.6</td>
<td>78.0</td>
<td>4x72.0</td>
</tr>
</tbody>
</table>

(Source: "NVIDIA's GT200: Inside a Parallel Processor")
GPUs for DLA

- Many DLA algorithms, due to high ratio of floating point calculations to data required, have been of very high performance.

- Therefore, special purpose architectures have not been able to significantly accelerate them up until recently.
  - Fatahalian et al. study SGEMM (in 2004) to conclude CPUs almost always outperform GPUs (only ATI X800XT produced 12 Gflop/s in single precision, comparable with a 3 GHz Pentium 4).
  - Galoppo et al. (in 2005) had similar results on LU (5.7 Gflop/s in single precision on an NVIDIA 7800, compared to 3.4GHz Pentium 4 at the time).

- This has changed as CPUs move to multi/manycores with an exponentially growing gap between processor speed and memory (and bandwidth shared between cores), while GPUs have consistently outpaced them both in performance and memory bandwidth.
GEMM on Current CPUs and GPUs
(Intel multicores and NVIDIA GPUs)

Note that in SP the GTX 280 is **10 x faster** than a quad-core processor (at 2.33 GHz) and still **75 GFlop/s** faster than an entire quad-socket quad-core Intel Xeon Tigerton system (cores running at 2.4 GHz)
Computing on the GPU

From **non-programmable hardware**
(fixed-function pipeline)

To **simple streaming**
(data viewed as streams, computation as kernels)

**more powerful streaming**
(gather, scatter, reduction, etc more CPU features)

**CUDA (NVIDIA) / CTM (ATI)**
* general DRAM addressing (powerful gather, scatter, etc.)
* very fast shared memory
* support for DP arithmetic

... ?
* fusion with CPU?
CUDA

We can easily use LAPACK with CUBLAS C-like API.
Programming model

A highly multithreaded coprocessor

* thread block
  (a batch of threads with fast shared memory executes a kernel)

* Grid of thread blocks
  (blocks of the same dimension, grouped together to execute the same kernel; reduces thread cooperation)

```cpp
// set the grid and thread configuration
Dim3 dimBlock(3,5);
Dim3 dimGrid(2,3);

// Launch the device computation
MatVec<<<dimGrid,
dimBlock>>>( . . . );
```

```cpp
__global__ void MatVec( . . . ) {
  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  . . .
}
```
Memory model

Quadro FX 5600:
Theoretical memory bandwidth: 76.8 GB/s
Interface: PCI Express x 16
* up to 4GB/s peak per direction
* up to 8BG/s concurrent bandwidth
Hardware Model

Quadro FX 5600

Some numbers:
- processors: 128 (total)
- registers: 8192 per block
- warp size: 32
- max threads per block: 512

Execution of Grid of thread blocks:
- one or more blocks are executed on multiprocessor using time slicing
- each block is split into group of threads (called warps)
- a block is processed by only 1 multiprocessor
- issue order of warps within a block is undefined (but execution can be synchronized)
- issue order of blocks within a grid is undefined
  - thread of different blocks (on the same grid) can not safely communicate through global memory
- many other issues ... (in the manual)
Performance issues

• Instruction throughput
  - Arithmetic instructions (some more 'expensive' than others)
  - Control flow (if, switch, do, for, while can significantly impact on performance)
  - Memory instructions (400..600 cycles in memory latency to read/write to global memory; 4 cycles for shared can be hidden by thread scheduler if there is enough computation to be overlapped with)
  - Thread synchronization (4 cycles to issue for a warp if no thread has to wait)

• Memory bandwidth
  - Load data to shared memory -> synchronize with other block threads to have safe/fast read/writes -> process -> synchronize again for updates -> write result back
  - Global memory (aligning data for faster access; not cached)
  - Constant memory (cached)
  - Texture memory (cached; optimized for 2D spatial locality)
  - Shared memory (divided into 16 banks; can be accessed simultaneously; successive 32-bit words go into successive banks; bandwidth: 32 bits/2 cycles)

• Other (tune number of threads/block; data transfer between CPU and GPU, etc ...)
Programming DLA for GPUs using CUDA

1. Use BLAS level parallelism, where the matrix resides on the GPU, and the CPU is running for example LAPACK style code, e.g. represented as a sequence of CUBLAS kernels, using the GPU pointer arithmetic
2. Offload small kernels, inefficient for the GPU to the CPU
3. Use asynchronicity between CPU and GPU whenever possible in the offload/load process

Example:

Standard LAPACK pseudo-code
ssyrk("L", "N", &jb, &i_3, &c_b13, a_ref(j,1), ...)
spotrf("L", &jb, a_ref(j, j), lda, info)
sgemm("N", "T", &i_3, ...)
strsm("R", "L", "T", "N", &i_3, ...)

Hybrid Single Core-GPU code

ssyrk("L", "N", &jb, &i_3, &c_b13, a_ref(j,1), ...)

((GPU)
spotrf("L", &jb, a_ref(j, j), lda, info)
sgemm("N", "T", &i_3, ...)
strsm("R", "L", "T", "N", &i_3, ...)

Hybrid implementation:
• a_ref points to the GPU memory
• GPU kernels are started asynchronously which results in overlapping the GPU's sgemm with CPU's spotrf

Cholesky factorization in SP
Here we considered 3 steps of iterative refinement on symmetric and positive definite matrices (using Cholesky).

**CPU**: Intel Xeon 2.33 GHz

**GPU**: NVIDIA GTX280
Current work

Main factorizations in **DP** on **CPU + GPU**

- Auto-tuning
Limited amount of pivoting (within the block size NB or more; denoted by LP) is justified by a specially designed unitary transformation: experiments with random matrices show that LP LU (NB+64) for example is comparable in accuracy to PP LU, and LP LU(NB) looses about 2 digits of accuracy to gain up to 30% in speed compared to PP LU.
The Challenge of Tuning

- Algorithms for hybrid systems have a large design space which makes them difficult to hand-tune

Shown are results on “auto-tuning” the DGEMMs occurring in the block LU factorization of matrix of size 6144 x 6144.

GPU: NVIDIA GTX280
We used V. Volkov’s CUDA based SGEMM to implement code generators for both SGEMM and DGEMM that take 6 parameters (block sizes $BM$, $BK$, $BN$; thread block $Tx$, $Ty$; and Trans for BA) as input to generate various kernels which are run on the platform of interest to determine the one that gives best performance.
Conclusions

• **GPU computing**: significantly outperform current multicores on many real world applications (illustrated for DLA which has been traditionally of HP on x86 architectures)

• **Hybrid GPU+CPU computing**: there are still applications - or at least part of them - that do not map well on GPU architectures and would benefit much more a hybrid one

• **Architecture trends**: towards heterogeneous designs, where both GPU and CPU functionalities are needed (currently major CPU manufacturers start to include more GPU features and vice-versa)

• **The need for DLA for hybrid systems will grow**

  motivating our Future work directions

• **Towards** a self contained **DLA library** similar to LAPACK but **for heterogeneous architectures** (currently featuring hybrid manycore / GPU systems)

  for more information see the **MAGMA** project