When is a GPU a good solution?

Robert Strzodka
Application Specific Computing
Heidelberg University
Contents

• Comparison of CPUs and GPUs
• Programming Styles
• What is Easy to Accelerate?
• Libraries
CPU or GPU?

Which die is the CPU, which one the GPU?

GK110

XEON-E7
Bandwidth in an Accelerator System

CPU socket (4-16 cores)

- CPU core
  - SIMD 8x
  - 40 GB/s
  - on-chip memory

- Channels: 15 GB/s
- System memory: 200GiB-2000GiB

Accelerator socket

- SIMD 32x
- 2000 GB/s
- On-chip memory
- 300 GB/s
- Device memory: 20GiB

- 4 GB/s
- 12 GB/s
## GPUs vs. CPUs

<table>
<thead>
<tr>
<th></th>
<th>Tesla K20</th>
<th>Xeon E7-4800 4P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core count</strong></td>
<td>13 SMs 64/832 (DP), 192/2,496 (SP)</td>
<td>10 Cores</td>
</tr>
<tr>
<td><strong>Frequency</strong></td>
<td>0.7GHz</td>
<td>2.4GHz</td>
</tr>
<tr>
<td><strong>Peak Compute Performance</strong></td>
<td>1,165 GFLOPS (DP) 3,494 GFLOPS (SP)</td>
<td>96 GFLOPS (DP)</td>
</tr>
<tr>
<td><strong>Use model</strong></td>
<td>throughput-oriented</td>
<td>latency-oriented</td>
</tr>
<tr>
<td><strong>Latency treatment</strong></td>
<td>toleration</td>
<td>minimization</td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td>1000s-10,000s of threads</td>
<td>10s of threads</td>
</tr>
<tr>
<td><strong>Memory bandwidth</strong></td>
<td>250 GBytes/sec</td>
<td>34 GByte/s (per P)</td>
</tr>
<tr>
<td><strong>Memory capacity</strong></td>
<td>5 GB</td>
<td>up to 2TB</td>
</tr>
<tr>
<td><strong>Die size</strong></td>
<td>550mm²</td>
<td>684 mm²</td>
</tr>
<tr>
<td><strong>Transistor count</strong></td>
<td>7.1 billion</td>
<td>2.3 billion</td>
</tr>
<tr>
<td><strong>Technology</strong></td>
<td>28nm</td>
<td>32nm</td>
</tr>
<tr>
<td><strong>Power consumption</strong></td>
<td>250W</td>
<td>130W</td>
</tr>
<tr>
<td><strong>Power efficiency</strong></td>
<td>4.66 GFLOPs/Watt (DP) 14 GFLOPs/Watt (SP)</td>
<td>0.74 GFLOPs/Watt (DP)</td>
</tr>
</tbody>
</table>
Parallelism on CPUs and GPUs

**CPU**
- SIMD AVX 32B, Phi 64B
  - MADD with 8-16 floats
  - MADD with 4-8 doubles
  - Coding: explicit, automatic
- **Minimum** multi-threading
  - #threads=#cores (good)
  - #threads= 2*#cores (good)
  - #threads= 10*#cores (difficult)
  - Coding: explicit with resident threads, implicit with libraries

**GPU**
- SIMD warp size 32
  - MADD with 32 floats
  - MADD with 32 doubles
  - Coding: implicit, partly explicit
- **Maximum** multi-threading
  - #warps=#’cores’≈15 (bad)
  - #warps ≈100 (difficult)
  - #warps>1000 (good)
  - Coding: implicit with max. parallelism, explicit (advanced)
Memory on CPUs and GPUs

**CPU**
- Deep and large memories
  - Core: Reg, L1, L2
  - Shared: L3, eDRAM
  - Coding: implicit
- Usage
  - Optimize 1D locality
  - Optimize size of working sets
  - Prefetch, pipeline
  - Coding: implicit, explicit

**GPU**
- Smaller and specialized memories
  - Core: Reg, L1 or shmem
  - Shared: L2, constant
  - Coding: explicit, implicit with libs
- Usage
  - Optimize 1D, 2D, 3D locality
  - Decide on data location: Reg, L1, shmem, constant
  - Many warps and low latency vs. amount of local data per warp
  - Coding: explicit
CPU or GPU?
Which die is the CPU, which one the GPU?

GK110

XEON-E7
# Similarities and Differences

## CPU
- For high performance
  - SIMD
  - Multi-threading
  - Memory access alignment
  - Minimal latency with large caches
  - Working set opt. wrt deep caches
  - Locality opt. wrt cache lines, NUMA
- Opt. serial performance
  - High normal and boost frequency
  - Low latency caches
  - Speculative execution

## GPU
- For high performance
  - SIMD
  - Multi-threading
  - Memory access alignment
  - Minimal latency with many warps
  - Working set opt. wrt #warps
  - Locality opt. wrt memory types
- Opt. throughput performance
  - Lower normal and boost frequency
  - L2 latency is high
  - No speculative execution
CUDA Ecosystem
What is Easy to Accelerate?
Embarrassingly Parallel Loop

- for(int i=0; i<SIZE; ++i) {
  c[i] = a[i+1] + b[i] * a[i-yoff];
  func(a,b,c,i);
}

- Relevant for performance
  - SIZE > 10k
  - Arithmetic intensity
  - Regularity of memory access
  - Amount of local state and reuse
Branches in Loops

• for(int i=0; i<SIZE; ++i) {
  if(cond(i)) special_func(a,b,c,i);
  else normal_func(a,b,c,i);
}

• Only a problem if all these conditions hold:
  – Special case is more than 10% of cases
  – Normal and special case differ largely in execution times
  – Data of special cases is scattered in memory
Index Dependencies

- for(int i=1; i<SIZE; ++i) {
  a[i] += a[i-1];
}

- Replace serial dependence
  - Use equivalent parallel variant
  - If allowed, use approximate parallel variant
  - Check if parent computation can use other ingredients
Data Movement is Critical

• CPU_func1(a,b,c);
  GPU_func1(a,b,c); // implicit transfer!
  CPU_func2(a,b,c);
  GPU_func2(a,b,c); // implicit transfer!

• Such CPU-GPU alternation only works well if
  – Execution time of GPU_func* is at least a millisecond
  – High arithmetic intensity, e.g. matrix*matrix, not matrix*vector

• Otherwise
  – GPU must perform multiple operations on the same data, e.g. multiple
    vector-vector or matrix-vector operations.
Linked List

- for(; elm!=nullptr; elm= elm->next) {
  func(elm->data);
}

- Do not do this!
  - Unless all parallelism can be used efficiently in func()
  - Terrible performance on CPUs and GPUs
  - Vector almost always dramatically faster than list
  - Even insert(pos), delete(pos) much faster in vector if we first search for pos
# Data Structures on GPUs

<table>
<thead>
<tr>
<th></th>
<th>Difficulty</th>
<th>Speed</th>
<th>Support</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>easy</td>
<td>fast</td>
<td>everywhere</td>
<td>contiguous data</td>
</tr>
<tr>
<td>dense matrix</td>
<td>easy</td>
<td>fast</td>
<td>many libs</td>
<td>contiguous data</td>
</tr>
<tr>
<td>sparse matrix</td>
<td>moderate</td>
<td>fast</td>
<td>many libs</td>
<td>CSR, BCSR, (B)CSC, COO, special</td>
</tr>
<tr>
<td>graph</td>
<td>moderate</td>
<td>fast</td>
<td>multiple libs</td>
<td>CSR, special</td>
</tr>
<tr>
<td>tree</td>
<td>difficult</td>
<td>fast</td>
<td>little</td>
<td>various special formats</td>
</tr>
<tr>
<td>list</td>
<td>moderate</td>
<td>slow</td>
<td>none</td>
<td>special formats</td>
</tr>
</tbody>
</table>
Where to Put the Parallelism?

- for(int i=0; i<SIZE; i+=block_size) {
  func1(a,b,c,i,block_size);
}
- for(int i=0; i<SIZE; i+=block_size) {
  func2(a,b,c,i,block_size);
}
- for(int i=0; i<SIZE; i+=block_size) {
  func3(a,b,c,i,block_size);
}
Libraries
GPU Libraries

- **Dense linear algebra**
  - cuFFT: Fast Fourier transforms
  - cuBLAS: BLAS operations
  - MAGMA, FLAME: LAPACK operations
- **Sparse linear algebra**
  - cuSPARSE: Sparse matrix operations
  - cuSOLVER: Sparse direct solvers
  - AmgX: Sparse iterative solvers
- **General tools**
  - cuRAND: Random number generation
  - Thrust: Algorithms & data structures
- **Specialized libraries**
  - NPP: Primitives for image & video
  - NVBIO: Sequence analysis
  - cuDNN: Primitives for neural networks
- **Any type of matrix and vector operations**
  - Vector-vector
  - Dense matrix with vector
  - Sparse matrix with vector
  - Graph operations (sparse matrix)
- **Matrix decompositions and solvers of linear equation systems**
  - With dense and sparse matrices
  - For example: LU, QR, Ax=b
- **STL type algorithms**
  - For example: transforms, merges, sorting
  - Composition of algorithms
- **Specialized libraries**
Performance
Summary

• For high performance on CPUs and GPUs
  – High parallelism and high data locality
  – Optimizations are similar in concept, but different and very involved in detail
  – Difficult to do by hand → use libraries

• What is easy to accelerate?
  – Large loops with no/simple index dependencies
  – Data placement and movement are crucial
Resources

• CUDA toolkit, current version 7.5
  – Compiler & tools, **libs**, **docs** and **samples**

• More libraries