Peta Thread Processing

Michael C. Shebanow
Vice President, Advanced Processor Lab,
Samsung Electronics, SRA
Disclaimer

• All of the positions and opinions in this presentation are my own.

• Nothing in this presentation should be construed to represent Samsung’s opinion, present business operations, or strategic direction.
Motivation

PETA THREADS
What is Peta Thread Processing?

- **Thread = Virtual machine**
  - Von Neumann processor (stored program computer)
  - “Processor State” (PC, registers, etc.)
  - “Memory State” (stored program, non-volatile data)
  - “I/O” (volatile data)

- **Peta Thread Processing =**
  - Up to $\sim 10^{15}$ “threads” running simultaneously
  - Single, common shared memory
Why Peta Threads?

- HW deployment rapidly approaching that capability
- Some problems need both cloud and client
- Seamless means to blend the two to solve very hard problems?
Peta-thread-scale Applications?

• Killer App?
  – Improved UX (user experience)

• Some examples:
  – AR (augmented reality): Google Glass: real-time annotation of a view
    • LBS integrated with camera orientation integrated with scene recognition
    • Scene processing specific to client
    • Project annotations onto background
  – Enhanced “shopping AR”
    • LBS and object recognition
    • Big data search for comparables
    • Annotation of pricing options, availability, etc.

• Anything integrating Big Data, mobile client AR, real time deadlines, and parallelizable (!)
How Can Threading Help?

- **Divide and Conquer**
  - Take a problem that takes time $T$
  - Divide it into two parallel problems that take time $\geq T/2$
  - Iterate

- **“Map-Reduce”**
  - Obviously not always easy
  - BUT, if we can construct algorithms that are “embarrassingly parallel”...
Map-Reduce Time Model

- Definitions:
  - $T_e = \text{expansion time step}$
  - $T_m = \text{map time step}$
  - $T_r = \text{reduction time step}$
  - $N = \text{expansion factor}$
  - $T_c = \text{compute time}$

- We assume “work” (threads) expands exponentially at some rate $E$ each expansion step
  - $E \rightarrow E^2 \rightarrow E^3 \rightarrow \ldots \rightarrow E^N$

- Then:
  \[
  T_c = (N-1) T_e + T_m + N T_r
  \]

{expansion} {map} {reduce}
Solving for Expansion Rate Constraints

- Invert the Compute Time relation to compute maximum expansion factor, N
  - Number of parallel threads is $E^N$
  - For $E = 32$, a “warp”, grows pretty quickly
- Formula:

  \[ N = \frac{T_c - T_m + T_e}{T_r + T_e} \]

- For the graphs at right:
  - Expansion time step = \{1% or 50%\} of Map time step
  - Reduction time step = \{10%, 20%, 50%, or 100%\} of Map time step
  - Sweep compute time as a multiple of Map time step (\{~>1, 2, 4, 8, ... 64\}
Solving for Maximal Map Time Constraints

- Similarly, given a compute time constraint and a maximum expansion factor, compute a limit on map time
- Assume expansion time step and reduction time steps are fractions of map time step
- Formula:

\[ T_m \leq \frac{T_c}{(N-1)T_e + 1 + NT_r} \]

- For the graphs at right:
  - Selected compute time to 25 msec or 1 sec (short/long)
  - Four different reduction time steps as a percentage of map time
  - Sweep expansion factor against fixed compute time
- And, \(32^{10} = \) a peta thread
The Rise of the GPU (Capability)

A HIGHLY THREADED WORLD
GPU Market Penetration

- Jon Peddie Research (JPR) has estimated GPU deployment through 2018
- Each core $O(10^3$ to $10^4$) threads
- Net: in 2018, $O(10^{14})$ threads deployed
Mobile Market Impact

• In 2012, an estimated 800+ million mobile GPUs shipped
  – ~123M tablets
  – ~712M smart phones
• Will easily exceed 1B in the coming years
• Each GPU ranges up to $O(10^4)$-way multithreading
  – Net: $O(10^{13})$ thread-capable GPUs shipped each year
  – In mobile alone
• Over time, likely to increase
  – Discrete GPU relatively flat
  – Mobile is growing rapidly
WW Internet Traffic

• Source: Cisco VNI

• Internet traffic growth rate is staggering
  – Current traffic (2011) is on average 47.3 GB per person per year for the whole planet
  – Mobile traffic alone (2011) is on average 1.03 GB per person per year for the whole planet

<table>
<thead>
<tr>
<th>Year</th>
<th>IP Traffic (TB/sec)</th>
<th>growth rate</th>
<th>Fixed INET Traffic (TB/sec)</th>
<th>Mobile INET Traffic (TB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>0.9</td>
<td></td>
<td>0.8</td>
<td>0.00</td>
</tr>
<tr>
<td>2006</td>
<td>1.5</td>
<td>65%</td>
<td>1.3</td>
<td>0.00</td>
</tr>
<tr>
<td>2007</td>
<td>2.5</td>
<td>61%</td>
<td>2.0</td>
<td>0.01</td>
</tr>
<tr>
<td>2008</td>
<td>3.8</td>
<td>54%</td>
<td>2.9</td>
<td>0.01</td>
</tr>
<tr>
<td>2009</td>
<td>5.6</td>
<td>45%</td>
<td>4.1</td>
<td>0.04</td>
</tr>
<tr>
<td>2010</td>
<td>7.8</td>
<td>40%</td>
<td>5.8</td>
<td>0.10</td>
</tr>
<tr>
<td>2011</td>
<td>10.6</td>
<td>36%</td>
<td>8.0</td>
<td>0.23</td>
</tr>
</tbody>
</table>
Where the World is Heading...

• Enormous quantity of GPUs
  – Each with more and more threads
  – Total thread-level parallelism will be at peta scale

• Large amount of interconnectivity

• GPUs could (will?) penetrate the cloud as well
Key features...

CURRENT GPUS
GPU Block Diagram

- Canonical GPU model
- “System Interface” talks to CPU driver
- “Work Distributor” launches “thread blocks” to 1 of N cores
- Cores execute blocks, accessing system memory as required
Thread Launches

- GPUs generally have very high launch BW
  - Driven by need to have good “PPC” (pixels per clock)
  - Can be as high as 1 warp per clock per core
- Each “launch” requires:
  - Allocating all thread-specific state structures (registers, private memory, etc.)
  - Allocating thread-block shared memory
  - Initializing thread-private and thread-block-private state
Thread States

Every thread has 4 states:

- INVALID → dead
- READY → code can be executed
- RUN → thread is executing code
- WAIT → thread is blocked waiting for some event
Warp Selection

• Threads in a GPU are grouped into warps
  – For example, warp = 32 threads
• At some rate, GPU core picks a next warp to execute (from pool of READY warps)
  – “Warp Selection”
• Mimics SW OS scheduler, but in HW
GPU Memory Pipeline

WARP 0
- INVALID ID
- READ
- WAIT
- RUN
- exit

WARP 1
- INVALID ID
- READ
- WAIT
- RUN
- exit

WARP N-1
- INVALID ID
- READ
- WAIT
- RUN
- exit

WARP SELECTION

AGEN

L1$

“MEM”
Handling Stalls…

• Memory “pipeline” (purple flops) is “L” stages deep
• Throughput of memory system is $\lambda$ measured in requests per clock
• Little’s Law defines # of threads needed to keep the pipe full

$$N = \lambda L$$
Batching

- A “batch” is a string of independent load operations
- Stall doesn’t occur until the first use of a load result
- Batching reduces pressure on thread count

\[ N = \lambda \frac{L}{B} \]
Hierarchical streaming processing

NETWORK GPUS
Parallelism

- Parallel vs. Sequential
  - Parallel $\rightarrow$ independence
  - Sequential $\rightarrow$ dependence

- Three fundamental forms of parallelism
  - Spatial: executing operations between threads at the same time
  - Temporal: executing operations between threads at the same place
  - ILP: executing operations from within the same thread in parallel

- Fundamental differences between ILP-only machines and massive TLP-ILP machines
  - CPUs vs. GPUs
Throughput vs. Latency

- Throughput = rate at which operations complete
- Latency = time it takes to complete an operation or set of operations

- CPUs versus GPUs
  - In CPUs, the primary objective is low latency
  - In GPUs, the primary objective is high throughput

- CPUs versus GPUs
  - In an application suitable for CPUs, we assume a low degree of TLP
  - In an application suitable for GPUs, we assume a high degree of TLP
What is a “Network GPU”?

• Concept:
  – Treat systems as GPU cores
  – Treat each system’s memory as part of a COMA
  – Treat the “network” as interconnect
  – Employ SW work distributors
“Real” GPU vs. “Network GPU”

“Real” GPU

- Processor cores = shader engines
  - O(1000) threads
  - High internal BW (125-1000 GB/sec per core)
- Relatively short latencies
  - O(1 usec)
- HW Thread Launch (~500M warps/sec per core)

“Network GPU”

- Orders of magnitude more cores – each GPU is in effect a core
  - O(10^{12} to 10^{15}) threads foreseeable
  - Lower “core” to memory BWs (6-200 GB/sec per GPU)
  - Inter-“core” BWs at network rates
- Latencies from GPU scale (O(1 usec)) to very long (10’s, 100’s of msec)
- SW-induced thread launches
Some Key Challenges

• Memory latencies, locality, and addressing
• Thread creation overhead
• Virtual system
• Property rights
Hiding Long Latencies

- Long latencies can be hidden, but warp count might be quite large
  - Implication: need to keep most references “local”
- Some GPU memory systems are “in order”
  - Forces all latencies to equal the worst latency
Locality Management

• Analogous GPU problem
  – “Inverted Wedding Cake”
  – The closer the memory is to the processor:
    • The higher the capacity
    • The higher the available BW

• Data must be kept close to the processor

• Possible solutions:
  – Migration
  – Replication
  – Paging
Addressing

• Potentially a tremendous amount of memory
  – \(O(10^{20})\) bytes DRAM alone (soon an underestimate?)

• Addressing a critical issue

• One solution: capabilities
Single System Image (SSI) OS

• Old idea
  – An OS that spans multiple HW systems, perhaps the “net”
  – UCLA Locus 1983 → Tandem → OpenSSI
  – UCB Sprite 1984
  – Amoeba ~1991 (python)

• Peta-scale?
Why SSI?

• Simple virtual machine model, easy to program

• Invisible to the programmer:
  – Ability to dynamically migrate tasks and data across system boundaries
  – Ability to access data across system boundaries
Thread Creation

- From the map-reduce model, low thread creation overhead is ideal

- Today’s systems:
  - No way to have HW threads in system “B” due to activity in system “A”

- Can we add cross-system thread launch?
  - HW cross-system queue?
“Not in MY Backyard…”

• Resource allocation is a key issue
  – Property rights
• Does client A get to run threads on client B’s HW?
• Mobile device sharing:
  Power
  – Phones today are ~5-7 watt-hours
Use Economics?

• One option is to use economics to control resources
• For example, solar energy
  – You generate more than you use, you get paid
  – You use more than you generate, you do the paying
• Extend to network-based processed
  – Pay as you go use
  – Credit for resource sharing
• “Push-Pull” Economy
In summary...

RECAP
The Opportunity

• If we could build a peta thread scale virtual machine, possible killer apps:
  – Big Data
  – Client AR
  – Real-time constraints
  – Parallelizable
GPUs

- GPUs playing a transformative role in computing
- At their heart, GPUs rely on streaming programmable cores
- The streaming programming model well adept at tolerating long latencies
- Exploiting TLP, GPUs capable of running from 1000s to tens of 1000s of threads simultaneously
“Network GPUs”

- Huge number of GPUs deployed
  - World collectively can run near peta thread computing
- Similar to the inner workings of a GPU, analogously extend the GPU model to network scale
Key Challenges

• Latency, Locality, Addressing
• Single System Image OS
  – Unified thread model
  – Unified storage model
  – Unified connectivity model
• Cross-system HW-accelerated thread/storage/connectivity managers
  – We need to keep “expansion” overhead low
  – Need rapid access to shared resources
• Property rights management
That's all Folks!