Data-Parallel Programming on Manycore Graphics Processors

Bryan Catanzaro

Universal Parallel Computing Research Center
University of California, Berkeley
Overview

- Terminology: Multicore, Manycore, SIMD
- The CUDA Programming model
- Mapping CUDA to Nvidia GPUs
- Experiences with CUDA
Multicore and Manycore

- **Multicore: yoke of oxen**
  - Each core optimized for executing a single thread
- **Manycore: flock of chickens**
  - Cores optimized for aggregate throughput, deemphasizing individual performance
### Multicore & Manycore, *cont.*

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Core i7 960</th>
<th>GTX285</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Elements</td>
<td>4 cores, 4 way SIMD @3.2 GHz</td>
<td>30 cores, 8 way SIMD @1.5 GHz</td>
</tr>
<tr>
<td>Resident Strands/Threads (max)</td>
<td>4 cores, 2 threads, 4 way SIMD:</td>
<td>30 cores, 32 SIMD vectors, 32 way SIMD: 30720 threads</td>
</tr>
<tr>
<td></td>
<td>32 strands</td>
<td></td>
</tr>
<tr>
<td>SP GFLOP/s</td>
<td>102</td>
<td>1080</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>25.6 GB/s</td>
<td>159 GB/s</td>
</tr>
<tr>
<td>Register File</td>
<td>-</td>
<td>1.875 MB</td>
</tr>
<tr>
<td>Local Store</td>
<td>-</td>
<td>480 kB</td>
</tr>
</tbody>
</table>

Core i7 (45nm)

GTX285 (55nm)
What is a core?

- Is a core an ALU?
  - ATI: We have 800 streaming processors!!
    - Actually, we have 5 way VLIW * 16 way SIMD * 10 “SIMD cores”
  
- Is a core a SIMD vector unit?
  - Nvidia: We have 240 streaming processors!!
    - Actually, we have 8 way SIMD * 30 “multiprocessors”
      - To match ATI, they could count another factor of 2 for dual issue

- In this lecture, we’re using core consistent with the CPU world
  - Superscalar, VLIW, SIMD, SMT, etc. are part of a core’s architecture, not the number of cores
Single Instruction Multiple Data architectures make use of data parallelism

- SIMD can be area and power efficient
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler
SIMD: Neglected Parallelism

- It is difficult for a compiler to exploit SIMD
- How do you deal with sparse data & branches?
  - Many languages (like C) are difficult to vectorize
  - Fortran is somewhat better
- Most common solution:
  - Either forget about SIMD
    - Pray the autovectorizer likes you
  - Or instantiate intrinsics (assembly language)
  - Requires a new code version for every SIMD extension
A Brief History of x86 SIMD

- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX+FMA
- 3dNow!
- SSE4.A
- SSE5
- Larrabee

Subsets and Future Subsets:
- Subset
- Future Subset

Data Types:
- 8 x 8 bit Integer
- 4 x 32 bit SP Float
- 2 x 64 bit DP Float
- 16 x 32 bit SP Float
- 8 x 32 bit SP Float
- 3 operands
What to do with SIMD?

- Neglecting SIMD in the future will be more expensive
  - AVX: 8 way SIMD, Larrabee: 16 way SIMD, Nvidia: 32 way SIMD, ATI: 64 way SIMD

- This problem composes with thread level parallelism

- We need a programming model which addresses both problems
The CUDA Programming Model

- CUDA is a recent programming model, designed for
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability
- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization & data sharing between small groups of threads
- CUDA programs are written in C + extensions
- OpenCL uses very similar programming model, but is HW & SW vendor neutral
Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program

- Threads are grouped into thread blocks
  - threads in the same block can cooperate

- Threads/blocks have unique IDs
What is a CUDA Thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

- CUDA threads might be physical threads
  - as on NVIDIA GPUs

- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU
What is a CUDA Thread Block?

- Thread block = *virtualized multiprocessor*
  - freely choose processors to fit data
  - freely customize for each kernel launch

- Thread block = a (data) *parallel task*
  - all blocks in kernel have the same entry point
  - but may execute any code they want

- Thread blocks of kernel must be *independent tasks*
  - program valid for *any interleaving* of block executions
Synchronization

- Threads within a block may synchronize with **barriers**
  
  ```
  ... Step 1 ...
  __syncthreads();
  ... Step 2 ...
  ```

- Blocks **coordinate** via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`

- Implicit barrier between **dependent kernels**

  ```
  vec_minus<<<nbblocks, blksize>>>(a, b, c);
  ---------------------------------------------
  vec_dot<<<nbblocks, blksize>>>(c, c);
  ```
Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially

- Blocks may coordinate but not synchronize
  - shared queue pointer: **OK**
  - shared lock: **BAD** ... can easily deadlock

- Independence requirement gives **scalability**
Scalability

- Manycore chips exist in a diverse set of configurations

- CUDA allows one binary to target all these chips
- Thread blocks bring scalability!
Hello World: Vector Addition

//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main() {
    //Run N/256 blocks of 256 threads each
    vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
Flavors of parallelism

- **Thread parallelism**
  - each thread is an independent thread of execution

- **Data parallelism**
  - across threads in a block
  - across blocks in a kernel

- **Task parallelism**
  - different blocks are independent
  - independent kernels
Memory model
Memory model

Sequential Kernels

Kernel 0

Kernel 1

Per Device Global Memory
Memory model

Host Memory

Device 0 Memory

Device 1 Memory

cudaMemcpy()
Using per-block shared memory

- Variables shared across block
  ```c
  __shared__ int *begin, *end;
  ```

- Scratchpad memory
  ```c
  __shared__ int scratch[BLOCKSIZE];
  scratch[threadIdx.x] = begin[threadIdx.x];
  // ... compute on scratch values ... 
  begin[threadIdx.x] = scratch[threadIdx.x];
  ```

- Communicating values between threads
  ```c
  scratch[threadIdx.x] = begin[threadIdx.x];
  __syncthreads();
  int left = scratch[threadIdx.x - 1];
  ```

- Per-block shared memory is very fast
  - Often just as fast as a register file access

- It is relatively small: On GTX280, the register file is 4x bigger
CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live
  ```c
  __global__ void KernelFunc(...);  // kernel callable from host
  __device__ void DeviceFunc(...);  // function callable on device
  __device__ int GlobalVar;        // variable in device memory
  __shared__ int SharedVar;        // in per-block shared memory
  ```

- Extend function invocation syntax for parallel kernel launch
  ```c
  KernelFunc<<<500, 128>>>(...);  // 500 blocks, 128 threads each
  ```

- Special variables for thread identification in kernels
  ```c
  dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
  ```

- Intrinsics that expose specific operations in kernel code
  ```c
  __syncthreads();  // barrier synchronization
  ```
CUDA: Features available on GPU

- Double and single precision

- Standard mathematical functions
  - \texttt{sinf}, \texttt{powf}, \texttt{atanf}, \texttt{ceil}, \texttt{min}, \texttt{sqrtf}, etc.

- Atomic memory operations
  - \texttt{atomicAdd}, \texttt{atomicMin}, \texttt{atomicAnd}, \texttt{atomicCAS}, etc.

- These work on both global and shared memory
CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`

- Explicit memory copy for host ⇄ device, device ⇄ device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...

- Texture management
  - `cudaBindTexture()`, `cudaBindTextureToArray()`, ...

- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...
Mapping CUDA to Nvidia GPUs

- CUDA is designed to be functionally forgiving
- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs

- Threads:
  - each thread is a SIMD vector lane

- Warps:
  - A SIMD instruction acts on a “warp”
  - Warp width is 32 elements: **logical** SIMD width

- Thread blocks:
  - Each thread block is scheduled onto a processor
  - Peak efficiency requires multiple thread blocks per processor
Mapping CUDA to a GPU, continued

- The GPU is very deeply pipelined
  - Throughput machine, trying to hide memory latency
- This means that performance depends on the number of thread blocks which can be allocated on a processor
- Therefore, resource usage costs performance:
  - More registers => Fewer thread blocks
  - More shared memory usage => Fewer thread blocks
  - In general: More resources => less effective parallelism
- It is often worth trying to reduce register count in order to get more thread blocks to fit on the chip
SIMD & Control Flow

- Nvidia GPU hardware handles control flow divergence and reconvergence
  - Write scalar thread code, compiler & hardware auto-vectorize
  - One caveat: \_\_sync\_threads() can’t appear in a divergent path
    - This will cause programs to hang
  - Good performing code will try to keep the execution convergent within a warp
    - Inter-warp divergence is free modulo instruction cache
Memory, Memory, Memory

- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem

- Lots of processors, only one socket
- Memory concerns dominate performance tuning
Memory is SIMD too

- Virtually all processors have SIMD memory subsystems

```
0 1 2 3 4 5 6 7
```

cache line width

- This has two effects:
  - Sparse access wastes bandwidth
    ```
    0 1 2 3 4 5 6 7
    ```
    2 words used, 8 words loaded: ¼ effective bandwidth
  - Unaligned access wastes bandwidth
    ```
    0 1 2 3 4 5 6 7
    ```
    4 words used, 8 words loaded: ½ effective bandwidth
Coalescing

- Current GPUs don’t have cache lines as such, but they do have similar issues with alignment and sparsity.
- Nvidia GPUs have a "coalescer", which examines memory requests dynamically and coalesces them into vectors.
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries
Multidimensional arrays are usually stored as monolithic vectors in memory.

Care should be taken to assure aligned memory accesses for the necessary access pattern.
Sparse Matrix Vector Multiply

- Problem: Sparse Matrix Vector Multiplication
- How should we parallelize the computation?
- How should we represent the matrix?
  - Can we take advantage of any structure in this matrix?
Since this matrix has nonzeros only on diagonals, let’s project the diagonals into vectors.

Sparse representation becomes dense.

Launch a thread per row.

Are we done?

The straightforward diagonal projection is not aligned.
Optimized Diagonal Representation

- Skew the diagonals again
- This ensures that all memory loads from matrix are coalesced
- Don’t forget padding!
Different data access patterns may also require transposing data structures.

- The cost of a transpose on the data structure is often much less than the cost of uncoalesced memory accesses.
Experiences with CUDA

- Image Contour Detection
- Support Vector Machines
Image Contours

- Contours are subjective – they depend on personal perspective
- Surprise: Humans agree (more or less)
- J. Malik’s group has developed a “ground truth” benchmark
gPb Algorithm: Current Leader

- **global Probability of boundary**
- Currently, the most accurate image contour detector
- 3.9 mins per small image (0.15 MP) limits its applicability
  - ~3 billion images on web
  - 10000 computer cluster would take 2 years to find their contours
- How many new images would there be by then?

Maire, Arbelaez, Fowlkes, Malik, CVPR 2008
gPb Computation Outline

Image →
Convert Colorspace →
\[ \text{Lg} \quad \text{Ag} \quad \text{Bg} \quad \text{Combine} \quad \text{Non-max suppression} \]

Textons: K-means →

Intervening Contour →
Generalized Eigensolver →
Oriented Energy Combination →
Combine, Normalize →
Contours
## Time breakdown

<table>
<thead>
<tr>
<th>Computation</th>
<th>Original MATLAB/C++</th>
<th>C + Pthreads (8 threads, 2 sockets)</th>
<th>Damascus (GTX280)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textons</td>
<td>8.6</td>
<td>1.35</td>
<td>0.152</td>
</tr>
<tr>
<td>Gradients</td>
<td>53.8</td>
<td>12.92</td>
<td>0.84</td>
</tr>
<tr>
<td>Intervening Contour</td>
<td>6.3</td>
<td>1.21</td>
<td>0.03</td>
</tr>
<tr>
<td>Eigensolver</td>
<td>151.0</td>
<td>14.29</td>
<td>0.81</td>
</tr>
<tr>
<td>Overall</td>
<td>222 seconds</td>
<td>29.79 seconds</td>
<td>1.8 seconds</td>
</tr>
</tbody>
</table>

**gPb: CVPR 2008**

**Pthreads**
- Textons
- Gradients
- Intervening
- Eigensolver
- Other

**GTX280**
- Textons
- Gradients
- Intervening
- Eigensolver
- Other
Scalability Results

- Scalability examined on Nvidia GPUs from 2 to 30 cores
- Algorithm scales well
- Is memory bandwidth & architecture dependent

Scaling behavior with respect to image size is good
- Bimodal distribution due to eigensolver runtime
- Limited by memory size:
  - 1.8 MP image: 4 GB of memory required
We achieve equivalent accuracy on the BSDS contour detection benchmark

C + Pthreads port done by Yunsup Lee and Andrew Waterman
SVM Training: Quadratic Programming

**Quadratic Program**

\[ F(\alpha) = \max \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \alpha^T Q \alpha \]

s.t. \( 0 \leq \alpha_i \leq C, \quad \forall i \in [1, l] \)
\[ y^T \alpha = 0 \]

\[ Q_{ij} = y_i y_j \Phi(x_i, x_j) \]

**Variables:**

\( \alpha \): Weight for each training point (determines classifier)

**Data:**

\( l \): number of training points

\( y \): Label (+/- 1) for each training point

\( x \): training points

**Example Kernel Functions:**

\( \Phi(x_i, x_j) = x_i \cdot x_j \)

\( \Phi(x_i, x_j; a, r) = \tanh(ax_i \cdot x_j + r) \)

\( \Phi(x_i, x_j; a, r, d) = (ax_j \cdot x_j + r)^d \)

\( \Phi(x_i, x_j; \gamma) = \exp\{-\gamma||x_i - x_j||^2\} \)
SMO Algorithm

- The Sequential Minimal Optimization algorithm (Platt, 1999) is an iterative solution method for the SVM training problem.
- At each iteration, it adjusts only 2 of the variables (chosen by heuristic):
  - The optimization step is then a trivial one dimensional problem:

\[
y_1\alpha_1 + y_2\alpha_2 = k
\]

- Computing full kernel matrix \( Q \) not required.
- Despite name, algorithm can be quite parallel.
- Computation is dominated by KKT optimality condition updates.
Training Results

<table>
<thead>
<tr>
<th>Name</th>
<th>#points</th>
<th>#dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>USPS</td>
<td>7291</td>
<td>256</td>
</tr>
<tr>
<td>Face</td>
<td>6977</td>
<td>381</td>
</tr>
<tr>
<td>Adult</td>
<td>32561</td>
<td>123</td>
</tr>
<tr>
<td>Web</td>
<td>49749</td>
<td>300</td>
</tr>
<tr>
<td>MNIST</td>
<td>60000</td>
<td>784</td>
</tr>
<tr>
<td>Forest</td>
<td>561012</td>
<td>54</td>
</tr>
</tbody>
</table>

Training Time (seconds)

- LibSVM running on Intel Core 2 Duo 2.66 GHz
- Our solver running on Nvidia GeForce 8800GTX
- Gaussian kernel used for all experiments
- 9-35x speedup
SVM Classification

- To classify a point $z$, evaluate:

$$\hat{z} = \left\{ b + \sum_{i=1}^{l} y_i \alpha_i \Phi(x_i, z) \right\}$$

- For standard kernels, SVM Classification involves comparing all support vectors and all test vectors with a dot product.
- We take advantage of the common situation when one has multiple data points to classify simultaneously.
- We cast the dot products as a Matrix-Matrix multiplication, and then use Map Reduce to finish the classification.
Classification Results

Classification Time (seconds)

- CPU optimized version achieves 3-30x speedup
- GPU version achieves an additional 5-24x speedup, for a total of 81-138x speedup
- Results identical to serial version
CUDA Summary

- CUDA is a programming model for manycore processors
- It abstracts SIMD, making it easy to use wide SIMD vectors
- It provides good performance on today’s GPUs
- In the near future, CUDA-like approaches will map well to many processors & GPUs
- CUDA encourages SIMD friendly, highly scalable algorithm design and implementation