Logistics

• Midterm: March 26

• Project proposal presentations: March 19
  – Have to be approved by me by March 14

• Final project presentations: May 3
  – Report due May 6
Overview

• Project ideas
• Parallel Patterns: Convolution
  – Constant memory
  – Cache
• Parallel Patterns: Reduction Trees
Project Proposal

• Problem description
  – What is the computation and why is it important?
  – Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page

• Suitability for GPU acceleration
  – Amdahl’s Law: describe the inherent parallelism. Argue that it is close to 100% of computation.
  – Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
  – Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.

• Intellectual Challenges
  – Generally, what makes this computation worthy of a project?
  – Point to any difficulties you anticipate at present in achieving high speedup
Some Ideas

• k-means
• Perceptron
• Boosting
  – General
  – Face detector (group of 2)
• Mean Shift
• Normal estimation for 3D point clouds
More Ideas

• Look for parallelizable problems in:
  – Image processing
  – Cryptanalysis
  – Graphics
    • GPU Gems
  – Nearest neighbor search

<table>
<thead>
<tr>
<th>Version</th>
<th>Time Elapsed*</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>C# CPU Version w/ GUI and CPU-only solver</td>
<td>~900 seconds</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>C CPU Version Command-line only CPU solver</td>
<td>236.65 seconds</td>
<td>Reference</td>
<td>Reference</td>
</tr>
<tr>
<td>Kernel1 Working solver on GPU</td>
<td>16.07 seconds</td>
<td>14.72x</td>
<td>14.73x</td>
</tr>
<tr>
<td>Kernel3 Added reduction kernel</td>
<td>9.18 seconds</td>
<td>1.75x</td>
<td>25.78x</td>
</tr>
<tr>
<td>Kernel4 Changed data structure to array instead of AoS</td>
<td>8.47 seconds</td>
<td>1.08x</td>
<td>27.94x</td>
</tr>
<tr>
<td>Kernel5 Simple caching w/ shared memory</td>
<td>7.25 seconds</td>
<td>1.17x</td>
<td>32.64x</td>
</tr>
</tbody>
</table>

GPU: Shared Memory
512 Zombies
Average FPS: 45.9
Even More…

• Particle simulations
• Financial analysis
• MCMC
• Games/puzzles
  – Mastermind example
k-means

• See also http://www.cs.stevens.edu/~mordohai/classes/cs559_f10.html
  – Notes 13
SSE Criterion Function

Let $n_i$ be the number of samples, then the mean is:

$$\mu_i = \frac{1}{n_i} \sum_{x \in D_i} x$$

The sum-of-squared errors criterion function (to minimize) is:

$$J_{SSE} = \sum_{i=1}^{c} \sum_{x \in D_i} ||x - \mu_i||^2$$

Note that the number of clusters, $c$, is fixed.
K-means Clustering

1. Initialize
   - Pick \( k \) cluster centers arbitrarily
   - Assign each example to closest center

2. Compute sample means for each cluster

3. Reassign all samples to the closest mean

4. If clusters changed at step 3, go to step 2
K-means Clustering

Consider steps 2 and 3 of the algorithm

2. compute sample means for each cluster

3. reassign all samples to the closest mean

If we represent clusters by their old means, the error has decreased

$$J_{SSE} = \sum_{i=1}^{k} \sum_{x \in D_i} \| x - \mu_i \|^2$$
K-means Clustering

• We can prove that by repeating steps 2 and 3, the objective function is reduced.
• Thus k-means converges after a finite number of iterations of steps 2 and 3.
• However k-means is not guaranteed to find a global minimum.

2-means gets stuck here

global minimum of $J_{SSE}$
K-means Clustering

• Finding the optimum of $J_{SSE}$ is NP-hard
• In practice, k-means clustering usually performs well
• To avoid local minima, in practice we randomly re-initialize it several times
Perceptron

• See also
http://www.cs.stevens.edu/~mordohai/classes/cs559_f10.html
  – Notes 10
The Problem

• Assume we have 2 classes
  – Samples: \( y_1, \ldots, y_n \), some in class 1, some in class 2
• Use samples to determine weights \( a \) in the discriminant function \( g(y) = a^t y \)
• We want to minimize the training error (the number of misclassified samples \( y_1, \ldots, y_n \))
  
  If: 
  
  \[
  g(y_i) > 0 \implies y_i \text{ classified as } c_1 \\
  g(y_i) < 0 \implies y_i \text{ classified as } c_2
  \]

• Thus training error is 0 if 
  \[
  \begin{cases} 
  g(y_i) > 0 & \forall y_i \in c_1 \\
  g(y_i) < 0 & \forall y_i \in c_2 
  \end{cases}
  \]
“Normalization”

• Thus training error is 0 if:
  \[
  \begin{align*}
  a^t y_i > 0 & \quad \forall y_i \in c_1 \\
  a^t y_i < 0 & \quad \forall y_i \in c_2
  \end{align*}
  \]

• Equivalently, training error is 0 if:
  \[
  \begin{align*}
  a^t y_i > 0 & \quad \forall y_i \in c_1 \\
  a^t (-y_i) > 0 & \quad \forall y_i \in c_2
  \end{align*}
  \]

• This suggests “normalization” (a.k.a. reflection):
  1. Replace all examples from class 2 by:
     \[
     y_i \rightarrow -y_i \quad \forall y_i \in c_2
     \]
  2. Seek weight vector \( a \) such that
     \[
     a^t y_i > 0 \quad \forall y_i
     \]
     – If such \( a \) exists, it is called a separating or solution vector
     – Original samples \( x_1, \ldots, x_n \) can indeed be separated by a line
Normalization

before normalization

• Seek a hyperplane that separates patterns from different categories

after “normalization”

• Seek hyperplane that puts normalized patterns on the same (positive) side
Perceptron Criterion Function

\[ J_p(a) = \sum_{y \in Y_M} (-a^t y) \]

- If \( y \) is misclassified, \( a^t y < 0 \)
- Thus \( J_p(a) > 0 \)
- \( J_p(a) \) is \( \|a\| \) times the sum of distances of misclassified examples to decision boundary
- \( J_p(a) \) is piecewise linear and thus suitable for gradient descent
Perceptron Batch Rule

\[ J_p(a) = \sum_{y \in Y_M} (-a^t y) \]

- Gradient of \( J_p(a) \) is: \( \nabla J_p(a) = \sum_{y \in Y_M} (-y) \)
  - \( Y_M \) are samples misclassified by \( a^{(k)} \)
  - It is not possible to solve \( \nabla J_p(a) = 0 \) analytically because of \( Y_M \)

- Update rule for gradient descent: \( x^{(k+1)} = x^{(k)} - \eta^{(k)} \nabla J(x) \)

- Thus the **gradient decent batch update rule** for \( J_p(a) \) is:
  \[ a^{(k+1)} = a^{(k)} + \eta^{(k)} \sum_{y \in Y_M} y \]

- It is called batch rule because it is based on all misclassified examples
Boosting

• See also
  http://www.cs.stevens.edu/~mordohai/classes/cs559_f10.html
  – Notes 12
Boosting

• Idea: given a set of weak learners, run them multiple times on (reweighted) training data, then let learned classifiers vote

• At each iteration $t$:
  – Weight each training example by how incorrectly it was classified
  – Learn a hypothesis - $h_t$
  – Choose a strength for this hypothesis - $\alpha_t$

• Final classifier: weighted combination of weak learners
Learning from Weighted Data

• Sometimes not all data points are equal
  – Some data points are more equal than others

• Consider a weighted dataset
  – $D(i)$ - weight of $i^{th}$ training example $(x_i, y_i)$
  – Interpretations:
    • $i^{th}$ training example counts as $D(i)$ examples
    • If I were to “resample” data, I would get more samples of “heavier” data points

• Now, in all calculations the $i^{th}$ training example counts as $D(i)$ “examples”
Definition of Boosting

• Given training set \((x_1,y_1),\ldots,(x_m,y_m)\)
• \(y_i \in \{-1,+1\}\) correct label of instance \(x_i \in X\)
• For \(t=1,\ldots,T\)
  – construct distribution \(D_t\) on \(\{1,\ldots,m\}\)
  – find weak hypothesis \(h_t: X \rightarrow \{-1,+1\}\)
    with small error \(\epsilon_t\) on \(D_t\)
  \[
  \epsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i]
  \]
• Output final hypothesis \(H_{\text{final}}\)
AdaBoost

• Constructing $D_t$
  – $D_1=1/m$
  – Given $D_t$ and $h_t$: 
    \[
    D_{t+1}(i) = \frac{D_t(i)}{Z_t} \cdot \begin{cases} 
    e^{-\alpha_t} & \text{if } y_i = h_t(x_i) \\
    e^{\alpha_t} & \text{if } y_i \neq h_t(x_i)
    \end{cases}
    \]
    where $Z_t$ is a normalization constant

• Final hypothesis:
  \[
  H_{\text{final}}(x) = \text{sign} \left( \sum_t \alpha_t h_t(x) \right)
  \]

\[
\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right) > 0
\]
Face Detection

• I see this as a two person project
  – One implements boosting as before
  – One implements the face-specific parts

• See also
  http://www.cs.stevens.edu/~mordohai/classes/cs559_f10.html
  – Notes 12
Classifier is Learned from Labeled Data

• Training Data
  – 5000 faces
    • All frontal
  – 10^8 non faces
  – Faces are normalized
    • Scale, translation

• Many variations
  – Across individuals
  – Illumination
  – Pose (rotation both in plane and out)
Boosted Face Detection: Image Features

“Rectangle filters”

Similar to Haar wavelets

\[ h_t(x_i) = \begin{cases} \alpha_t & \text{if } f_t(x_i) > \theta_t \\ \beta_t & \text{otherwise} \end{cases} \]

\[ C(x) = \theta \left( \sum_t h_t(x) + b \right) \]

\[ 60,000 \times 100 = 6,000,000 \]

Unique Binary Features
Feature Selection

• For each round of boosting:
  – Evaluate each rectangle filter on each example
  – Sort examples by filter values
  – Select best threshold for each filter
  – Select best filter/threshold (= Feature)
  – Reweight examples
Feature Localization

• Learned features reflect the task
Output of Face Detector on Test Images
Mean Shift

• See also
  http://www.cs.stevens.edu/~mordohai/classes/cs559_f10.html
  – Notes 13
Intuitive Description

Objective: Find the densest region
Distribution of identical billiard balls
Computing The Mean Shift

Simple Mean Shift procedure:
- Compute mean shift vector

\[
m(x) = \left[ \frac{\sum_{i=1}^{n} x_i g \left( \frac{\|x - x_i\|^2}{h} \right)}{\sum_{i=1}^{n} g \left( \frac{\|x - x_i\|^2}{h} \right)} \right] - x
\]

- Translate the Kernel window by \( m(x) \)
Segmentation

Example
Segmentation Example
Normal Estimation for 3D Point Clouds
Scatter Matrix

• Compute the symmetric positive definite covariance matrix from N neighbors of a 3-D point
  \[
  \{X_i\} = \{(x_i, y_i, z_i)^T\}
  \]
  \[
  \frac{1}{N} \sum_{i=1}^{N} (X_i - \bar{X})(X_i - \bar{X})^T
  \]

• Then, the eigenvector that corresponds to the smallest eigenvalue is the normal to the surface at each point
  – If each point belonged to a smooth surface
Classification

• Points can be classified according to eigenvalues into surfaces, foliage, ground plane etc.
  – Images from Lalonde et al. 2006
Markov Chain Monte Carlo

- Randomized algorithms based on sampling from probability distributions to generate sequences of observations
- Applications
  - Approximate integration
  - Optimization of energy/cost functions in very large search spaces
  - Risk assessment in finance
Sample Proposal

3 Intellectual Challenges

The main challenge is going to be how to partition the work. As mentioned above, the overall algorithm is finding the minimum across a set. However, there is also an internal operation that involves a maximum operation. In terms of mapping this to CUDA, there are going to need to be some testing to determine how heavy a thread should be. For example, one configuration would be to make every thread calculate the worst-case scenario for one element in the set. Another configuration would be to calculate that maximum on the block-level, making the threads perform much less work.

The main obstacle for performance is going to be synchronization. Especially in a case where every block produces one out of 32,768 results that need to be minimized, doing atomic operations to a global memory location is bound to have consequences. A lot of parameterization is going to be necessary so that different combinations of strategies can be fully tested.

The Problem Description above focused on Knuth’s algorithm for solving Mastermind puzzles. There have been a few papers published since then which propose better solutions, such as the often cited 1993 paper by Koyama and Lai\textsuperscript{2} and a more recent 2005 paper by Kooi\textsuperscript{3}. In the course of the actual project, I plan to investigate those other algorithms and if they are equally parallel-capable and seem to perform better, I will switch the algorithm.

I believe this project has a great chance to show how CUDA can be used to improve the performance of existing algorithms, increasing their domain of effectiveness.
2D Convolution, Constant Memory and Constant Caching
2D Convolution - Inside Cells

© David Kirk/NVIDIA and Wen-mei W. Hwu
University of Illinois, 2007-2011
2D Convolution - Ghost Cells

The figure illustrates the concept of ghost cells in 2D convolution. ghost cells

© David Kirk/NVIDIA and Wen-mei W. Hwu
University of Illinois, 2007-2011
Access Pattern for M

• M is referred to as mask (a.k.a. kernel, filter, etc.)
  – Elements of M are called mask (kernel, filter) coefficients

• Calculation of all output P elements need M

• M is not changed during kernel

• Bonus - M elements are accessed in the same order when calculating all P elements

• M is a good candidate for Constant Memory
Programmer View of CUDA Memories (Review)

Each thread can:
- Read/write per-thread registers (~1 cycle)
- Read/write per-block shared memory (~5 cycles)
- Read/write per-grid global memory (~500 cycles)
- Read/only per-grid constant memory (~5 cycles with caching)
Memory Hierarchies

• If every time we needed a piece of data, we had to go to main memory to get it, computers would take a lot longer to do anything

• On today’s processors, main memory accesses take hundreds of cycles

• One solution: Caches
Cache

• In order to keep cache fast, it needs to be small, so we cannot fit the entire data set in it
Cache

• Cache is unit of volatile memory storage

• A cache is an “array” of cache lines

• Cache line can usually hold data from several consecutive memory addresses

• When data is requested from memory, an entire cache line is loaded into the cache, in an attempt to reduce main memory requests
Caches

Some definitions:

– Spatial locality: is when the data elements stored in consecutive memory locations are access consecutively

– Temporal locality: is when the same data element is access multiple times in short period of time

• Both spatial locality and temporal locality improve the performance of caches
Scratchpad vs. Cache

• Scratchpad (shared memory in CUDA) is another type of temporary storage used to relieve main memory contention.

• In terms of distance from the processor, scratchpad is similar to L1 cache.

• Unlike cache, scratchpad does not necessarily hold a copy of data that is in main memory

• It requires explicit data transfer instructions, whereas cache does not
Cache Coherence Protocol

- A mechanism for caches to propagate updates by their local processor to other caches (processors)
CPU and GPU have different caching philosophy

• CPU L1 caches are usually coherent
  – L1 is also replicated for each core
  – Even data that will be changed can be cached in L1
  – Updates to local cache copy invalidates (or less commonly updates) copies in other caches
  – Expensive in terms of hardware and disruption of services (cleaning bathrooms at airports..)

• GPU L1 caches are usually incoherent
  – Avoid caching data that will be modified
GPU Cache Coherence

• Current CUDA implementation:
  – Provide coherence by disabling L1 cache after writes
  – There is room for improvement

• Custom implementations
  – Temporal coherence: invalidates cache using synchronized counters without message passing
  – Stall writes to cache blocks until they have been invalidated
How to Use Constant Memory

• Host code allocates, initializes variables the same way as any other variables that need to be copied to the device

• Use `cudaMemcpyToSymbol(dest, src, size)` to copy the variable into the device memory
  – Declare `__const__ float M[MAX_WIDTH]` first

• This copy function tells the device that the variable will not be modified by the kernel and can be safely cached
More on Constant Caching

- Each SM has its own L1 cache
  - Low latency, high bandwidth access by all threads
- However, there is no way for threads in one SM to update the L1 cache in other SMs
  - No L1 cache coherence

This is not a problem if a variable is NOT modified by a kernel.
#define KERNEL_SIZE 5

// Matrix Structure declaration
typedef struct {
    unsigned int width;
    unsigned int height;
    unsigned int pitch;
    float* elements;
} Matrix;
AllocateMatrix

// Allocate a device matrix of dimensions height*width
// If init == 0, initialize to all zeroes.
// If init == 1, perform random initialization.
// If init == 2, initialize matrix parameters, but do
// not allocate memory
Matrix AllocateMatrix(int height, int width, int init)
{
    Matrix M;
    M.width = M.pitch = width;
    M.height = height;
    int size = M.width * M.height;
    M.elements = NULL;
AllocateMatrix() (Cont.)

// don't allocate memory on option 2
if(init == 2)
    return M;
M.elements = (float*) malloc(size*sizeof(float));
for(unsigned int i = 0; i < M.height * M.width; i++)
{
    M.elements[i] = (init == 0) ? (0.0f) : (rand() / (float)RAND_MAX);
    if(rand() % 2)
        M.elements[i] = - M.elements[i]
}
return M;
// global variable, outside any function
__constant__ float Mc[KERNEL_SIZE][KERNEL_SIZE];

... allocate N, P, initialize N elements, copy N to Nd Matrix  M;
M = AllocateMatrix(KERNEL_SIZE, KERNEL_SIZE, 1);
// initialize M elements
... cudaMemcpyToSymbol(Mc, M.elements,
                    KERNEL_SIZE*KERNEL_SIZE*sizeof(float));

ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
Intermission

• We will revisit this part next time
Reduction Trees
Partition and Summarize

• A commonly used strategy for processing large input data sets
  – There is no required order of processing elements in a data set (associative and commutative)
  – Partition the data set into smaller chunks
  – Have each thread to process a chunk
  – Use a reduction tree to summarize the results from each chunk into the final answer

• We will focus on the reduction tree step for now

• Google and Hadoop MapReduce frameworks are examples of this pattern
Reduction enables other techniques

• Reduction is also needed to clean up after some commonly used parallelizing transformations

• Privatization
  – Multiple threads write into an output location
  – Replicate the output location so that each thread has a private output location
  – Use a reduction tree to combine the values of private locations into the original output location
What is a reduction computation

- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
- Often with user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
A sequential reduction algorithm performs N operations - O(N)

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- Scan through the input and perform the reduction operation between the result value and the current input value
A parallel reduction tree algorithm performs N-1 Operations in log(N) steps
A tournament is a reduction tree with “max” operation
A Quick Analysis

- For N input values, the reduction tree performs
  - \((1/2)N + (1/4)N + (1/8)N + \ldots + (1/N) = (1 - (1/N))N = N-1\) operations
  - In Log \((N)\) steps - 1,000,000 input values take 20 steps
    - Assuming that we have enough execution resources
  - Average Parallelism \((N-1)/\text{Log}(N))\)
    - For \(N = 1,000,000\), average parallelism is 50,000
    - However, peak resource requirement is 500,000!

- This is a work-efficient parallel algorithm
  - The amount of work done is comparable to sequential
  - Many parallel algorithms are not work efficient
A Sum Reduction Example

• Parallel implementation:
  – Recursively halve # of threads, add two values per thread in each step
  – Takes log(n) steps for n elements, requires n/2 threads

• Assume an in-place reduction using shared memory
  – The original vector is in device global memory
  – The shared memory is used to hold a partial sum vector
  – Each step brings the partial sum vector closer to the sum
  – The final sum will be in element 0
  – Reduces global memory traffic due to partial sum values
### Vector Reduction with Branch Divergence

<table>
<thead>
<tr>
<th>Data</th>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
<th>Thread 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0+1</td>
<td>2+3</td>
<td>4+5</td>
<td>6+7</td>
<td>8+9</td>
<td>10+11</td>
</tr>
<tr>
<td>1</td>
<td>0..3</td>
<td></td>
<td>4..7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0..7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8..15</td>
<td></td>
</tr>
</tbody>
</table>

Steps:
- **Thread 0**: 0, 0+1, 0..3, 0..7
- **Thread 1**: 1, 2+3, 2+3, 4..7
- **Thread 2**: 2, 3, 4+5, 4..7
- **Thread 3**: 3, 4, 5, 6+7
- **Thread 4**: 5, 6, 7, 8+9
- **Thread 5**: 6, 7, 8, 10+11

**Partial Sum elements**

<table>
<thead>
<tr>
<th>Partial Sum elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0+1, 2+3, 4+5, 6+7, 8+9, 10+11</td>
</tr>
</tbody>
</table>

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
A Sum Example

Thread 0  Thread 1  Thread 2  Thread 3

Data

3  4  11  25
1  7  7  5
0  0  14  0
4  1  9  3

Active Partial Sum elements
Simple Thread Index to Data Mapping

- Each thread is responsible of an even-index location of the partial sum vector
  - One input is the location of responsibility

- After each step, half of the threads are no longer needed

- In each step, one of the inputs comes from an increasing distance away
A Simple Thread Block Design

• Each thread block takes 2* blockDim input elements
• Each thread loads 2 elements into shared memory

```c
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] =
    input[start+ blockDim.x+t];
```
The Reduction Steps

for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t]+= partialSum[2*t+stride];
}
Barrier Synchronization

• `syncthreads()` are needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

• Why do we not need another `syncthreads()` at the end of the reduction loop?
Back to the Global Picture

• Thread 0 in each thread block writes the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x

• There can be a large number of such sums if the original vector is very large
  – The host code may iterate and launch another kernel

• If there are only a small number of sums, the host can simply transfer the data back and add them together
Some Observations

• In each iteration, two control flow paths will be sequentially traversed for each warp
  – Threads that perform addition and threads that do not
  – Threads that do not perform addition still consume execution resources

• No more than half of threads will be executing after the first step
  – All odd index threads are disabled after first step
  – After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence.
  • This can go on for a while, up to 5 more steps (1024/32=16= 2^5), where each active warp only has one productive thread until all warps in a block retire
Thread Index Usage Matters

• In some algorithms, one can shift the index usage to improve the divergence behavior
  – Commutative and associative operators

• Example - given an array of values, “reduce” them to a single value in parallel
  – Sum reduction: sum of all values in the array
  – Max reduction: maximum of all values in the array
  – ...

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
A Better Strategy

• Always compact the partial sums into the first locations in the partialSum[] array

• Keep the active threads consecutive
An Example of 16 threads

Thread 0  Thread 1  Thread 2

Thread 14  Thread 15

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al, University of Illinois, 2007-2011
A Better Reduction Kernel

for (unsigned int stride = blockDim.x/2; stride >= 1; stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
A Quick Analysis

• For a 1024 thread block
  – No divergence in the first 5 steps
  – 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
  – The final 5 steps will still have divergence