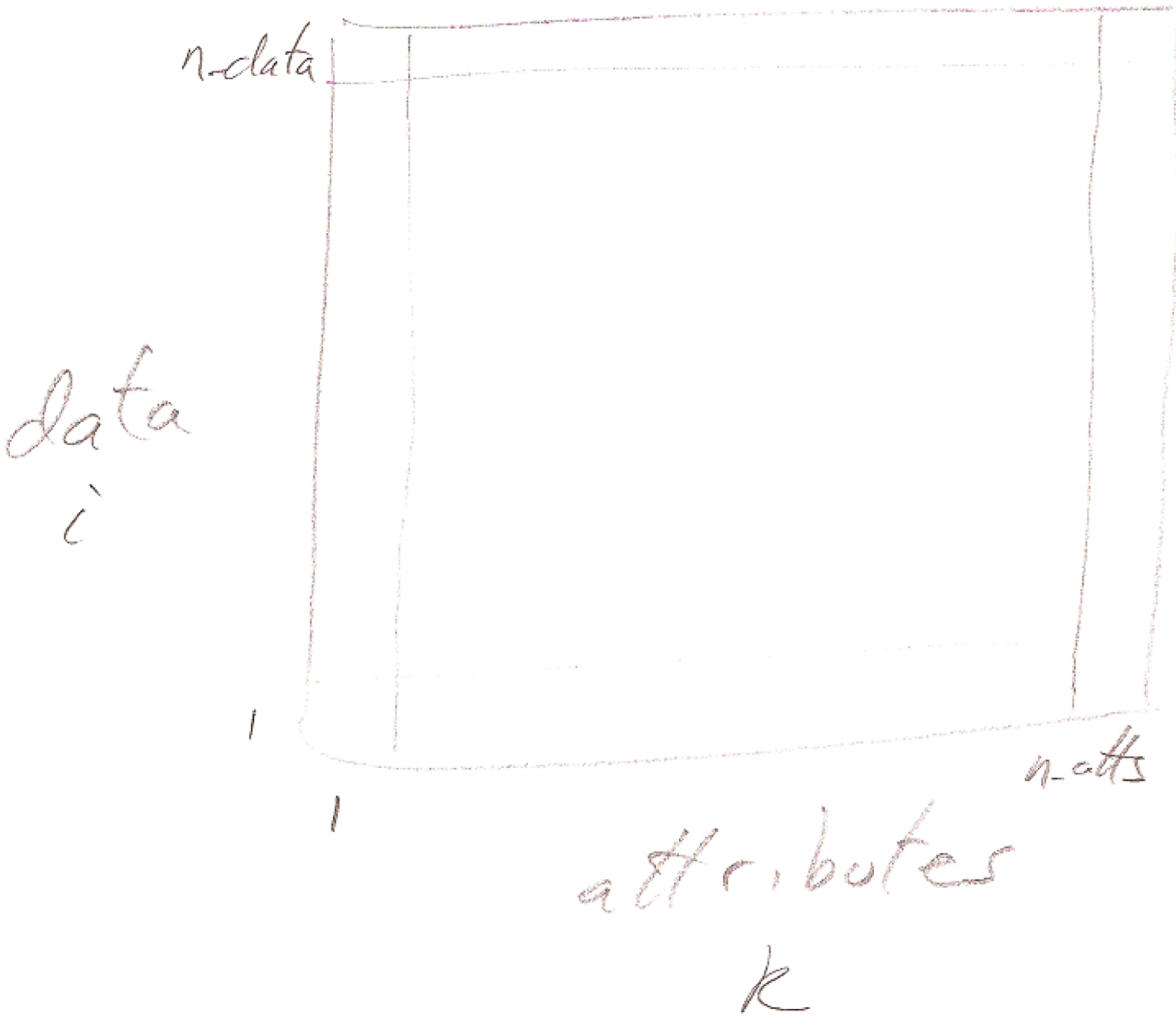
Parallel AutoClass

Kevin R Keane

CSE 633

Spring 2009

Given Observed Data



Estimate Class Distribution Parameters

–

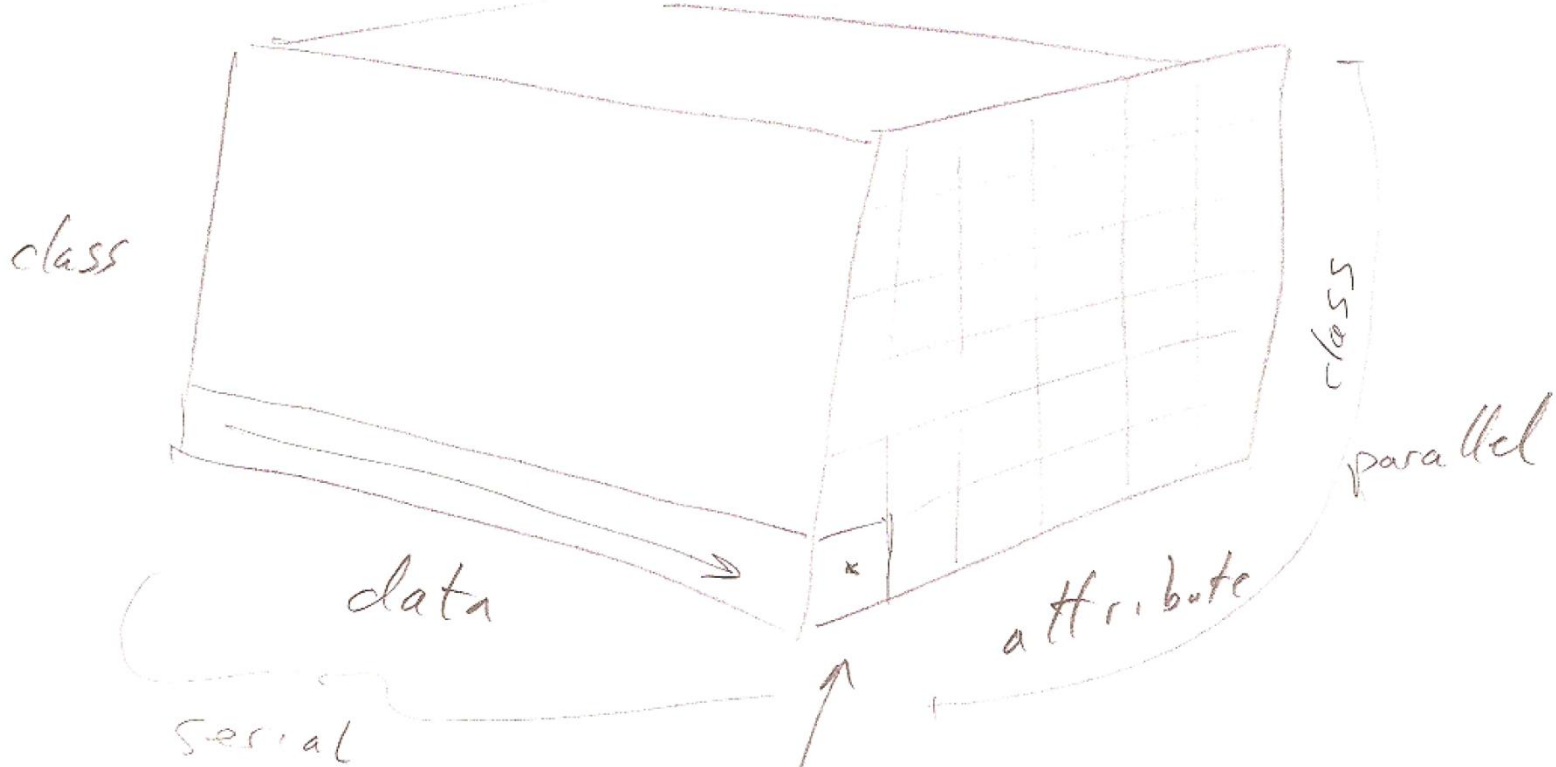
$$\mu_{j,k} = \frac{\sum_{i=1}^{N} w_{i,j} x_{i,k}}{\sum_{i=1}^{N} w_{i,j}}$$

$$\sigma_{j,k}^2 = \frac{\sum_{i=1}^{N} w_{i,j} (x_{i,k} - \mu_{j,k})^2}{\sum_{i=1}^{N} w_{i,j}}$$

Original *update_parameters()*

```
void update_parameters( clsf_DS clsf){
   // ...
   for (n_cl=0; n_cl<n_classes; n_cl++) {
      cl = classes[n_cl];
      update_params_fn(cl, n_classes, database, collect);
   }

   // ...
  }

void update_params_fn( class_DS class, int n_classes,
  database_DS data_base, int collect){
  int i, n_atts;
  tparm_DS tparm;

  class->pi_j = (class->w_j + (1.0 / n_classes)) / (data_base->n_data + 1.0);
  class->log_pi_j = (float) safe_log((double) class->pi_j);
  n_atts = data_base->n_atts;
  for (i=0; i<n_atts; i++) {
    tparm=class->tparms[i];

  // ...

  }
```

Parameter Estimation



Launch thread for each class/attribute combo.

Obtain

$$\sum w_{ij} x_{ik} \rightarrow \mu_{jk}$$

$$\sum w_{ij} (x_{ik} - \mu_{jk})^2 \rightarrow \sigma^2_{jk}$$

New *update_parameters()*

```
int bx=16,by=16,gx,gy;
cudaError_t err;
gx = (n_classes -1)/ bx + 1;
gy = (n_atts -1)/by + 1;
dim3 dimBlock(bx,by);
dim3 dimGrid(gx,gy);

update_params_fn<<< dimGrid, dimBlock >>>
  (d_classes, n_classes, d_database, collect);
```

New - *inside the former loops*

```
__global__ void update_params_fn( class_DS *classes, int n_classes,
                database_DS data_base, int collect)
{
  int i, j;
  tparm_DS tparm;

  int ix = blockDim.x * blockIdx.x + threadIdx.x;
  int iy = blockDim.y * blockIdx.y + threadIdx.y;

  if(!(ix < n_classes && iy < classes[0]->model->n_terms))return;

  j = ix; // class
  i = iy; // attribute
  class_DS xclass = classes[j];

  // this looks like trouble - lots of threads updating common pi_j / log_pi_j
  if(iy==0){
        xclass->pi_j = (xclass->w_j + (1.0 / n_classes)) / (data_base->n_data + 1.0);
        xclass->log_pi_j = (float) safe_log((double) xclass->pi_j);
  }
```

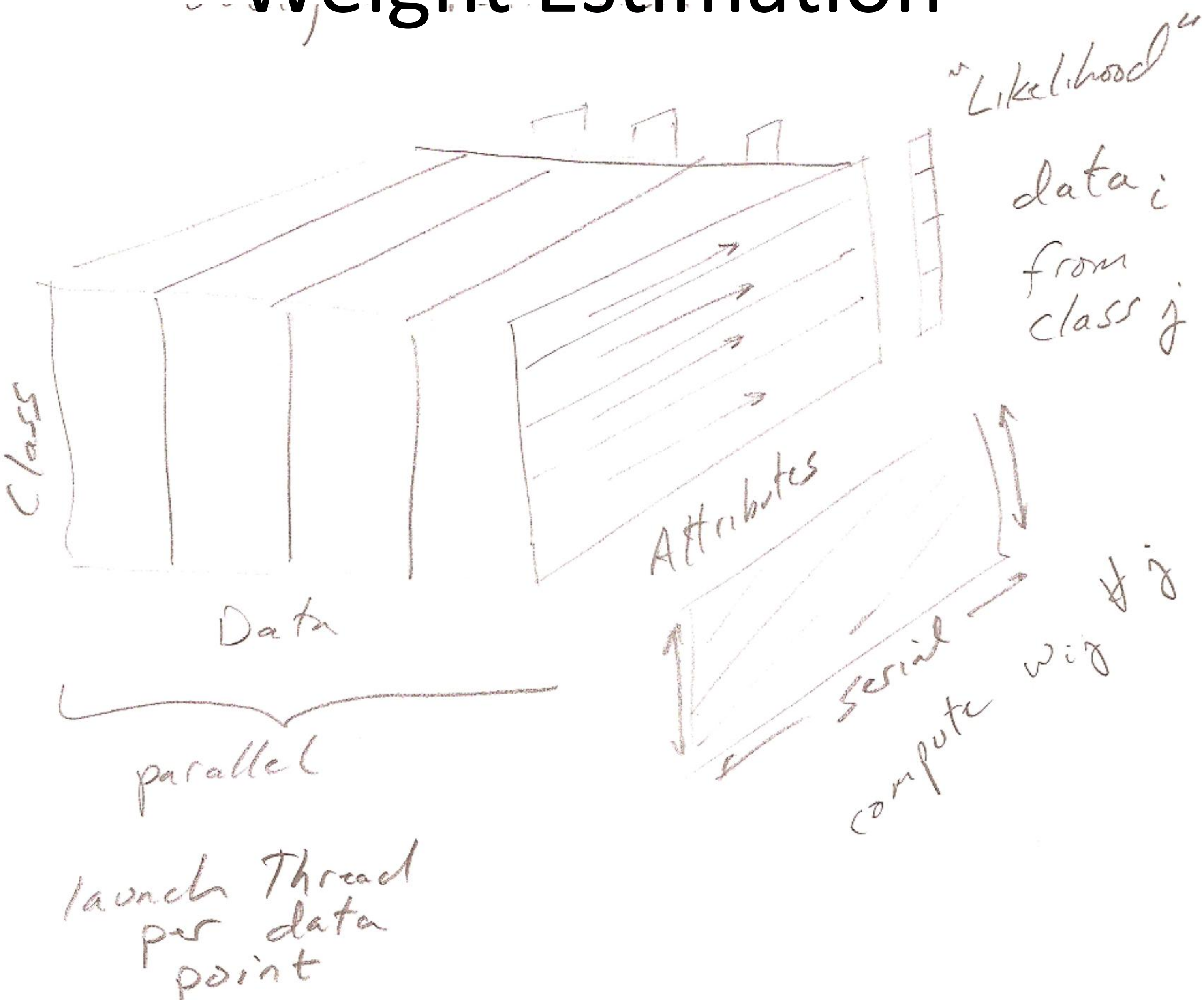
```
// allocate storage for classes on device
class_DS d_classes_buffer, *d_classes;
cudaMalloc((void**)&d_classes_buffer, n_classes*sizeof(d_classes_buffer[0]));
cudaMalloc((void**)&d_classes, n_classes*sizeof(d_classes[0]));
for(i=0;i<n_classes;i++){
        d_classes[i] = d_classes_buffer + i;
        cudaMemcpy(d_classes[i],classes[i],
          sizeof(d_classes_buffer[0]), cudaMemcpyHostToDevice);
}
// allocate storage for db on device - MOVE THIS - needed only once
float *d_data_buffer, **d_data;
cudaMalloc((void**)&d_data_buffer, n_data*n_atts*sizeof(d_data_buffer[0]));
cudaMalloc((void**)&d_data, n_data*sizeof(d_data[0]));
for(i=0;i<n_data;i++){
        d_data[i] = d_data_buffer + i*n_atts;
        cudaMemcpy(d_data[i],data[i],
          sizeof(d_data_buffer[0])*n_atts, cudaMemcpyHostToDevice);
}
database_DS d_database;
cudaMalloc((void**)&d_database, sizeof(d_database[0]));
d_database->n_data = n_data;
d_database->n_atts = n_atts;
d_database->data = d_data;

// allocate space for wts
float *d_wts_buffer, **d_wts;
cudaMalloc((void**)&d_wts_buffer, n_data*n_classes*sizeof(d_wts_buffer[0]));
cudaMalloc((void**)&d_wts, n_classes*sizeof(d_wts[0]));
for(i=0;i<n_classes;i++){
        d_wts[i] = d_wts_buffer + i*n_data;
        d_classes[i]->wts = d_wts[i];
        cudaMemcpy(d_classes[i]->wts,classes[i]->wts,
          sizeof(d_wts_buffer[0])*n_data, cudaMemcpyHostToDevice);
}


int bx=16,by=16,gx,gy;
cudaError_t err;
gx = (n_classes -1)/ bx + 1;
gy = (n_atts -1)/by + 1;
dim3 dimBlock(bx,by);
dim3 dimGrid(gx,gy);

update_params_fn<<< dimGrid, dimBlock >>>
  (d_classes, n_classes, d_database, collect);
```

Estimate Class Membership Weights

$$p(x_i | i \in C_j, \theta_j) = \prod_{k=1}^{K} \pi_j p(x_{i,k} | i \in C_j, \theta_{j,k})$$

$$w_i = \frac{\pi_j p(x_i | i \in C_j, \theta_j)}{\sum_{j=1}^{J} \pi_j p(x_i | i \in C_j, \theta_j)}$$

Weight Estimation



So much code … so little time
the speed bumps

- It took *8.5 seconds* to push hello world to the device.
  - After that, *hellos* arrived 12/millisecond.
- Apparently *try* and *class* are keywords in C++.
- The NVIDIA compiler, *nvcc* compiles the *.cu files in C++ mode. So, for the C portion of the program to link properly, you need to wrap headers:

```
extern "C" {
        int myCoolDemo(int argc,char **argv);
}
```

So much code … so little time *the speed bumps*

- 3-D dimensions have modest limits in the third dimension.
- Maximum dimensions for a *block*
  – 512, 512, and 64 for the x, y, and z
- A *grid* is at most two dimensional
  – Blocks can be arrayed in two dimensional grids to large size.
  – Third dimension is limited to 64. So, need to pick a dimension to be relegated to the 64 count limit.
- Perhaps that's ok for the class dimension, and then again maybe not.

So much code … so little time
the speed bumps

- There is not an intrinsic *all-reduce* function in CUDA
  - Required along the summation dimensions for parallel code
- Implementation
  - thread per each class / attribute combination for parameter estimation
  - Thread per datum for weight estimation
  - Summation and product dimensions handled serially
- Not so bad …
  - *#classes x #attributes* frequently 50-100 or more
  - Number of observations typically large, 1000 – 100,000+
- Significant parallel speedup still possible

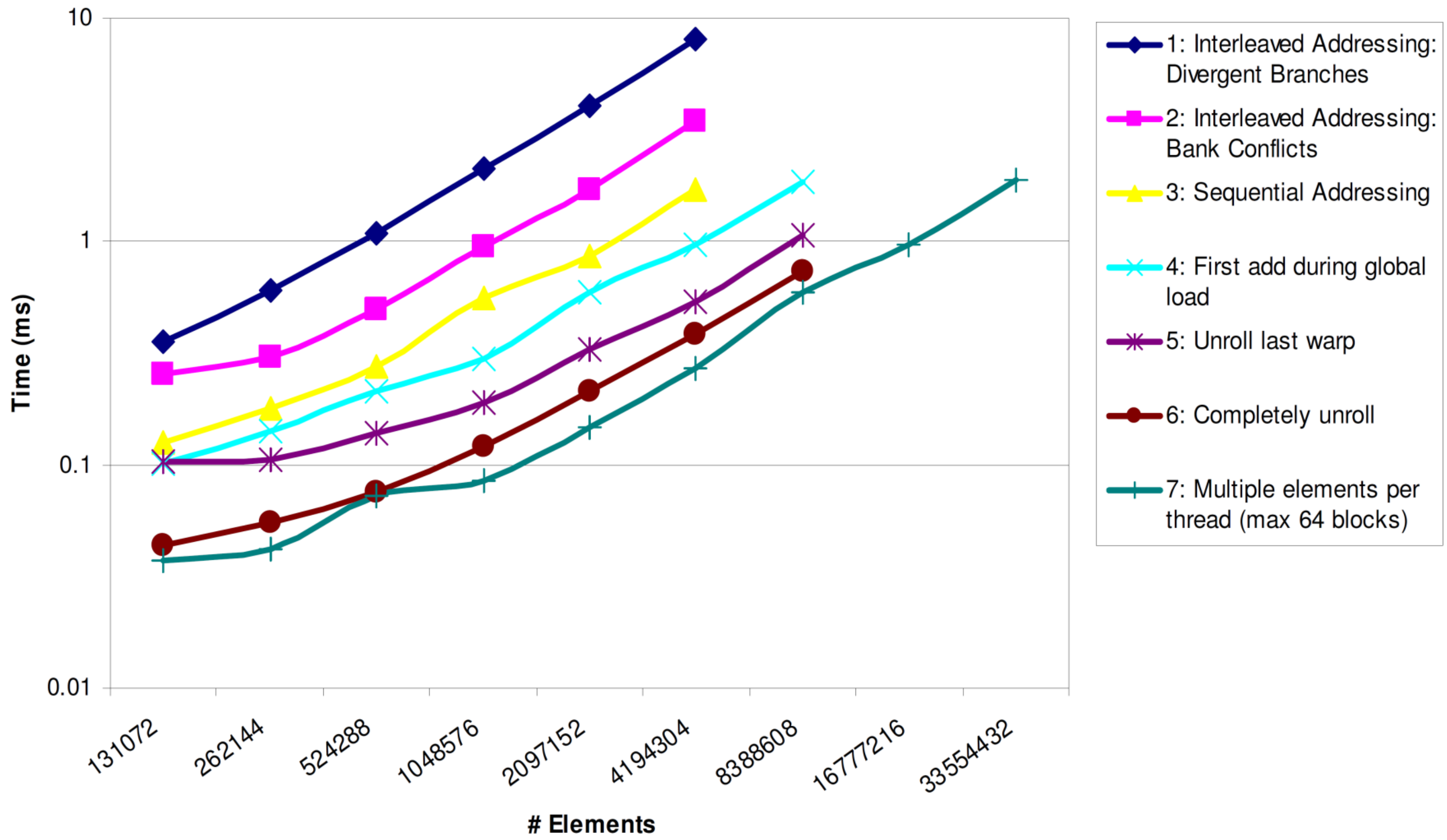
Optimizing reduction code provides further opportunity for speedup

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 32M elements: 73 GB/s!

Source: developer.download.nvidia.com

Optimizing reduction code provides further opportunity for speedup



Source: developer.download.nvidia.com

Lessons learned …

- Porting legacy code is not pretty …
  – Loops spread widely across functions
  – Data structures not compactly allocated
    - Copy & ship is a pain
- Probably best to design software directly to take advantage of architecture
- On the other hand –
  – Software per architecture is probably a bad idea
  – Coding is frequently the bottleneck