Introduction to CUDA

João Barbosa

Outline

- Thinking in terms of performance
  - Parallel reduction
- Some frameworks/libraries
  - Thrust
  - cudaBlas/cudaFFT
- CUDA debugging and Profilling

Parallel Reduction

Based on: Optimizing Parallel Reduction in CUDA

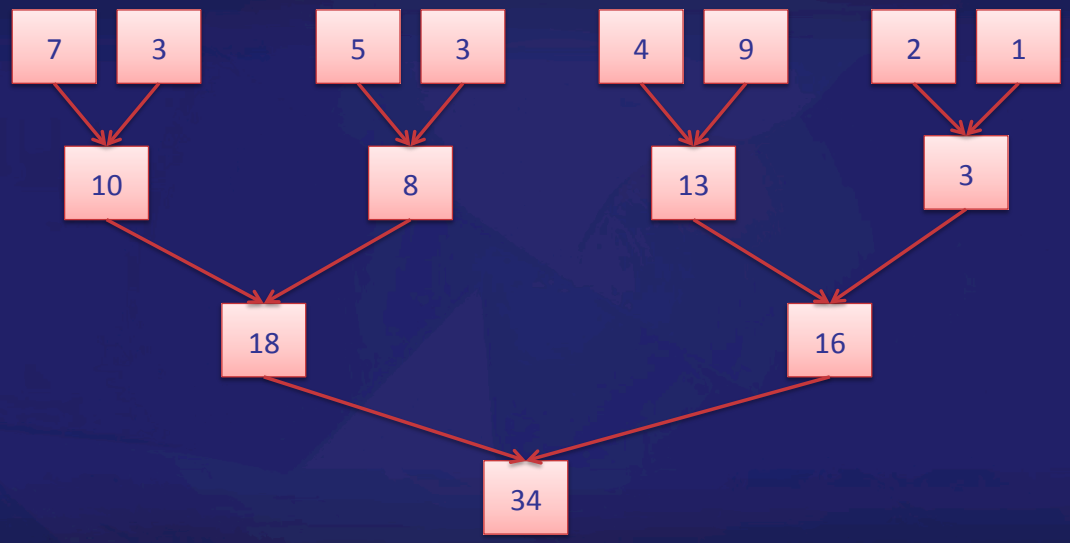
Mark Harris

NVIDIA Developer Technology

Parallel Reduction

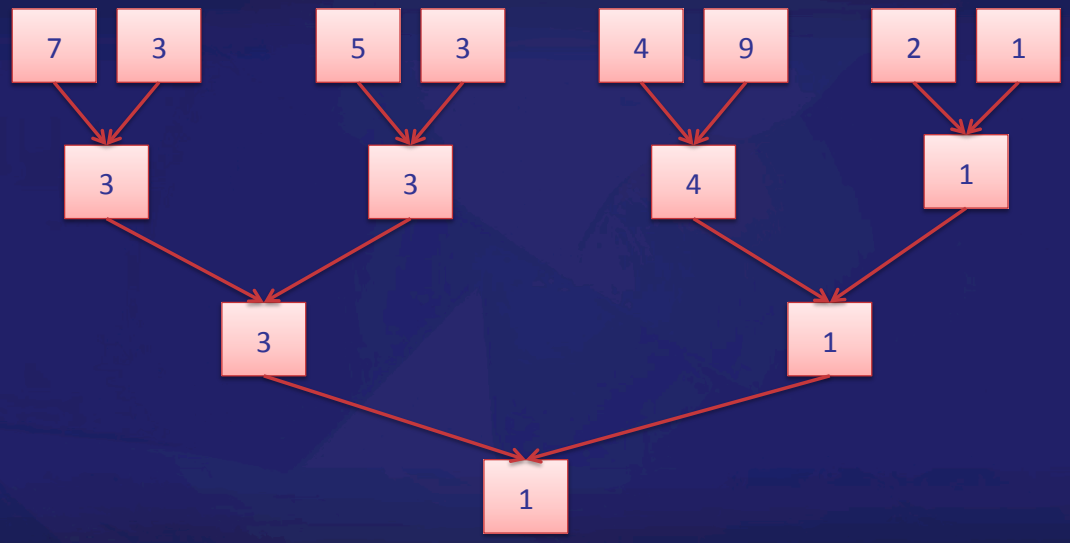
- Important data parallel primitive
- Easy to implement in CUDA
  - Hard to get performance
- We will use it to demonstrate several key performance optimizations

Parallel reduction



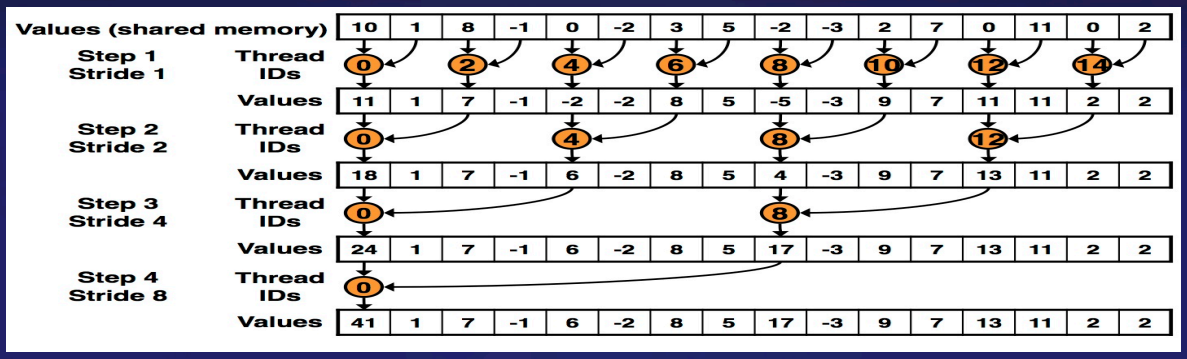
Operator: SUM

Parallel reduction



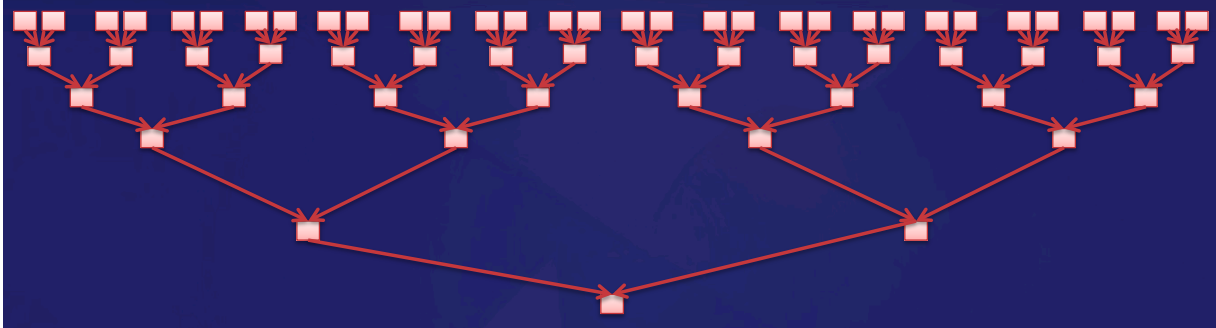
Operator: MIN

Parallel reduction



- Each thread block uses a pairwise approach
- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

Parallel Reduction

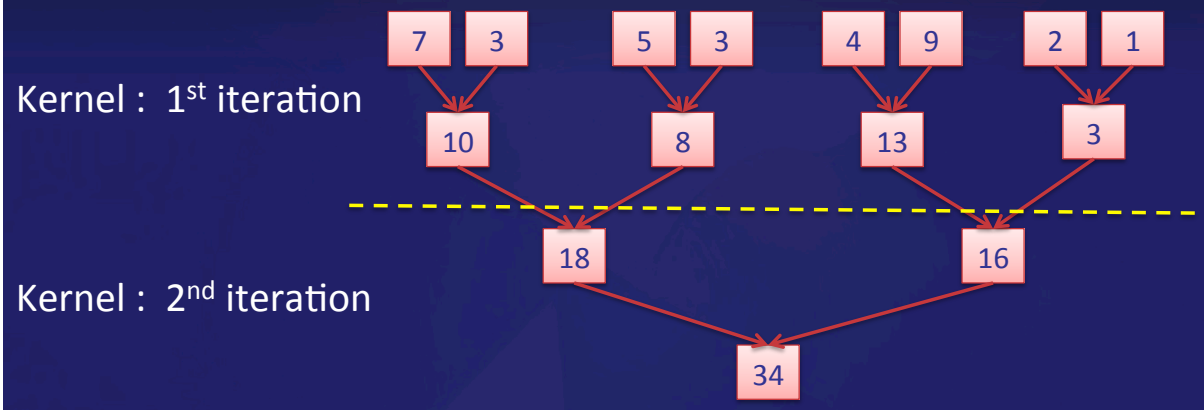


- Tree based computation (pairwise)

Parallel reduction

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

Parallel reduction



- Kernel code is the same
- Global synchronization avoided by decomposing computation into multiple kernel invocations

Parallel Reduction

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
```


Parallel reduction

```
// do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
```

Parallel Reduction Timing (4M integer)

	Time: 2^{22} integer	Bandwidth	Improvement
Kernel 1	4.4662 ms	3.7565 GB/s	

- Parallel reduction is memory bound problem
- Theoretical bandthwidth (Quadro FX5800)
  - 800 * 10^6 * (512/ 8) * 2 * 10 ^ -9 = 102.4 GB/s
- Effective bandwith
  - (2 ^ 22) * 4 (B/element) * 1000 ^ -3 * 1 (op/load) / TIME = 3.7565 GB/s
- Problem:

Highly divergence within a warp

Parallel Reduction

- Interleave addressing width divergence

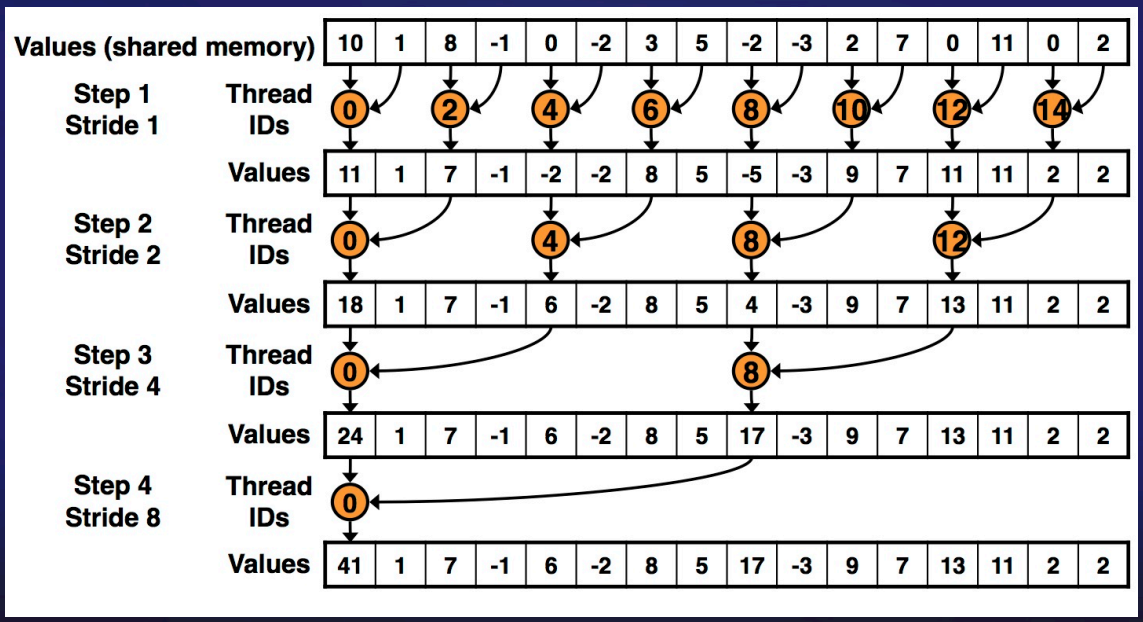
```
// do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
```

Parallel Reduction

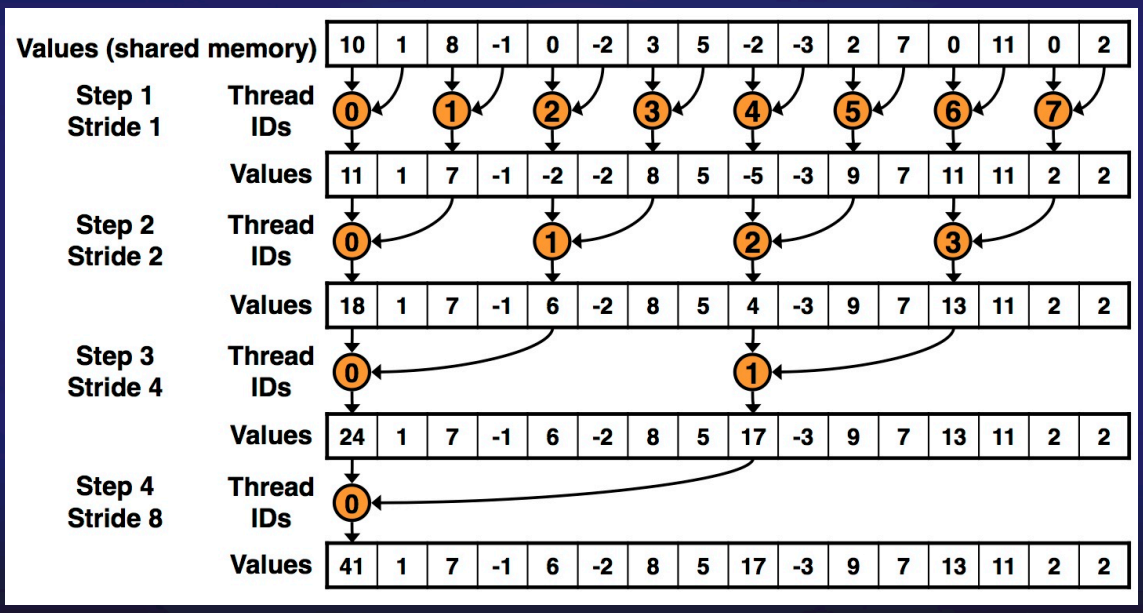
- Interleave addressing widthout divergence

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
        int index = 2 * s * tid;
        if (index < blockDim.x) {
                sdata[index] += sdata[index + s];
        }
        __syncthreads();
}
```


Parallel Reduction



Parallel Reduction



Parallel Reduction Timing (4M integer)

	Time: 2^{22} integer	Bandwidth	Improvement	Cumulative Improvement
Kernel 1	4.4662 ms	3.7565 GB/s		
Kernel 2	2.0614 ms	8.1388 GB/s	2.16	2.16

Problem:

Shared memory bank conflict

Shared memory bank conflict

- Shared Memory
  - Is on-chip:
    - much faster than the local and global memory,
    - as fast as a register when no bank conflicts,
    - divided into equally-sized memory banks.
  - Successive 32-bit words are assigned to successive banks,
  - Each bank has a bandwidth of 32 bits per clock cycle.

Shared memory bank conflict

- Shared Memory

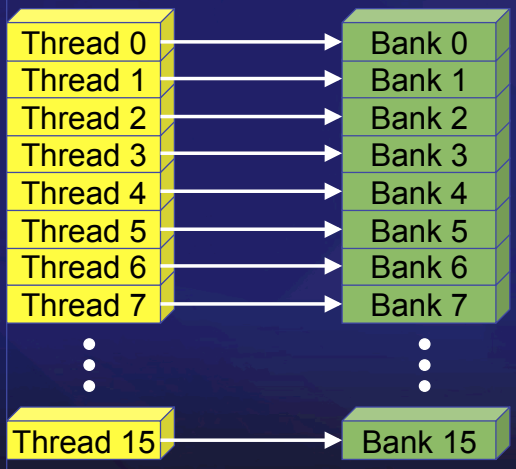
Reminder: warp size is 32, number of banks is 16

- memory request requires two cycles for a warp
  - One for the first half, one for the second half of the warp

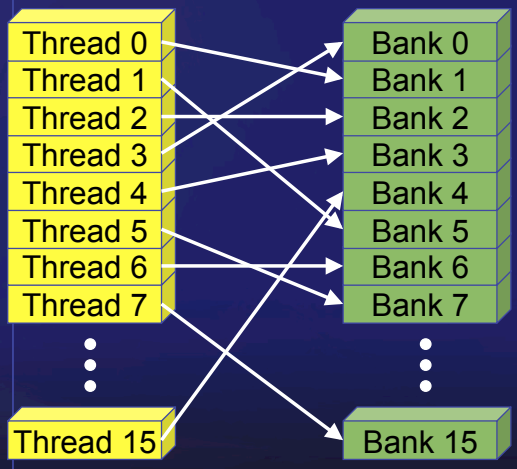
→No conflicts between threads from first and second half

Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

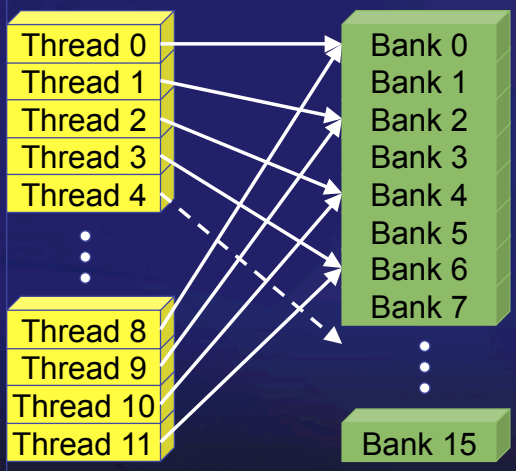


- No Bank Conflicts
  - Random 1:1 Permutation

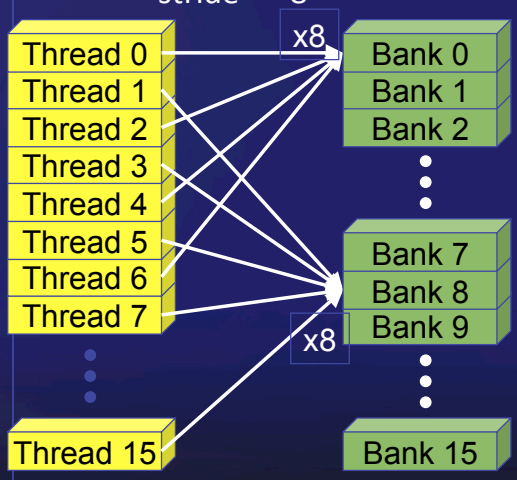


Bank Addressing Examples

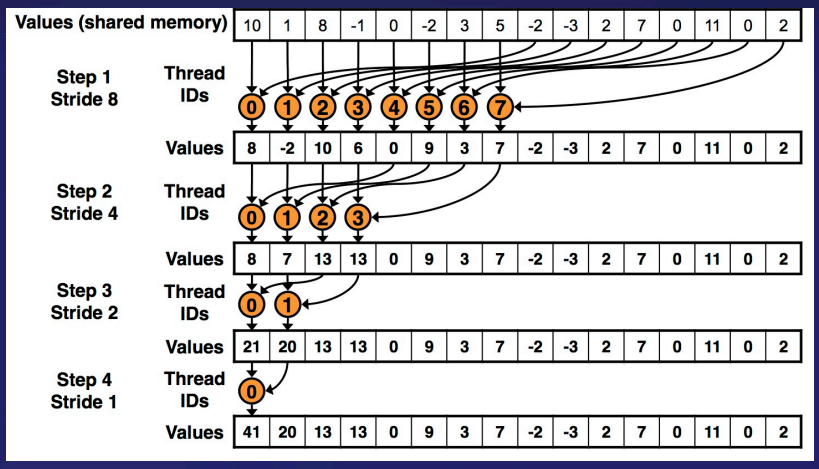
- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



Parallel reduction



Solution:
Sequential memory access

- Replace the inner loop stride
- Invert inner-loop
- Use threadID indexing

Parallel Reduction

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
        int index = 2 * s * tid;
        if (index < blockDim.x) {
                sdata[index] += sdata[index + s];
        }
        __syncthreads();
}
```

Parallel Reduction

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```


Parallel Reduction Timing (4M integer)

	Time: 2^{22} integer	Bandwidth	Improvement	Cumulative Improvement
Kernel 1	4.4662 ms	3.7565 GB/s		
Kernel 2	2.0614 ms	8.1388 GB/s	2.16	2.16
Kernel 3	0.9915 ms	16.921 GB/s	2.08	4.51

Parallel Reduction

- What is the problem where?

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```


Parallel Reduction

- What is the problem where?

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```

- Only half of the threads within a warp are used

Parallel Reduction

- Solution ?
- Launch half of the blocks, load two elements and add

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```


Parallel Reduction Timing (4M integer)

	Time: 2^{22} integer	Bandwidth	Improvement	Cumulative Improvement
Kernel 1	4.4662 ms	3.7565 GB/s		
Kernel 2	2.0614 ms	8.1388 GB/s	2.16	2.16
Kernel 3	0.9915 ms	16.921 GB/s	2.08	4.51
Kernel 4	0.5337 ms	31.434 GB/s	1.85	8.37

- Theoretic peak 192.4 GB/s
- Therefore the likely bottleneck is instruction overhead
- Strategy: unroll loops

Parallel Reduction

- As reduction proceeds, # "active" threads decreases
  - When s <= 32, we have only one warp left
  - Instructions are SIMD synchronous within a warp
- That means when s <= 32:
  - We don't need to __syncthreads()
  - We don't need "if (tid < s)" because it doesn't save any work
- Let's unroll the last 6 iterations of the inner loop

Parallel Reduction

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
        if (tid < s) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}

If(tid < 32) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];
        sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];
        sdata[tid] += sdata[tid +  1];
}
```

Parallel Reduction Timing (4M integer)

	Time: 2^{22} integer	Bandwidth	Improvement	Cumulative Improvement
Kernel 1	4.4662 ms	3.7565 GB/s		
Kernel 2	2.0614 ms	8.1388 GB/s	2.16	2.16
Kernel 3	0.9915 ms	16.921 GB/s	2.08	4.51
Kernel 4	0.5337 ms	31.434 GB/s	1.85	8.37
Kernel 5	0.3514 ms	47.750 GB/s	1.52	12.71

LAB 1.

- Execution: bin/reduce –kernel=K –n=4194304
  - kernel=0 – No optimization (Kernel 1)
  - Kernel=1 – Optimization 1 (Kernel 2)
  - Kernel=2 – Optimization 1 (Kernel 3)
  - Kernel=3 – Optimization 1 (Kernel 4)
  - Kernel=4 – Optimization 1 (Kernel 5)

LAB 1.

- On reduce.cpp find function **getNumBlocksAndThreads** and replace the code by

```
if (whichKernel < 3)
  {
     threads = (n < maxThreads) ? nextPow2(n) : maxThreads;
     blocks = (n + threads - 1) / threads;
  }
  else
  {
     threads = (n < maxThreads*2) ? nextPow2((n + 1)/ 2) : maxThreads;
     blocks = (n + (threads * 2 - 1)) / (threads * 2);
  }
```

On reduce_kernel.cu find the TODO: comments

Parallel Reduction Timing (4M integer)

	Time: 2^{22} integer	Bandwidth	Improvement	Cumulative Improvement
Kernel 1	4.4662 ms	3.7565 GB/s		
Kernel 2	2.0614 ms	8.1388 GB/s	2.16	2.16
Kernel 3	0.9915 ms	16.921 GB/s	2.08	4.51
Kernel 4	0.5337 ms	31.434 GB/s	1.85	8.37
Kernel 5	0.3514 ms	47.750 GB/s	1.52	12.71

- If we know the number of elements at compile time what can we do?

Parallel Reduction

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Luckily, the block size is limited by the GPU to 512 threads
  - Also, we are sticking to power-of-2 block sizes
- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don't know at compile time?
- Templates to the rescue!
  - CUDA supports C++ template parameters on device and host functions

Parallel Reduction

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid)
{
if (blockSize >=  64) sdata[tid] += sdata[tid + 32];
if (blockSize >=  32) sdata[tid] += sdata[tid + 16];
if (blockSize >=  16) sdata[tid] += sdata[tid +  8];
if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
}
```

Parallel Reduction

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int
n) {
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + tid;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
  sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;
}
__syncthreads();
```


Parallel Reduction

```
if (blockSize >= 512) {
if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
__syncthreads();
}
if (blockSize >= 256) {
if (tid < 128) {
sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
if (id <   64) { sdata[tid] += sdata[tid +   64]; }
__syncthreads();
}
if (tid < 32) warpReduce(sdata, tid);
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Parallel Reduction Timing (4M integer)

	Time: 2^{22} integer	Bandwidth	Improvement	Cumulative Improvement
Kernel 1	4.4662 ms	3.7565 GB/s		
Kernel 2	2.0614 ms	8.1388 GB/s	2.16	2.16
Kernel 3	0.9915 ms	16.921 GB/s	2.08	4.51
Kernel 4	0.5337 ms	31.434 GB/s	1.85	8.37
Kernel 5	0.3514 ms	47.750 GB/s	1.52	12.71
Kernel 6	0.2268 ms	75.261 GB/s	1.54	19.70

- See CUDA SDK reduction implementation

CUDA Specialized Libraries and Tools

Why Use Helper Libraries?

- Rapid development
  - Get code working quickly by not recreating basic classes and algorithms
- Robust implementations
  - Classes and functions are generally efficient
  - Many folks use them, so bugs are uncovered and fixed
- However, a custom implementation might be better if the class or function is heavily used
  - Remember Amdahl's law: making part of the code faster only helps overall runtime if that code was already a large fraction of the runtime

CUDA Libraries and Tools

- Thrust – C++ template library based on the STL
- CUBLAS – BLAS (basic linear algebra subprograms)
- CUFFT – discrete FFT (fast Fourier transform)
- CUSPARSE – sparse matrix BLAS
- CURAND – high---quality pseudorandom and quasirandom numbers

CUDA Libraries and Tools

- Overarching theme:
  - PROGRAMMER PRODUCTIVITY !!!
- Programmer productivity
  - Rapidly develop complex applications
  - Leverage parallel primitives
- Encourage generic programming
  - Don't reinvent the wheel
  - E.g. one reduction to rule them all
- High performance
  - With minimal programmer effort

CUDA Libraries and Tools

Libraries

- CUBLAS
- CUFFT
- MAGMA
- CULA
- Thrust
- …

Tools

- CUDA-gdb
- CUDA-Memcheck
- CUDA Visual Profiler
- Nexus
- …

CUDA Specialized Libraries: CUBLAS

- Cuda Based Linear Algebra Subroutines
- Saxpy, conjugate gradient, linear solvers.
- 3D reconstruction of planetary nebulae example.

CUDA Specialized Libraries: CUBLAS

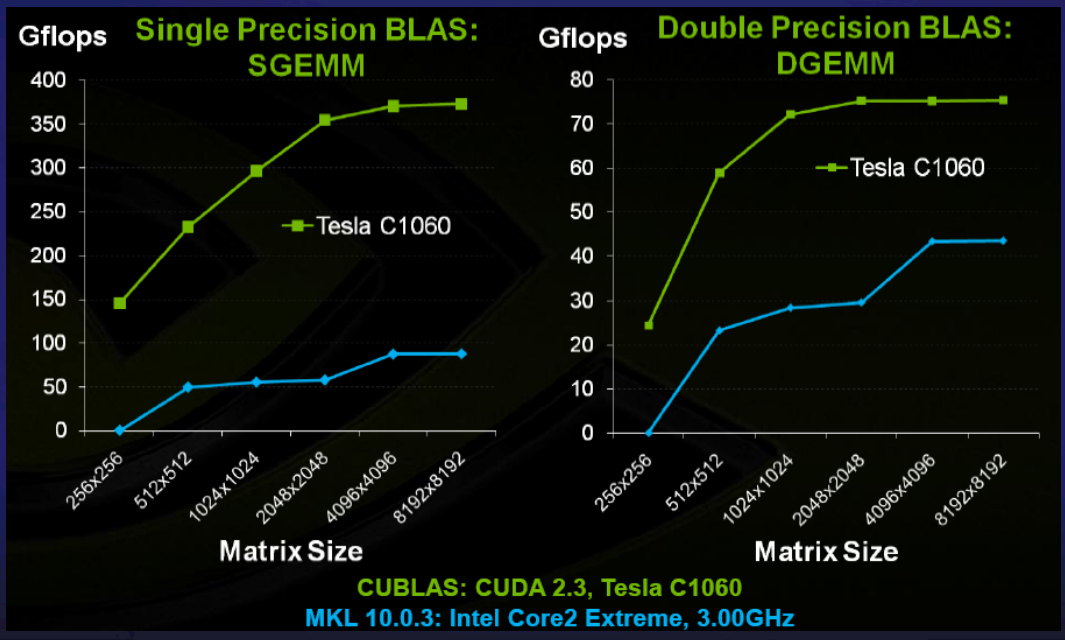
- Basic Linear Algebra Subprograms implementation in CUDA

```
while( i++ < max_iter && deltanew > stop_tol )
{
  cublasSgemv ('n', N, N, 1.0, d_A, N, d_d, 1, 0, d_y, 1);
  float alpha =  deltanew / cublasSdot(N,d_d,1,d_y,1);
  cublasSaxpy(N, alpha,d_d,1,d_x,1);

  // every 50 iterations, restart residual
  if (i % 50 == 0) {
    cublasSgemv('n', N, N, 1.0, d_A, N, d_x, 1, 0, d_y, 1);
    cublasScopy(N, d_b, 1, d_r, 1);
    cublasSaxpy(N, -1.0, d_y, 1, d_r, 1);
  }
  else
    cublasSaxpy(N,-alpha,d_y,1,d_r,1);
...
```

CUDA Specialized Libraries: CUBLAS

- Single-precision
  - Level 1/2/3 (vector-vector/vector-matrix/matrix-matrix)
- Complex-single precision
  - Level 1 and CGEMM
- Double precision
  - Level 1: DASUM, DAXPY, DCOPY, DDOT, DNRM2, DROT, DROTM, DSCAL, DSWAP, ISAMAX, IDAMIN
  - Level 2: DGEMV, DGER, DSYR, DTRSV
  - Level 3: ZGEMM, DGEMM, DTRSM, DTRMM, DSYMM, DSYRK, DSYR2K
- Following BLAS convention, CUBLAS uses column-major storage

CUBLAS: Performance – CPU vs GPU



CUDA Specialized Libraries: CUFFT

- Cuda Based Fast Fourier Transform Library.
- The FFT is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets,
- One of the most important and widely used numerical algorithms, with applications that include computational physics and general signal processing

CUDA Specialized Libraries: CUFFT

- The Fast Fourier Transform (FFT) is a divide-andconquer algorithm for efficiently computing discrete Fourier transform of complex or real-valued data sets.

- CUFFT is the CUDA FFT library
  - Provides a simple interface for computing parallel FFT on an NVIDIA GPU
  - Allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom, GPU-based FFT implementation

CUDA Specialized Libraries: CUFFT

- 1D, 2D and 3D transforms of complex and real-valued

data

- Batched execution for doing multiple 1D transforms in parallel
- 1D transform size up to 8M elements
- 2D and 3D transform sizes in the range [2,16384]
- In-place and out-of-place transforms for real and

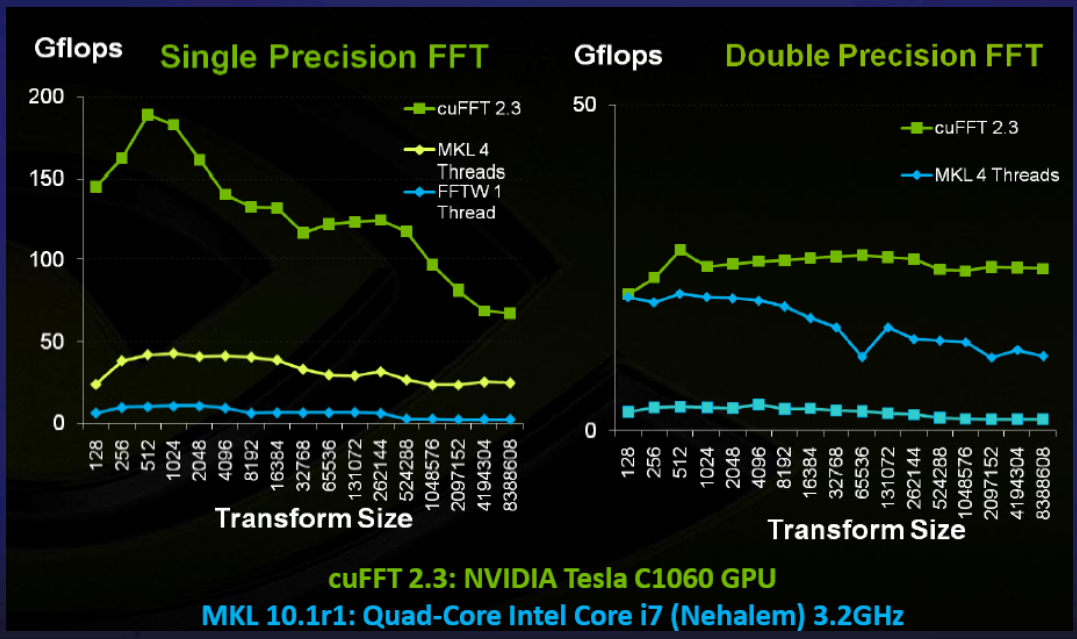
complex data.

Code example: 2D complex to complex transform

Complex 2D transform

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);

/* Create a 2D FFT plan. */
cufftPlan2d(&plan, NX,NY, CUFFT_C2C);

/* Use the CUFFT plan to transform the signal out of place. */
cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* Destroy the CUFFT plan. */
cufftDestroy(plan);

cudaFree(idata);
cudaFree(odata);
```

CUFFT: Performance – CPU vs GPU



THRUST

- C++ style template library, based on the STL (standard template library)
- Contains efficient host and device vector class
- Transformation algorithms to apply a function across a vector of elements
- Reductions across a vector of elements
- Scans to compute prefix sums
- Sorting operations
- “Fancy” iterators to transform, permute or combine elements in vectors
- Thrust library is on the default CUDA toolkit path
- Uses the thrust::namespace, similar to std::
- Full docs at http://wiki.thrust.googlecode.com/hg/html/index.html

THRUST: Algorithms + Containers

```
int main(void)
{
// generate random data on the host
thrust::host_vector<int> h_vec(1000000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);
// transfer to device
thrust::device_vector<int> d_vec = h_vec;
// sort 140M 32b keys/sec on GT200
thrust::sort(d_vec.begin(), d_vec.end());
return 0;
}
```


THRUST: Algorithms + Containers

```
int main(void)
{
// generate random data on the host
thrust::host_vector<int> h_vec(1000000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);
// transfer to device
thrust::device_vector<int> d_vec = h_vec;
// sort 140M 32b keys/sec on GT200
thrust::sort(d_vec.begin(), d_vec.end());
return 0;
}
```

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

What is THRUST?

- C++ template library for CUDA
  - Mimics Standard Template Library (STL)
- Containers
  - thrust::host_vector<T>
  - thrust::device_vector<T>
- Algorithms
  - thrust::sort()
  - thrust::reduce()
  - thrust::inclusive_scan()
  - thrust::segmented_inclusive_scan()
  - Etc.

THRUST: Containers

- Make common operations concise and readable
  - Hides cudaMalloc & cudaMemcpy

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
```

THRUST: Containers

- Compatible with STL containers
  - Eases integration
  - vector, list, map, ...

```
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method
thrust::device_vector<int> d_vec(h_list.begin(), h_list.end());
```


THRUST: Iterator

- Track memory space (host/device)
  - Guides algorithm dispatch

```
// initialize random values on host
thrust::host_vector<int> h_vec(1000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
thrust::device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());
```

THRUST: Algorithms

- Thrust provides ~50 algorithms
  - Reduction
  - Prefix Sums
  - Sorting
- Generic definitions
  - General Types
    - Builtin types (int, float, …)
    - User-defined structures
  - General Operators
    - reduce with plus(a,b)
    - scan with maximum(a,b)

THRUST: Algorithms

- General types and operators

```
// declare storage
device_vector<int>   i_vec = ...
device_vector<float> f_vec = ...

// sum of integers (equivalent calls)
reduce(i_vec.begin(), i_vec.end());
reduce(i_vec.begin(), i_vec.end(),    0, plus<int>());

// sum of floats (equivalent calls)
reduce(f_vec.begin(), f_vec.end());
reduce(f_vec.begin(), f_vec.end(), 0.0f, plus<float>());

// maximum of integers
reduce(i_vec.begin(), i_vec.end(),    0, maximum<int>());
```

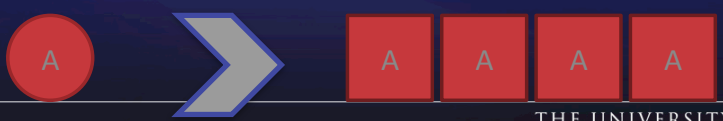
Fancy Iterators

- Behave like “normal” iterators
  - Algorithms don't know the difference
- Examples
  - constant_iterator
  - counting_iterator
  - transform_iterator
  - zip_iterator

Fancy Iterators

- constant_iterator
  - An infinite array filled with a constant value

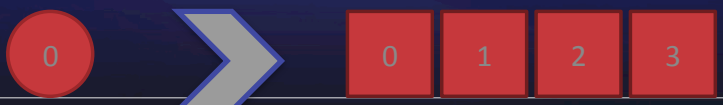
```
// create iterators
constant_iterator<int> first(10);
constant_iterator<int> last = first + 3;

first[0]   // returns 10
first[1]   // returns 10
first[100] // returns 10

// sum of [first, last)
reduce(first, last);   // returns 30 (i.e. 3 * 10)
```



Fancy Iterators

- counting_iterator
  - An infinite array with sequential values

```
// create iterators
counting_iterator<int> first(10);
counting_iterator<int> last = first + 3;

first[0]   // returns 10
first[1]   // returns 11
first[100] // returns 110

// sum of [first, last)
reduce(first, last);   // returns 33 (i.e. 10 + 11 + 12)
```



Fancy Iterators

- transform_iterator
  - Yields a transformed sequence
  - Facilitates kernel fusion

Fancy Iterators

- transform_iterator
  - Conserves memory capacity and bandwidth

```
// initialize vector
device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;

// create iterator (type omitted)
first = make_transform_iterator(vec.begin(), negate<int>());
last  = make_transform_iterator(vec.end(),   negate<int>());

first[0]   // returns -10
first[1]   // returns -20
first[2]   // returns -30

// sum of [first, last)
reduce(first, last);   // returns -60 (i.e. -10 + -20 + -30)
```


Fancy Iterators

- zip_iterator
  - Looks like an array of structs (AoS)
  - Stored in structure of arrays (SoA)

Fancy Iterators

- zip_iterator

```
// initialize vectors
device_vector<int>  A(3);
device_vector<char> B(3);
A[0] = 10;  A[1] = 20;  A[2] = 30;
B[0] = 'x'; B[1] = 'y'; B[2] = 'z';

// create iterator (type omitted)
first = make_zip_iterator(make_tuple(A.begin(), B.begin()));
last  = make_zip_iterator(make_tuple(A.end(),   B.end()));

first[0]   // returns tuple(10, 'x')
first[1]   // returns tuple(20, 'y')
first[2]   // returns tuple(30, 'z')

// maximum of [first, last)
maximum< tuple<int,char> > binary_op;
reduce(first, last, first[0], binary_op); // returns tuple(30, 'z')
```


Features & Optimizations

- gather & scatter
  - Works between host and device
- fill & reduce
  - Avoids G8x coalesing rules for char, short, etc.
- sort
  - Dispatches radix_sort for all primitive types
    - Uses optimal number of radix_sort iterations
  - Dispatches merge_sort for all other types

Example: 2D Bucket Sort

Procedure:

[Step 1] create random points

[Step 2] compute bucket index for each point

[Step 3] sort points by bucket index

[Step 4] compute bounds for each bucket

Example: 2D Bucket Sort

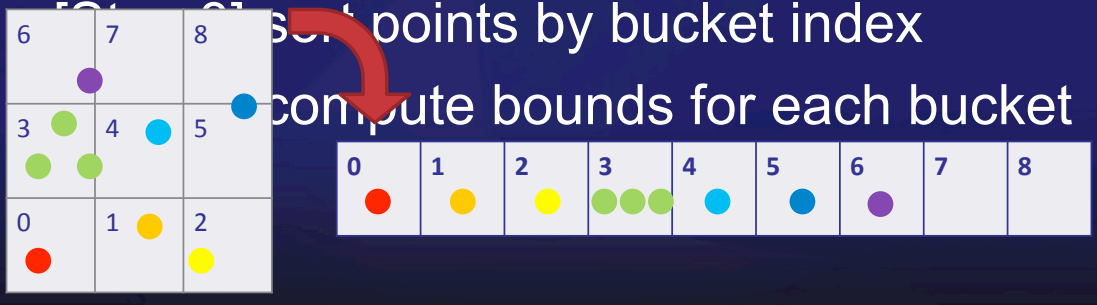
Procedure:

[Step 1] create random points

[Step 2] compute bucket index for each point

[Step 3] sort points by bucket index

[Step 4] compute bounds for each bucket



Example: 2D Bucket Sort

[Step 1] create random points

```
// number of points
const size_t N = 100000;

// return a random float2 in [0,1)^2
float2 make_random_float2(void)
{
  return make_float2( rand() / (RAND_MAX + 1.0f),
                      rand() / (RAND_MAX + 1.0f) );
}

// allocate some random points in the unit square on the host
host_vector<float2> h_points(N);
generate(h_points.begin(), h_points.end(), make_random_float2);

// transfer to device
device_vector<float2> points = h_points;
```


Example: 2D Bucket Sort

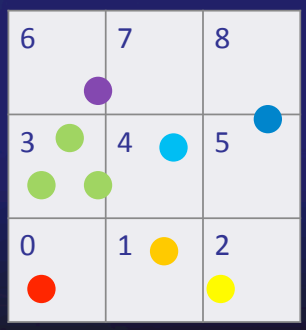
[Step 2] compute bucket index for each point

```
struct point_to_bucket_index
{
  unsigned int w, h;

  __host__ __device__
  point_to_bucket_index(unsigned int width, unsigned int height)
    :w(width), h(height){}

  __host__ __device__
  unsigned int operator()(float2 p) const
  {
    // coordinates of the grid cell containing point p
    unsigned int x = p.x * w;
    unsigned int y = p.y * h;

    // return the bucket's linear index
    return y * w + x;
  }
};
```

Example: 2D Bucket Sort

[Step 2] compute bucket index for each point

```
// resolution of the 2D grid
unsigned int w = 200;
unsigned int h = 100;

// allocate storage for each point's bucket index
device_vector<unsigned int> bucket_indices(N);

// transform the points to their bucket indices
transform(points.begin(),
          points.end(),
          bucket_indices.begin(),
          point_to_bucket_index(w,h));
```



Example: 2D Bucket Sort

[Step 3] sort points by bucket index

```
// sort the points by their bucket index
sort_by_key(bucket_indices.begin(),
            bucket_indices.end(),
            points.begin());
```

Example: 2D Bucket Sort

[Step 4] compute bounds for each bucket

```
// bucket_begin[i] indexes the first element of bucket i
// bucket_end[i] indexes one past the last element of bucket i
device_vector<unsigned int> bucket_begin(w*h);
device_vector<unsigned int> bucket_end(w*h);

// used to produce integers in the range [0, w*h)
counting_iterator<unsigned int> search_begin(0);

// find the beginning of each bucket's list of points
lower_bound(bucket_indices.begin(), bucket_indices.end(),
            search_begin, search_begin + w*h, bucket_begin.begin());

// find the end of each bucket's list of points
upper_bound(bucket_indices.begin(), bucket_indices.end(),
            search_begin, search_begin + w*h, bucket_end.begin());
```

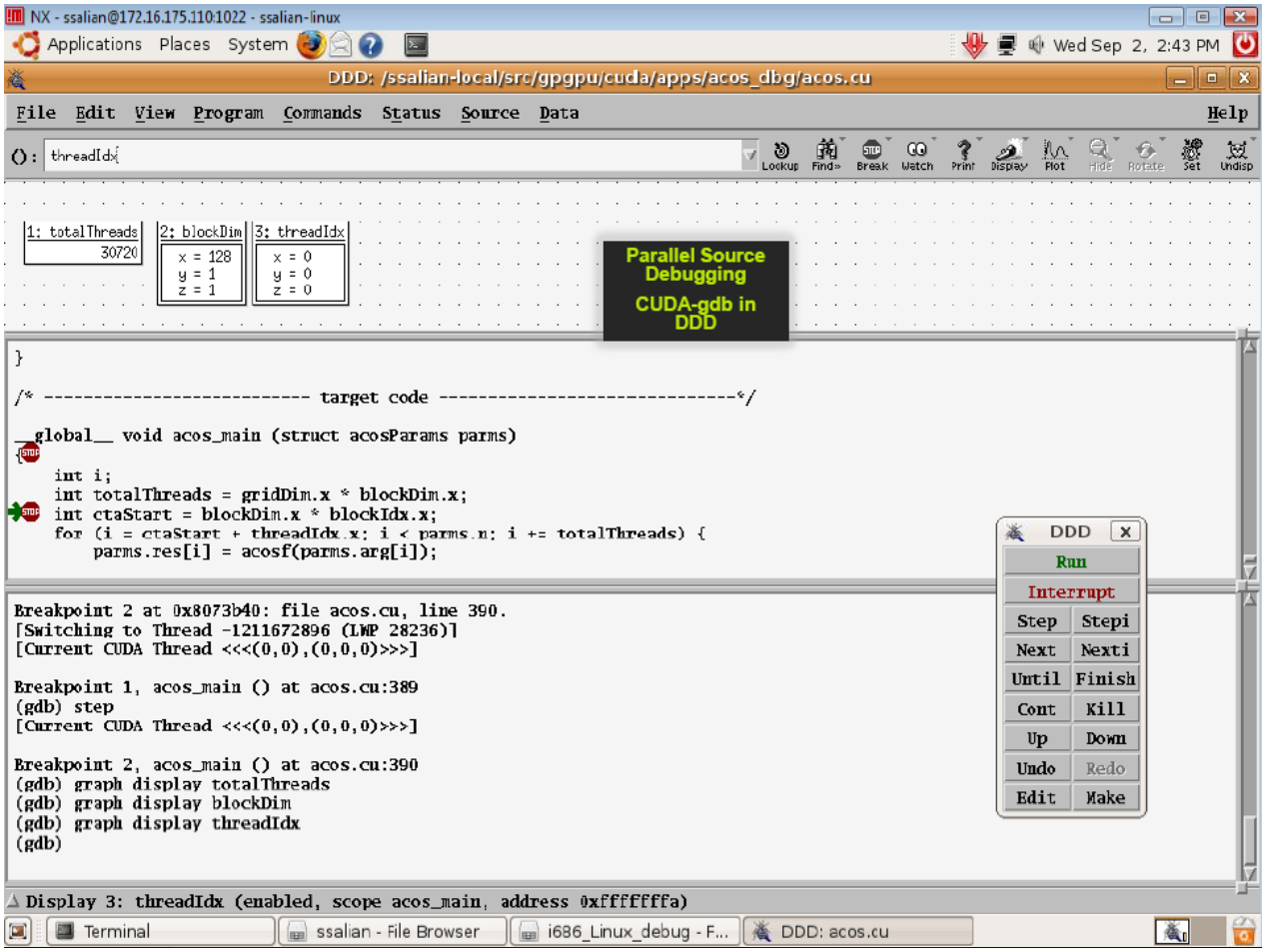

CUDA Development Tools

CUDA Development Tools : CUDA-gdb

- Simple Debugger integrated into gdb

CUDA-gdb

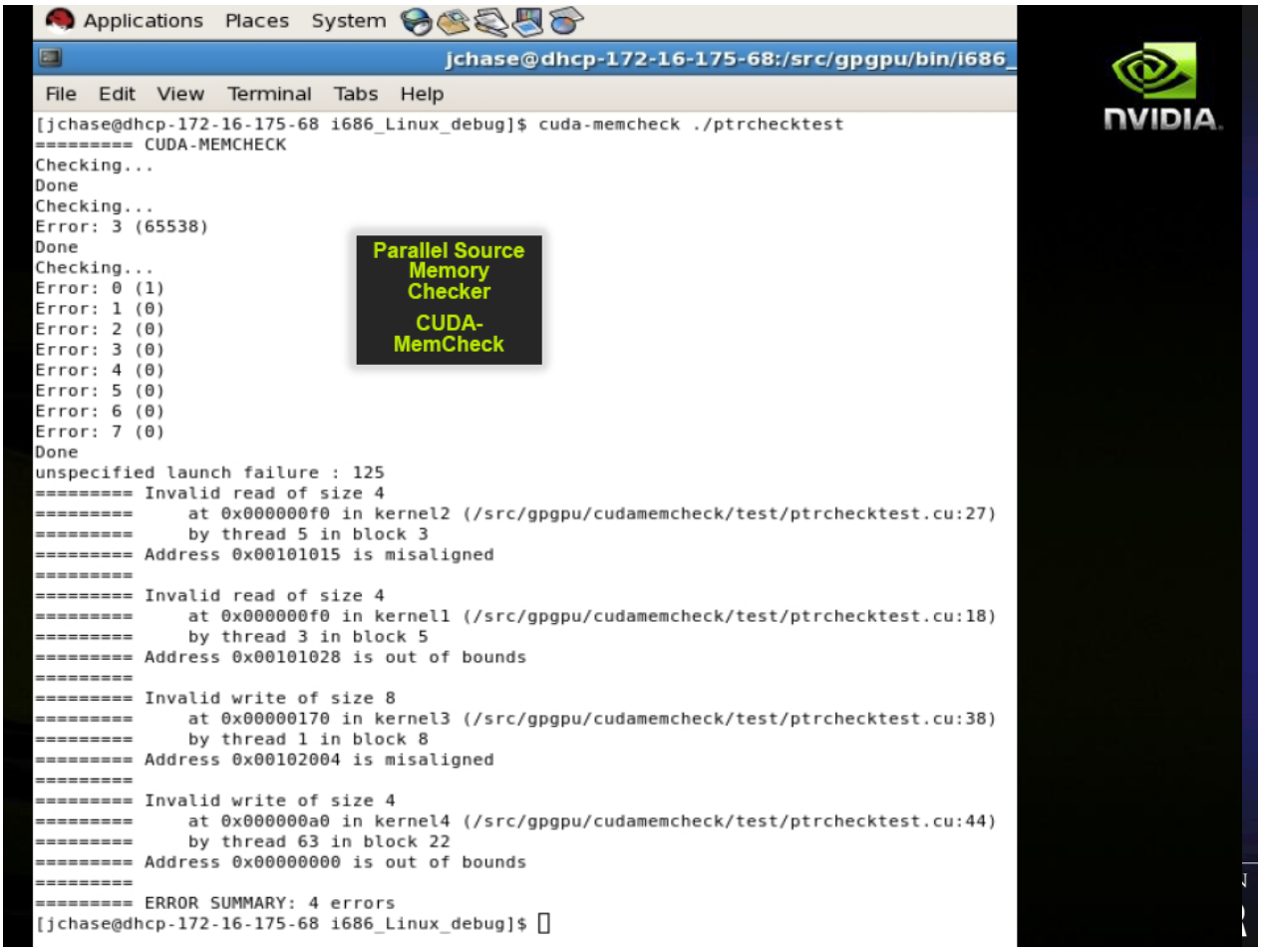
- Integrated into gdb
- Supports CUDA C
- Seamless CPU+GPU development experience
- Enable on all CUDA supported 32/64bit linux distros
- Set breakpoint and single step any source line
- Access and print all CUDA memory allocs, local, global, constant and shared vars.



CUDA Development Tools : MemCheck

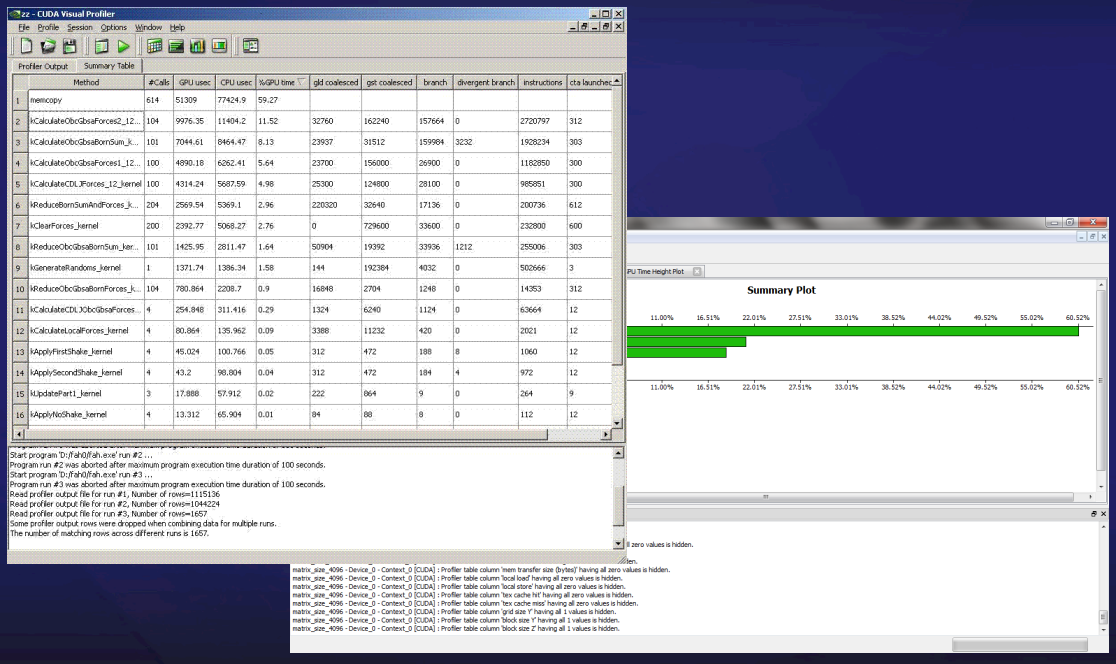
CUDA - MemCheck

- Track out of bound and misaligned accesses
- Supports CUDA C
- Integrated into the CUDA-GDB debugger
- Available as a standalone



CUDA Development Tools : Visual Profiler

CUDA Visual Profiler



CUDA Visual Profiler

- **Events are tracked with hardware counters on signals in the chip:**
  - timestamp
  - gld_incoherent
  - gld_coherent
  - gst_incoherent
  - gst_coherent

    Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent) (Compute 1.0/1.1)
  - local_load
  - local_store

    Local loads/stores
  - branch
  - divergent_branch

    Total branches and divergent branches taken by threads
  - instructions – instruction count
  - warp_serialize – thread warps that serialize on address conflicts to shared or constant memory
  - cta_launched – executed thread blocks

CUDA Development Tools : NSight

- IDE for GPU Computing on Windows: Code Named NSight

NVIDIA IDE: code name NSight

The first development environment for massively parallel applications.

- **Hardware** GPU Source Debugging
- **Platform-wide** Analysis
- Complete **Visual Studio-integration**