CUDA Technical Training

Volume II:

CUDA Case Studies

Prepared and Provided by NVIDIA

Q2 2008

Table of Contents

Computational Finance in CUDA

Options Pricing with Black-Scholes and Monte Carlo

Overview

- **CUDA is ideal for finance computations**
  - **Massive data parallelism in finance**
  - **Highly independent computations**
  - **High computational intensity (ratio of compute to I/O)**
  - **All of this results in high scalability**
- **We'll cover European Options pricing in CUDA with two methods**
  - **Black-Scholes**
  - **Monte Carlo simulation**

Black-Scholes pricing for European options using CUDA

Overview

This presentation will show how to implement an option pricer for European put and call options using CUDA:

- Generate random input data on host
- Transfer data to GPU
- Compute prices on GPU
- Transfer prices back to host
- Compute prices on CPU
- Check the results

Simple problem to map, each option price is computed independently.

European options

$$V_{call} = S \cdot CND(d_1) - X \cdot e^{-rT} \cdot CND(d_2)$$

$$V_{put} = X \cdot e^{-rT} \cdot CND(-d_2) - S \cdot CND(-d_1)$$

$$d_1 = \frac{\log(\frac{S}{X}) + (r + \frac{v^2}{2})T}{v\sqrt{T}}$$

$$d_2 = \frac{\log(\frac{S}{X}) + (r - \frac{v^2}{2})T}{v\sqrt{T}}$$

$$CND(-d) = 1 - CND(d)$$

S is current stock price, **X** is the strike price, **CND** is the Cumulative Normal Distribution function, **r** is the risk-free interest rate, **v** is the volatility

Cumulative Normal Distribution Function

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{u^2}{2}} du$$

Computed with a polynomial approximation (see Hull):
six-decimal place accuracy with a 5th degree polynomial

```
__host__ __device__ float CND(float d)
{
        float K = 1.0f / (1.0f + 0.2316419f * fabsf(d));
        float CND = RSQRT2PI * expf(- 0.5f * d * d) *
                (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));
        if(d > 0)
        CND = 1.0f - CND;

return CND;
}
```

- Make your code float safe.
- Compiler will generate 2 functions, one for the host , one for the device

Implementation steps

The following steps need to be performed:

1. Allocate arrays on host: hOptPrice(N), hOptStrike (N) ,…
2. Allocate arrays on device: dOptPrice(N),dOptStrike(N),…
3. Initialize input arrays
4. Transfer arrays from host memory to the corresponding arrays in device memory
5. Compute option prices on GPU with fixed configuration
6. Transfer results from the GPU back to the host
7. Compute option prices on GPU
8. Compare results
9. Clean-up memory

Code walk-through (steps 1-3)

```
/* Allocate arrays on the host */
    float *hOptPrice, *hOptStrike, *hOptYear;
    hOptPrice  = (float *) malloc(sizeof(float*N);
    hOptStrike = (float *) malloc(sizeof(float*N);
    hOptYear   = (float *) malloc(sizeof(float*N);

/* Allocate arrays on the GPU with cudaMalloc */
    float *dOptPrice, *dOptStrike, *dOptYear;
    cudaMalloc( (void **) &dOptPrice,  sizeof(float)*N);
    cudaMalloc( (void **) &dOptStrike, sizeof(float)*N);
    cudaMalloc( (void **) &dOptYear ,  sizeof(float)*N);
    ……

/* Initialize hOptPrice, hOptStrike, hOptYear on the host */
    ……
```

Code walk-through (steps 4-5)

```
/*Transfer data from host to device with
   cudaMemcpy(target, source, size, direction)*/
  cudaMemcpy (dOptPrice, hOptPrice, sizeof(float)*N ,
                              cudaMemcpyHostToDevice);
  cudaMemcpy (dOptStrike, hOptStrike, sizeof(float)*N    ,
                               cudaMemcpyHostToDevice);
  cudaMemcpy (dOptYears, hOptYears, sizeof(float)*N,
                              cudaMemcpyHostToDevice);

 /* Compute option prices on GPU with fixed configuration
      <<<Nblocks, Nthreads>>>*/
   BlackScholesGPU<<<128, 256>>>(
                     dCallResult, dPutResult,
                     dOptionStrike, dOptionPrice, dOptionYears,
                     RISKFREE, VOLATILITY, OPT_N);
```


Code walk-through (step 6-9)

```
/*Transfer data from device to host  with
   cudaMemcpy(target, source, size, direction)*/
      cudaMemcpy (hCallResult  , dCallResult  , sizeof(float)*N,
                                    cudaMemcpyDeviceToHost);
      cudaMemcpy (hPutResult  , dPutResult  , sizeof(float)*N,
                                    cudaMemcpyDeviceToHost);

/* Compute option prices on the CPU */
      BlackScholesCPU(…..);

/* Compare results */
      …….
/* Clean up memory on host and device*/
      free( hOptPrice);
      ……..
      cudaFree(dOptPrice);
      …….
```

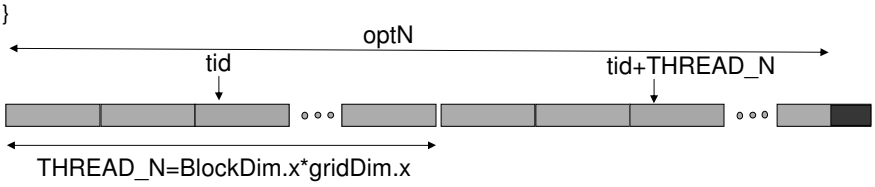
BlackScholesGPU

How to deal with a generic number of options OptN?

Maximum number of blocks = 65536, Max number of threads per block = 512 (this will limit OptN to 33M)

Solution: Each thread processes multiple options.

```
__global__ void BlackScholes (float *….,   int OptN)
{
      const int     tid = blockDim.x * blockIdx.x + threadIdx.x;
      const int     THREAD_N = blockDim.x * gridDim.x;

      for(int opt = tid; opt < OptN; opt += THREAD_N)
          BlackScholesBody(
             d_CallResult[opt], d_PutResult[opt], d_OptionPrice[opt],
             d_OptionStrike[opt],d_OptionYears[opt], Riskfree,Volatility );
}
```



Compile and run

- **Compile the example BlackScholes.cu:**

  nvcc –O3 –o BlackScholes BlacScholes.cu \\
  -I../../common/inc/ -L ../../lib/ -lcutil -lGL –lglut
  (path to the libraries and include may be different on your system)

- **Run the example: BlackScholes**

  European Options with Black-Scholes formula (1000000 options)

  Copying input data to GPU mem. Transfer time: 10.496000 msecs.

  Executing GPU kernel... GPU time: 0.893000 msecs.

  Reading back GPU results... Transfer time: 15.471000 msecs.

  Checking the results...
  ...running CPU calculations. CPU time: 454.683990 msecs.

  Comparing the results...
  L1 norm: 5.984729E-08 Max absolute error: 1.525879E-05

Improve PCI-e transfer rate

PCI-e transfers from regular host memory have a bandwidth of ~1-1.5 GB/s (depending on the host CPU, chipset).

Using page-locked memory allocation, the transfer speed can reach over 3 GB/s (4 GB/s on particular NVIDIA chipset with “LinkBoost”).

CUDA has special memory allocation functions for this purpose.

How to use pinned memory

- Replace cudaMalloc with cudaMallocHost
- Replace cudaFree with cudaFreeHost

```
/* Memory allocation  (instead of regular malloc)*/
    cudaMallocHost ((void **) &h_CallResultGPU, OPT_SZ);

/* Memory clean-up (instead of regular free) */
    cudaFreeHost(h_CallResultGPU);
```

Compile and run

- **Compile the example BlackScholesPinned.cu:**

  nvcc –O3 –o BlackScholesPinned BlacScholesPinned.cu \\
  -I../../common/inc/ -L ../../lib/ -lcutil -lGL –lglut
  (path to the libraries and include may be different on your system)

- **Run the example: BlackScholesPinned**

  European Options with Black-Scholes formula (1000000 options)

  Copying input data to GPU mem. Transfer time: 3.714000 msecs. (was 10.4)

  Executing GPU kernel... GPU time: 0.893000 msecs.

  Reading back GPU results... Transfer time: 2.607000 msecs. (was 15.4)

  Checking the results...
  ...running CPU calculations. CPU time: 454.683990 msecs.

  Comparing the results...
  L1 norm: 5.984729E-08 Max absolute error: 1.525879E-05

MonteCarlo simulation for European options using CUDA

Overview

This presentation will show how to implement a MonteCarlo simulation for European call options using CUDA.

Montecarlo simulations are very suitable for parallelization:

- High regularity and locality
- Very high compute to I/O ratio
- Very good scalability

Vanilla MonteCarlo implementation (no variance reductions techniques such as antithetic variables or control variate)

MonteCarlo approach

The MC simulation for option pricing can be described as:

i. Simulate sample paths for underlying asset price
ii. Compute corresponding option payoff for each sample path
iii. Average the simulation payoffs and discount the average value to yield the price of an option

```
% Monte Carlo valuation for a European call in MATLAB
% An Introduction to Financial Option Valuation: Mathematics, Stochastics and
%  Computation , D. Higham

S = 2; E = 1; r = 0.05; sigma = 0.25; T = 3; M = 1e6;
Svals = S*exp((r-0.5*sigma^2)*T + sigma*sqrt(T)*randn(M,1));
Pvals = exp(-r*T)*max(Svals-E,0);
Pmean = mean(Pvals)
width = 1.96*std(Pvals)/sqrt(M);
conf = [Pmean - width, Pmean + width]
```

MonteCarlo example

i. Generate M random numbers: M=200,000,000
   i. Uniform distribution via Mersenne Twister (MT)
   ii. Box-Müller transformation to generate Gaussian distribution
ii. Compute log-normal distributions for N options: N=128
iii. Compute sum and sum of the squares for each option to recover mean and variance
iv. Average the simulation payoffs and discount the average values to yield the prices of the options.

NB: This example can be easily extended to run on multiple GPUs, using proper initial seeds for MT. See the "MonteCarloMultiGPU" sample in the CUDA SDK v1.1

Generate Uniformly Distributed Random Numbers

The Random Number Generator (RNG) used in this example is a parallel version of the Mersenne Twister by Matsumoto and Nishimura, known as Dynamic Creator (DCMT):

- It is fast
- It has good statistical properties
- It generates many independent Mersenne Twisters

The initial parameters are computed off-line and stored in a file.

RandomGPU<<<32,128>>>(d_Random, N_PER_RNG, seed);

32 blocks *128 threads: 4096 independent random streams

On Tesla C870 (single GPU):
200 Million samples in 80 millisecond
2.5 Billion samples per second !!!!

Generating Gaussian Normal Distribution

Use the Box-Müller transformation to generate Gaussian normal distribution from uniformly distributed random values

BoxMullerGPU<<<32,128>>>(d_Random, N_PER_RNG, seed);

On Tesla C870 (single GPU):
200 Million samples in 120 milliseconds

```
#define PI 3.14159265358979323846264338327950288f
__device__ void BoxMuller(float& u1, float& u2){
   float   r = sqrtf(-2.0f * logf(u1));
   float phi = 2 * PI * u2;
   u1 = r * cosf(phi);
   u2 = r * sinf(phi);
}
```

•Possible alternative, Beasley-Springer-Moro algorithm for approximating the inverse normal

Log-normal distributions and partial sums

```
void MonteCarloGPU(d_Random,….)
{
   // Break the sums in 64*256 (16384) partial sums
   MonteCarloKernelGPU<<<64, 256, 0>>>(d_Random);

   //Read back the  partial sums to the host

   cudaMemcpy(h_Sum,  d_Sum,  ACCUM_SZ,  cudaMemcpyDeviceToHost) ;
   cudaMemcpy(h_Sum2, d_Sum2, ACCUM_SZ, cudaMemcpyDeviceToHost) ;

   // Compute sum and sum of squares on host

   double dblSum = 0, dblSum2 = 0;
   for(int i = 0; i < ACCUM_N; i++){
      dblSum  += h_Sum[i];
      dblSum2 += h_Sum2[i];
   }
}
```

Log-normal distributions and partial sums

```
__global__ void MonteCarloKernelGPU(…)
{
  const int     tid = blockDim.x * blockIdx.x + threadIdx.x;
  const int               threadN = blockDim.x * gridDim.x;
  //...

  for(int iAccum = tid; iAccum < accumN; iAccum += threadN)  {
    float sum = 0, sum2 = 0;

    for(int iPath = iAccum; iPath < pathN; iPath += accumN) {
      float r = d_Random[iPath];
      //...
      sum  += endOptionPrice;
      sum2 += endOptionPrice * endOptionPrice;
    }

    d_Sum[iAccum] = sum;
    d_Sum2[iAccum] = sum2;
}
```


Accurate Floating-Point Summation

The standard way of summing a sequence of N numbers , a_i , is the recursive formula:

$S_0 = 0$

$S_i = S_{i-1} + a_i$

$S = S_n$

When using floating-point arithmetics an error analysis (Wilkinson, 1963) shows that the accumulated round-off error can grow as fast as N^2.

By forming more than one intermediate sum, the accumulated round-off error can be significantly reduced

This is exactly how parallel summation works!

Compile and run

- **Compile the example Montecarlo.cu:**

```
nvcc –O3 –o  Montecarlo Montecarlo.cu \\
-I../../common/inc/ -L ../../lib/ -lcutil -lGL –lglut
(path to the libraries and include may be different on your system)
```

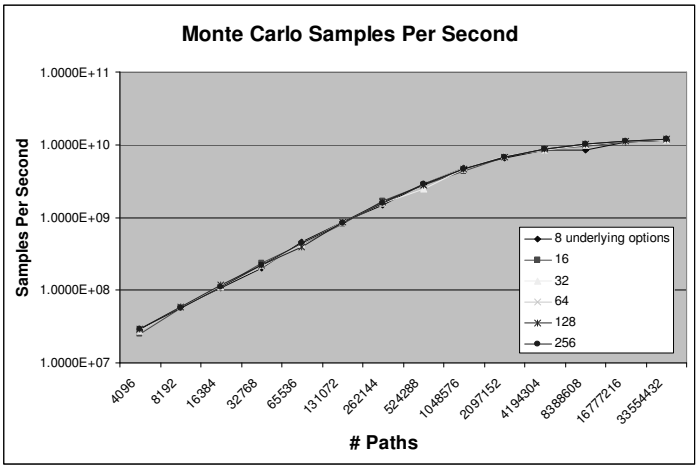
- **Run the example: Montecarlo**

```
MonteCarlo simulation for European call options  ( 200000000 paths)

Generate  Random Numbers on GPU                    80.51499 msecs.

Box Muller on GPU                                 122.62799 msecs.

Average time for option.                           15.471000 msecs.

Checking the results...
MC: 6.621926;    BS: 6.621852
Abs: 7.343292e-05;       Rel: 1.108948e-05;

MC: 8.976083;    BS: 8.976007
Abs: 7.534027e-05;       Rel: 8.393517e-06;
```

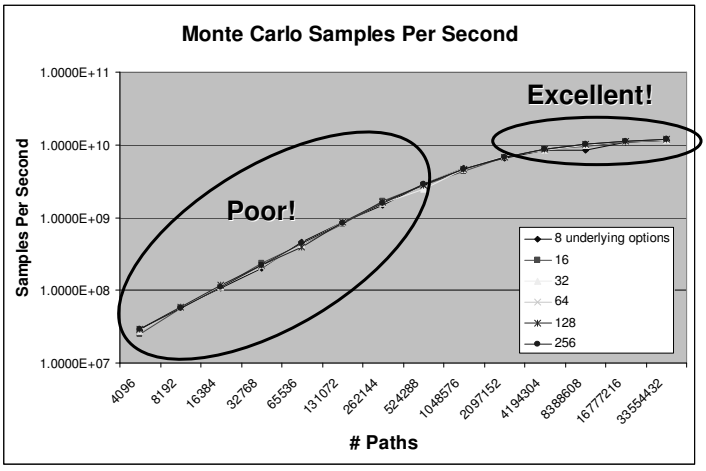

Optimizing for smaller problems

- **This Monte Carlo implementation is optimized for gigantic problems**
  - **e.g. 16M paths on 256 underlying options**
- **Most real simulations are much smaller**
  - **e.g. 256K paths on 64 underlying options**
- **Before optimization, take a detailed look at performance**
  - **Comparing performance for different problem sizes can provide a lot of insight**

Monte Carlo Samples Per Second

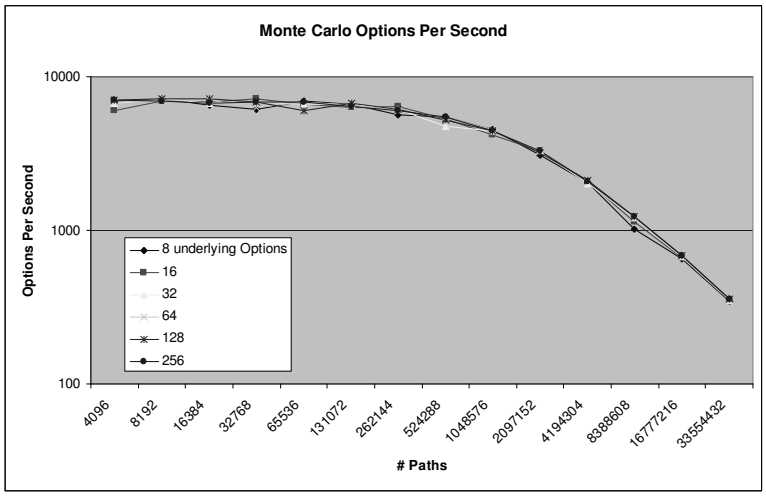


Monte Carlo Samples Per Second

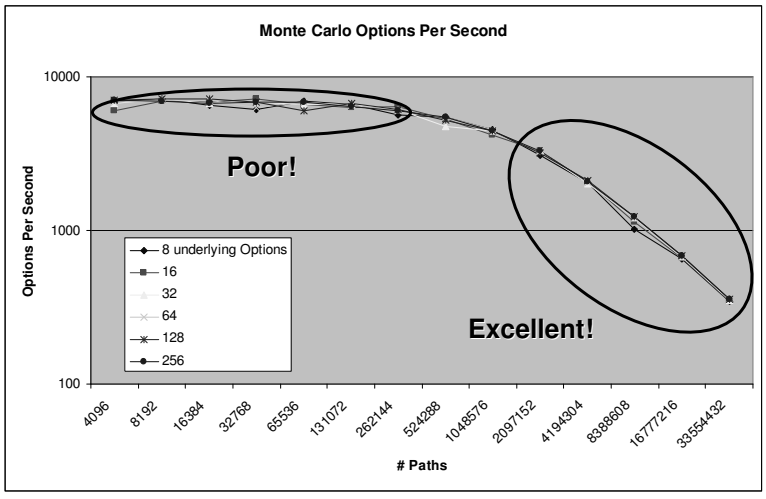


This graph should be a horizontal line!

Monte Carlo Options Per Second



Monte Carlo Options Per Second



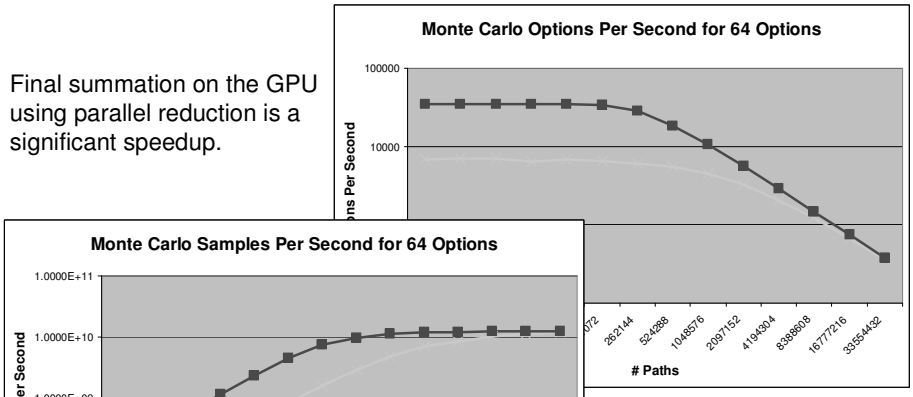
This graph should be a straight diagonal line!

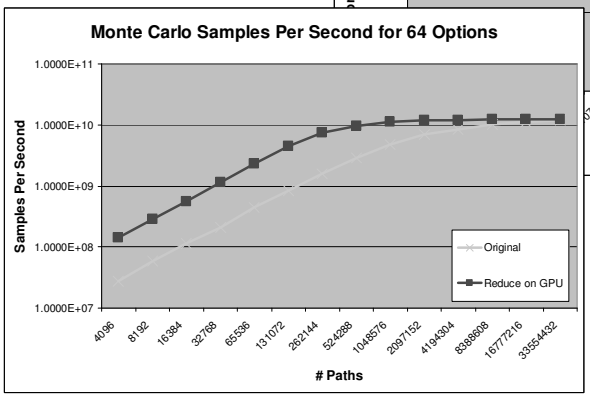
Inefficiencies

- Looking at the code, there were some inefficiencies
  - Final sum reduction on CPU rather than GPU
  - Loop over options on CPU rather than GPU
  - Multiple thread blocks launched per option
- Not evident for large problems because completely computation bound
- NB: In further comparisons, we choose 64 options as our optimization case

Move the reduction onto the GPU

Final summation on the GPU using parallel reduction is a significant speedup.



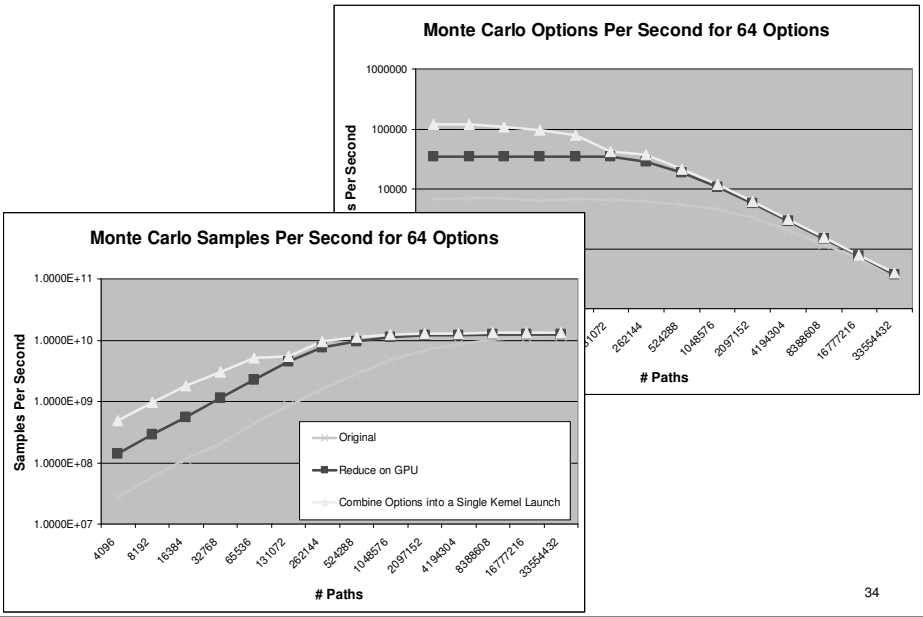


Read back a single sum and sum of squares for each thread block

All options in a single kernel launch

- Initial code looped on host, invoking the kernel once per option
  - 1D grid, multiple thread blocks per option
- Rather than looping, just launch a 2D grid
  - One row of thread blocks per option

All options in a single kernel launch



One Thread Block Per Option

- Pricing an option using multiple blocks requires multiple kernel launches
  - First kernel produces partial sums
  - Second kernel performs sum reduction to get final values
- For very small problems, kernel launch overhead dominates cost
  - And cudaMemcpy dominates if we reduce on CPU
- Solution: for small # paths, use a single thread block per option with a new kernel
  - Summation for entire option is computed in this kernel

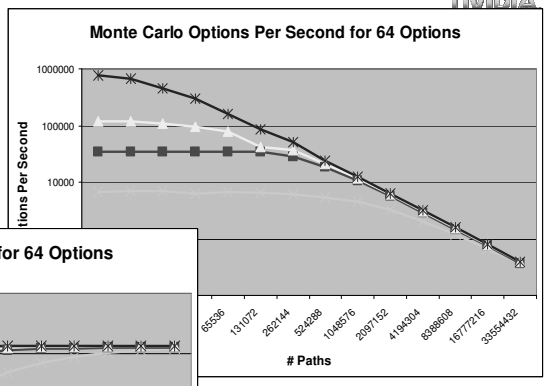
How small is small enough?

- We still want to use the old, two kernel method for large # paths
- How do we know when to switch?
- Imperically, we determined criterion:

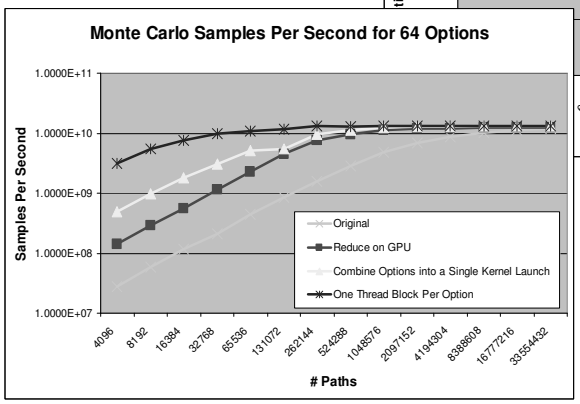
```
bool multiBlock = ((numPaths / numOptions) >= 8192);
```

- If multiBlock is false, we run the single block kernel
- Otherwise, run multi-block kernel followed by reduction kernel

One Thread Block Per Option



These lines are much straighter!



Good performance for small, medium, and large problems!

Details

- For details of these optimizations see the "MonteCarlo" sample code in the CUDA SDK 1.1
  - Latest version was optimized based on these experiments
- Included white paper provides further discussion
- Also see the "MonteCarloMultiGPU" example to see how to distribute the problem across multiple GPUs in a system

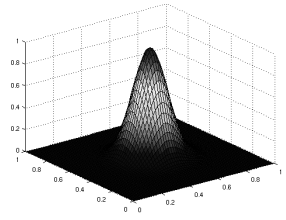
Conclusion

- CUDA is well-suited to computational finance
- Important to tune code for your specific problem
  - Study relative performance for different problem sizes
  - Understand source of bottlenecks for different size problems
  - Optimizations may differ depending on your needs

Spectral Poisson Equation Solver

Overview

In this example, we want to solve a Poisson equation on a rectangular domain with periodic boundary conditions using a Fourier-spectral method.



This example will show how to use the FFT library, transfer the data to/from GPU and perform simple computations on the GPU.

Mathematical background

$$\nabla^2\phi = r \xrightarrow{FFT} -(k_x^2 + k_y^2)\hat{\phi} = \hat{r}$$

1. Apply 2D forward FFT to r to obtain r(k), where k is the wave number
2. Apply the inverse of the Laplace operator to r(k) to obtain u(k): simple element-wise division in Fourier space

$$\hat{\phi} = -\frac{\hat{r}}{(k_x^2 + k_y^2)}$$

3. Apply 2D inverse FFT to u(k) to obtain u

Reference MATLAB implementation

```
% No. of Fourier modes
      N = 64;
% Domain size (assumed square)
      L = 1;
% Characteristic width of f (make << 1)
      sig = 0.1;
% Vector of wavenumbers
      k = (2*pi/L)*[0:(N/2-1) (-N/2):(-1)];
%Matrix of (x,y) wavenumbers corresponding
% to Fourier mode (m,n)
      [KX KY]  = meshgrid(k,k);
% Laplacian matrix acting on the wavenumbers
      delsq = -(KX.^2 + KY.^2);
% Kludge to avoid division by zero for
% wavenumber (0,0).
% (this waveno. of fhat  should be zero anyway!)
      delsq(1,1) = 1;
% Grid spacing
      h = L/N;
      x = (0:(N-1))*h ;
      y = (0:(N-1))*h;
      [X Y] = meshgrid(x,y);
% Construct RHS f(x,y) at the Fourier gridpoints
      rsq = (X-0.5*L).^2 + (Y-0.5*L).^2;
      sigsq = sig^2;
      f = exp(-rsq/(2*sigsq)).*…
          (rsq - 2*sigsq)/(sigsq^2);
% Spectral inversion of Laplacian
      fhat = fft2(f);
      u = real(ifft2(fhat./delsq));
% Specify arbitrary constant by forcing corner
%  u = 0.
      u = u - u(1,1);
%  Compute L2 and Linf norm of error
      uex = exp(-rsq/(2*sigsq));
      errmax = norm(u(:)-uex(:),inf);
      errmax2 = norm(u(:)-uex(:),2)/(N*N);
 %  Print L2 and Linf norm of error
      fprintf('N=%d\n',N);
      fprintf('Solution at (%d,%d): ',N/2,N/2);
      fprintf('computed=%10.6f …
          reference = %10.6f\n',u(N/2,N/2),
      uex(N/2,N/2));
       fprintf('Linf err=%10.6e L2 norm
            err = %10.6e\n',errmax, errmax2);
```

http://www.atmos.washington.edu/2005Q2/581/matlab/pois_FFT.m

Implementation steps

The following steps need to be performed:

1. Allocate memory on host: r (NxN), u (NxN) , kx (N) and ky (N)
2. Allocate memory on device: r_d, u_d, kx_d, ky_d
3. Transfer r, kx and ky from host memory to the correspondent arrays on device memory
4. Initialize plan for FFT
5. Compute execution configuration
6. Transform real input to complex input
7. 2D forward FFT
8. Solve Poisson equation in Fourier space
9. 2D inverse FFT
10. Transform complex output to real input and apply scaling
11. Transfer results from the GPU back to the host

We are not taking advantage of the symmetries (C2C transform for real data) to keep the code simple.

Solution walk-through (steps 1-2)

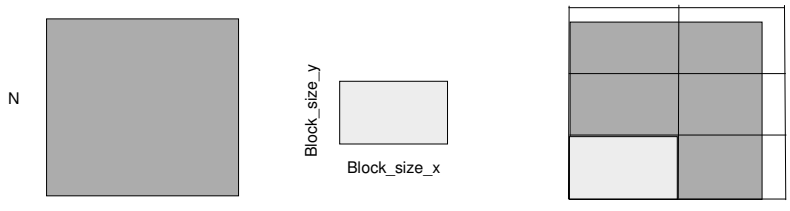
```
/*Allocate arrays on the host */
      float *kx, *ky, *r;
      kx = (float *) malloc(sizeof(float*N);
      ky = (float *) malloc(sizeof(float*N);
      r   = (float *) malloc(sizeof(float*N*N);

/* Allocate array on the GPU with cudaMalloc */
      float *kx_d, *ky_d, *r_d;
      cudaMalloc( (void **) &kx_d, sizeof(cufftComplex)*N);
      cudaMalloc( (void **) &ky_d, sizeof(cufftComplex)*N);
      cudaMalloc( (void **) &r_d  , sizeof(cufftComplex)*N*N);

      cufftComplex *r_complex_d;
      cudaMalloc( (void **) &r_complex_d, sizeof(cufftComplex)*N*N);
```


Code walk-through (steps 3-4)

```
/* Initialize r, kx and ky on the host */
  ……
/*Transfer data from host to device with
   cudaMemcpy(target, source, size, direction)*/
  cudaMemcpy (kx_d, kx, sizeof(float)*N    , cudaMemcpyHostToDevice);
  cudaMemcpy (ky_d, ky, sizeof(float)*N    , cudaMemcpyHostToDevice);
  cudaMemcpy (r_d  , r   , sizeof(float)*N*N, cudaMemcpyHostToDevice);

/* Create plan for CUDA FFT (interface similar to FFTW) */
   cufftHandle plan;
   cufftPlan2d( &plan, N, N, CUFFT_C2C);
```

Code walk-through (step 5)

```
/* Compute the execution configuration
   NB: block_size_x*block_size_y = number of threads
       On G80  number of threads < 512  */
  dim3 dimBlock(block_size_x, block_size_y);
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);

/* Handle N not multiple of block_size_x or block_size_y */
  if (N % block_size_x  !=0 ) dimGrid.x+=1;
  if (N % block_size_y  !=0 ) dimGrid.y+=1
```



Code walk-through (step 6-10)

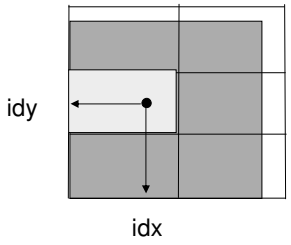
```
/* Transform real input to complex input */
  real2complex<<<dimGrid, dimBlock>>> (r_d, r_complex_d, N);

/* Compute in place forward FFT */
  cufftExecC2C (plan, r_complex_d, r_complex_d, CUFFT_FORWARD);

/* Solve Poisson equation in Fourier space */
  solve_poisson<<<dimGrid, dimBlock>>> (r_complex_d, kx_d, ky_d,N);

/* Compute in place inverse FFT */
  cufftExecC2C (plan, r_complex_d, r_complex_d, CUFFT_INVERSE);

 /* Copy the solution back to a real array and apply scaling ( an FFT followed by
     iFFT will give you back the same array times the length of the transform) */
  scale = 1.f / ( (float) N * (float) N );
  complex2real_scaled<<<dimGrid, dimBlock>>> (r_d, r_complex_d, N, scale);
```

Code walk-through (step 11)

```
/*Transfer data from device to host  with
   cudaMemcpy(target, source, size, direction)*/
  cudaMemcpy (r  , r_d   , sizeof(float)*N*N, cudaMemcpyDeviceToHost);

/* Destroy  plan and clean up memory on device*/
   cufftDestroy( plan);
   cudaFree(r_complex_d);
   …….
   cudaFree(kx_d);
```


real2complex

```
/*Copy real data to complex data */

  __global__ void real2complex (float *a,  cufftComplex *c, int N)
{
      /* compute idx and idy, the location of the element in the original NxN array */
      int idx = blockId.x*blockDim.x+threadIdx.x;
      int idy = blockId.y*blockDim.y+threadIdx.y;

      if ( idx < N && idy <N)
            {
             int index = idx + idy*N;
             c[index].x = a[index];
             c[index].y = 0.f;
            }
}
```



solve_poisson

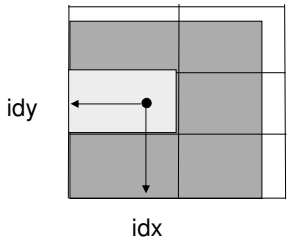
```
__global__ void solve_poisson (cufftComplex *c, float *kx, float *ky, int N)
{
    /* compute idx and idy, the location of the element in the original NxN
    array */
    int idx = blockId.x*blockDim.x+threadIdx.x;
    int idy = blockId.y*blockDim.y+threadIdx.y;

    if ( idx < N && idy <N)
        {
         int index = idx + idy*N;
         float scale =  -  ( kx[idx]*kx[idx] + ky[idy]*ky[idy] );
         if ( idx ==0 && idy == 0 ) scale =1.f;
         scale = 1.f / scale;
         c[index].x *= scale;
         c[index].y *= scale;
        }
}
```

$$\hat{\phi} = -\frac{\hat{r}}{(k_x^2 + k_y^2)}$$

complex2real_scaled

/*Copy real part of complex data into real array and apply scaling */

```
__global__ void complex2real_scaled (cufftComplex *c, float *a, int N,
                                     float scale)
{
    /* compute idx and idy, the location of the element in the original NxN array */
    int idx = blockId.x*blockDim.x+threadIdx.x;
    int idy = blockId.y*blockDim.y+threadIdx.y;

    if ( idx < N && idy <N)
        {
         int index = idx + idy*N;
         a[index] = scale*c[index].x ;
         }
}
```



Compile and run poisson_1

- **Compile the example poisson_1.cu:**

```
nvcc –O3 –o poisson_1 poisson_1.cu \\
-I/usr/local/cuda/include –L/usr/local/cuda/lib  -lcufft –lcudart
```

- **Run the example**

```
./poisson_1 -N64
Poisson solver on a domain  64 x 64
dimBlock 32 16 (512 threads)
dimGrid  2 4
L2 error 9.436995e-08:
Time       0.000569:
Time I/O 0.000200 (0.000136 + 0.000064):
Solution at (32,32)
computed=0.975879 reference=0.975882
```

- **Reference values from MATLAB:**

```
N=64
Solution at (32,32): computed=  0.975879 reference=   0.975882
Linf err=2.404194e-05 L2 norm err = 9.412790e-08
```


Profiling

Profiling the function calls in CUDA is very easy.
It is controlled via environment variables:

- **CUDA_PROFILE: to enable or disable**
  - **1 (enable profiler)**
  - **0 (default, no profiler)**
- **CUDA_PROFILE_LOG: to specify the filename**
  - **If set, it will write to "filename"**
  - **If not set, it will write to cuda_profile.log**
- **CUDA_PROFILE_CSV: control the format**
  - **1 (enable comma separated file)**
  - **0 (disable comma separated file)**

Profiler output from Poisson_1

./poisson_1 –N1024

method=[memcopy] gputime=[1427.200]
method=[memcopy] gputime=[10.112]
method=[memcopy] gputime=[9.632]

method=[real2complex] gputime=[1654.080] cputime=[1702.000] occupancy=[0.667]

method=[c2c_radix4] gputime=[8651.936] cputime=[8683.000] occupancy=[0.333]
method=[transpose] gputime=[2728.640] cputime=[2773.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8619.968] cputime=[8651.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2731.456] cputime=[2762.000] occupancy=[0.333]

method=[solve_poisson] gputime=[6389.984] cputime=[6422.000] occupancy=[0.667]

method=[c2c_radix4] gputime=[8518.208] cputime=[8556.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2724.000] cputime=[2757.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8618.752] cputime=[8652.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2767.840] cputime=[5248.000] occupancy=[0.333]

method=[complex2real_scaled] gputime=[2844.096] cputime=[3613.000] occupancy=[0.667]

method=[memcopy] gputime=[2461.312]

Improving performances

- **Use pinned memory to improve CPU/GPU transfer time:**

```
#ifdef PINNED
      cudaMallocHost((void **) &r,sizeof(float)*N*N); // rhs, 2D array
#else
      r  = (float *) malloc(sizeof(float)*N*N); // rhs, 2D array
#endif
```

```
$ ./poisson_1
Poisson solver on a domain  1024 x 1024
Total Time   : 69.929001 (ms)
Solution Time: 60.551998 (ms)
Time I/O     : 8.788000 (5.255000 + 3.533000) (ms)

$ ./poisson_1_pinned
Poisson solver on a domain  1024 x 1024
Total Time   : 66.554001 (ms)
Solution Time: 60.736000 (ms)
Time I/O     : 5.235000 (2.027000 + 3.208000) (ms)
```

Additional improvements

- Use shared memory for the arrays kx and ky in solve_poisson
- Use fast integer operations (__umul24)

solve_poisson (with shared memory)

```
__global__ void solve_poisson (cufftComplex *c, float *kx, float *ky, int N)
{
      unsigned int idx   = __umul24(blockIdx.x,blockDim.x)+threadIdx.x;
      unsigned int idy   = __umul24(blockIdx.y,blockDim.y)+threadIdx.y;
// use shared memory to minimize multiple access to same k values
    __shared__ float kx_s[BLOCK_WIDTH], ky_s[BLOCK_HEIGHT]
     if (threadIx.x < 1) kx_s[threadIdx.x] = kx[idx];
     if (threadIx.y < 1) ky_s[threadIdx.y] = ky[idy];
     __syncthreads();
      if ( idx < N && idy <N)
            {
            unsigned int index = idx +__umul24(idy ,N);
            float scale =  -  ( kx_s[threadIdx.x]*kx_s[threadIdx.x]
                           +    ky_s[threadIdy.y]*ky_s[threadIdy.y] );
            if ( idx ==0 && idy == 0 ) scale =1.f;
            scale = 1.f / scale;
             c[index].x *= scale;
             c[index].y*= scale;
            }
}
```

Profiler output from Poisson_2

./poisson_2 –N1024 –x16 –y16

method=[memcopy] gputime=[1426.048]
method=[memcopy] gputime=[9.760]
method=[memcopy] gputime=[9.472]

method=[real2complex] gputime=[1611.616] cputime=[1662.000] occupancy=[0.667] (was 1654)

method=[c2c_radix4] gputime=[8658.304] cputime=[8689.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2731.424] cputime=[2763.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8622.048] cputime=[8652.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2738.592] cputime=[2770.000] occupancy=[0.333]

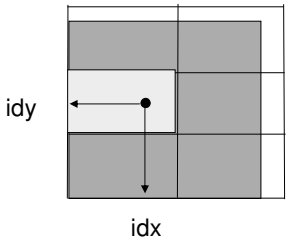
method=[solve_poisson] gputime=[2760.192] cputime=[2792.000] occupancy=[0.667] (was 6389)

method=[c2c_radix4] gputime=[8517.952] cputime=[8550.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2729.632] cputime=[2766.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8621.024] cputime=[8653.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2770.912] cputime=[5252.000] occupancy=[0.333]

method=[complex2real_scaled] gputime=[2847.008] cputime=[3616.000] occupancy=[0.667]
???????
method=[memcopy] gputime=[2459.872]

complex2real_scaled (fast version)

```
__global__ void complex2real_scaled (cufftComplex *c, float *a, int N, float
    scale)
{
    /* compute idx and idy, the location of the element in the original NxN array */
    int idx = blockId.x*blockDim.x+threadIdx.x;
    int idy = blockId.y*blockDim.y+threadIdx.y;
    volatile float2 c2;
        if ( idx < N && idy <N)
        {
         int index = idx + idy*N;
         c2.x= c[index].x;
         c2.y= c[index].y;
         a[index] = scale*c2.x ;
         }
}
```



From the ptx file, we discover that the compiler is optimizing out the vector load which prevents memory coalescing. Use volatile to force vector load

Profiler output from Poisson_3

```
method=[ memcopy ] gputime=[ 1427.808 ]
method=[ memcopy ] gputime=[ 9.856 ]
method=[ memcopy ] gputime=[ 9.600 ]

method=[ real2complex] gputime=[ 1614.144 ] cputime=[ 1662.000 ] occupancy=[ 0.667 ]

method=[ c2c_radix4]       gputime=[ 8656.800 ] cputime=[ 8688.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2727.200 ] cputime=[ 2758.000 ] occupancy=[ 0.333 ]
method=[ c2c_radix4 ]      gputime=[ 8607.616 ] cputime=[ 8638.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2729.888 ] cputime=[ 2761.000 ] occupancy=[ 0.333 ]

method=[ solve_poisson ] gputime=[ 2762.656 ] cputime=[ 2794.000 ] occupancy=[ 0.667 ]

method=[ c2c_radix4 ]      gputime=[ 8514.720 ] cputime=[ 8547.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2724.192 ] cputime=[ 2760.000 ] occupancy=[ 0.333 ]
method=[ c2c_radix4 ]      gputime=[ 8620.064 ] cputime=[ 8652.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2773.920 ] cputime=[ 4270.000 ] occupancy=[ 0.333 ]

method=[ complex2real_scaled ] gputime=[ 1524.992 ] cputime=[ 1562.000 ] occupancy=[ 0.667 ]

method=[ memcopy ] gputime=[ 2468.288 ]
```


Performance improvement

	Non-pinned memory	Pinned memory
Initial implementation (r2c, poisson, c2r)	67ms (10.8ms)	63ms
+Shared memory +Fast integer mul	63.4ms (7.1ms)	59.4ms
+Coalesced read in c2r	62.1ms (5.8ms)	58.2ms

Tesla C870, pinned memory, optimized version: 10.4ms

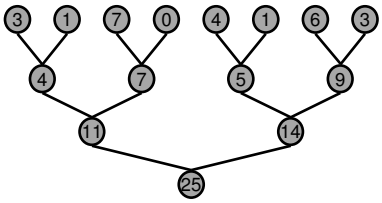
Parallel Reduction

Parallel Reduction

- Common and important data parallel primitive
- Easy to implement in CUDA
  - Harder to get it right
- Serves as a great optimization example
  - We'll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies

Parallel Reduction

- Tree-based approach used within each thread block

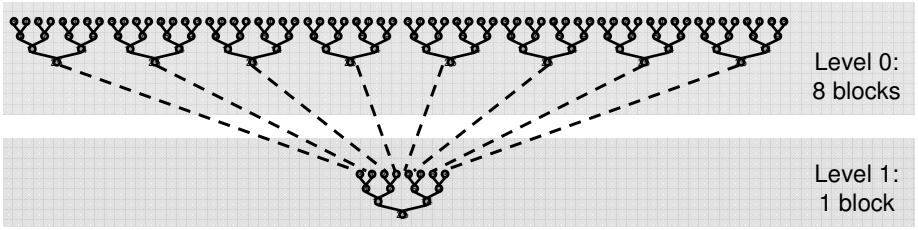


- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

The Global Synchronization "Myth"

- 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
- Problem: GPU has limited resources
  - GPU has *M* multiprocessors
  - Each multiprocessor can support a limited # of blocks, *b*
  - If total # blocks *B* > *M* * *b*… **DEADLOCK!**
- Also, GPUs rely on large amount of *independent* parallelism to cover memory latency
  - Global synchronization destroys independence

Solution: Kernel Decomposition



- Avoid global sync by decomposing computation into multiple kernel invocations
- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

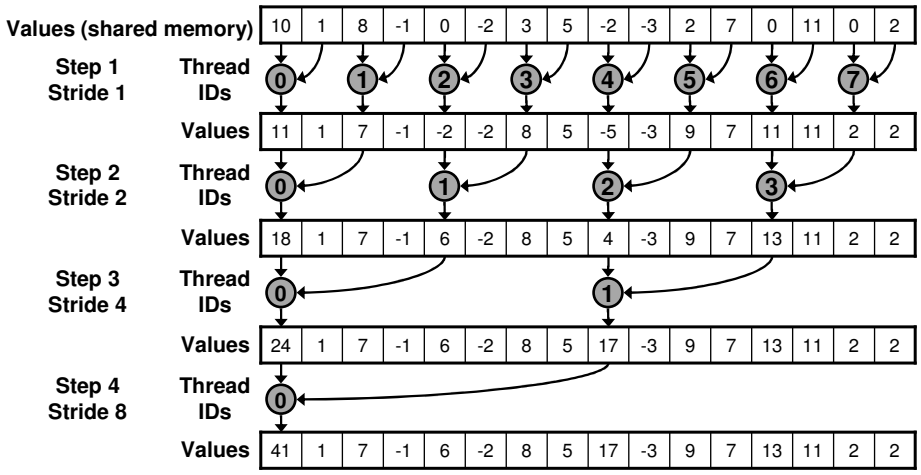
What is Our Optimization Goal?

- We should strive to reach GPU peak performance
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth
- Will use G80 GPU for this example
  - 384-bit memory interface, 900 MHz DDR
  - 384 * 1800 / 8 = 86.4 GB/s

Reduction #1: Interleaved Addressing

```
__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: Interleaved Addressing



Reduction #1: Interleaved Addressing

```
__global__ void reduce1(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];
}
```

Problem: highly divergent branching results in very poor performance!

Performance for 4M element reduction

Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

Note: Block Size = 128 threads for all tests

Reduction #2: Interleaved Addressing

Just replace divergent branch in inner loop:

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

With strided index and non-divergent branch:

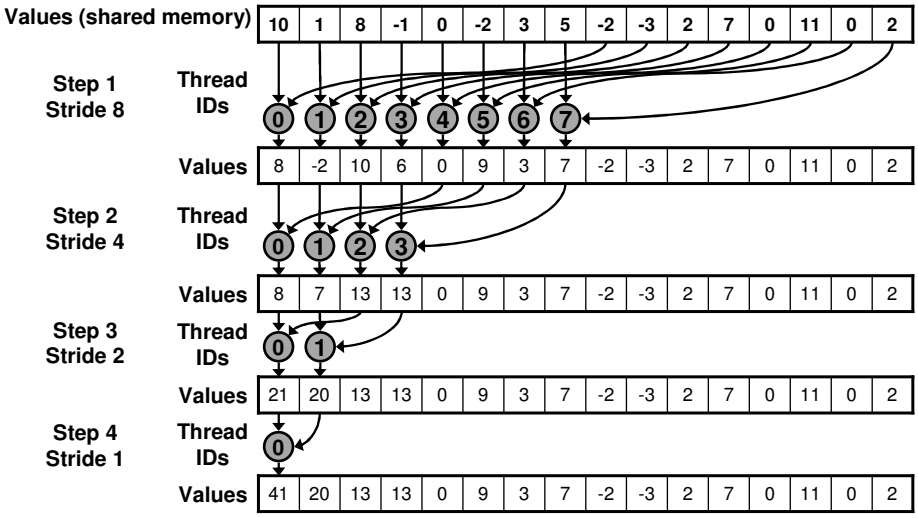
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
   int index = 2 * s * tid;

   if (index < blockDim.x) {
      sdata[index] += sdata[index + s];
   }
   __syncthreads();
}
```

New Problem: Shared Memory Bank Conflicts

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	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

Parallel Reduction: Sequential Addressing



Sequential addressing is conflict free

Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop:

```
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();
}
```

With reversed loop and threadID-based indexing:

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

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	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
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x

Idle Threads

Problem:

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

Half of the threads are idle on first loop iteration!

This is wasteful…

Reduction #4: First Add During Load

Halve the number of blocks, and replace single load:

```
// 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();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
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();
```


Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	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
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x

Instruction Bottleneck

- At 17 GB/s, we're far from bandwidth bound
  - And we know reduction has low arithmetic intensity
- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - In other words: address arithmetic and loop overhead
- Strategy: unroll loops

Unrolling the Last Warp

- 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

Reduction #5: Unroll the Last Warp

```
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];
}
```

Note: This saves useless work in *all* warps, not just the last one!
Without unrolling, all warps execute every iteration of the for loop and if statement

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	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
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x

Complete Unrolling

- 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

Unrolling with Templates

- Specify block size as a function template parameter:

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

Reduction #6: Completely Unrolled

```
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 (tid <  64)  { sdata[tid] += sdata[tid +   64]; } __syncthreads();
}

if (tid < 32) {
    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];
}
```

Note: all code in RED will be evaluated at compile time.

Results in a very efficient inner loop!

Invoking Template Kernels

- Don't we still need block size at compile time?
  - Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
    {
    case 512:
      reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
      reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
      reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
      reduce5<  64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
      reduce5<  32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
      reduce5<  16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
      reduce5<   8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
      reduce5<   4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
      reduce5<   2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
      reduce5<   1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	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
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x

Parallel Reduction Complexity

- Log(*N*) parallel steps, each step *S* does $N/2^S$ independent ops
  - Step Complexity is O(log *N*)
- For $N=2^D$, performs $\sum_{S\in[1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is O(*N*) – It is work-efficient
  - i.e. does not perform more operations than a sequential algorithm
- With *P* threads physically in parallel (*P* processors), time complexity is O(N/P + log *N*)
  - Compare to O(*N*) for sequential reduction
  - In a thread block, N=P, so O(log N)

What About *Cost?*

- *Cost* of a parallel algorithm is processors × time complexity
  - Allocate threads instead of processors: O(N) threads
  - Time complexity is O(log N), so *cost* is O(N log N) : not cost efficient!
- Brent's theorem suggests O(N/log N) threads
  - Each thread does O(log N) sequential work
  - Then all O(N/log N) threads cooperate for O(log N) steps
  - Cost = O((N/log N) * log N) = O(N)
- Known as *algorithm cascading*
  - Can lead to significant speedups in practice

Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- Brent's theorem says each thread should sum O(log n) elements
  - i.e. 1024 or 2048 elements per block vs. 256
- In my experience, beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks
- On G80, best perf with 64-256 blocks of 128 threads
  - 1024-4096 elements per thread

Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```
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();
```

With a while loop to add as many as necessary:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

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


Performance for 4M element reduction

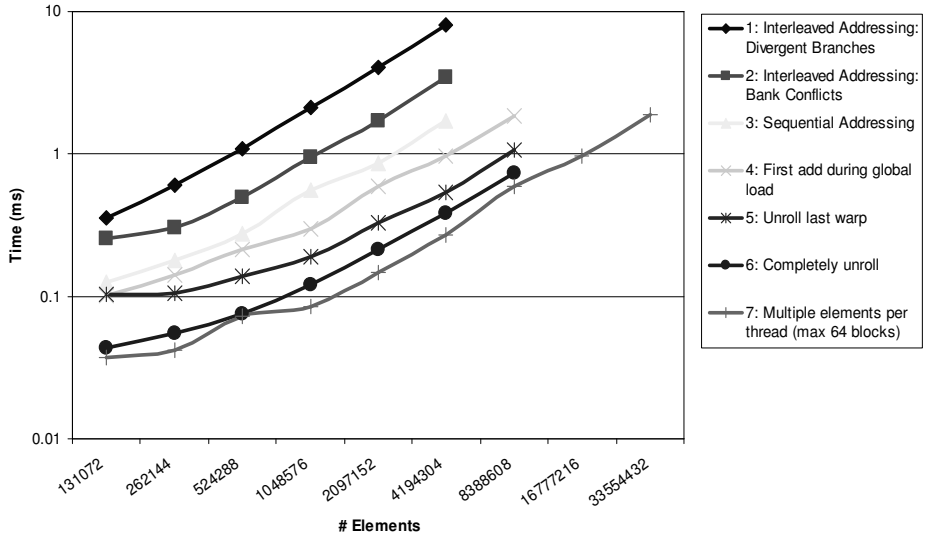
	Time (2^{22} ints)	Bandwidth	Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s	
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x
Kernel 4: first add during global load	0.965 ms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

Kernel 7 on 32M elements: 72 GB/s! **Total Speedup: 30x!**

```
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;

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

    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 (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

    if (tid < 32) {
        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];
    }

    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Final Optimized Kernel

Performance Comparison



NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com