Review GPU Computing

Hartwig Anzt
General Purpose Computing on GPUs

• properties of GPUs
  - extremely high computing power
  - high bandwidth
  - high parallelism
  - small caches

• why
  - traditional use for graphics
  - no dependencies when updating pixels on a screen - but need to update many pixels at the same time

many lightweight cores
General Purpose Computing on GPUs

- suitable for highly parallel applications with:
  - SIMD operations
  - no data dependencies
  - uniform memory access
  - no branching (if…)

- high speedup potential compared to CPU code for:
  - Monte-Carlo simulations
  - MD code
  - statistics
  - dense linear algebra
  - AXPY, DOT, stencils…
General Purpose Computing on GPUs

- what about the memory wall?
  - on CPUs: sophisticated memory hierarchy

- on GPUs: fast switching between threads hides latencies
  - latencies: 100… 1000 … 10000 cycles
  - execute operation on data present
  - many threads have to be active
Performance vs. Algorithm Complexity

- Regular data structures
- Explicit time stepping
- Simple update rules

- Unstructured mesh
- Implicit solvers
- Adaptive alg.
PCI Bottleneck

- CPU
  - processing elements
  - cache: 40 GB/s
- co-processor
- system memory: 50 GB/s
- device memory: 288 GB/s
- PCI: 4 GB/s
NVIDIA CUDA

C extension to program an NVIDIA GPU:
  • program execution on the GPU multi-processors
  • target highly-parallel algorithms

abstraction layer:
  • hierarchy of thread-blocks
  • shared memory for communication between threads in the same block
  • barrier synchronization
  • scales up to hundreds of cores/thousands of threads

focus on parallel algorithm, not hardware properties
NVIDIA CUDA

- kernel written as program for one thread
- all threads execute that code
- every thread is executed by one core
- threads are gathered in thread blocks (up to 3D)
- every thread block is assigned to a multiprocessor (SM)
- 32 thread=1warp are executed in parallel
- 16 threads parallel data access
- many thread-blocks can be active
- a kernel is executed by a (up to 3D) grid of thread blocks
Input and Output arrays

Single-threaded:
input and output array may overlap

multi-threaded:
overlap may result in read/write conflicts
GPUs have no fixed execution order for thread-blocks
CUDA execution scheme

- thread
- thread-block
- grid of thread-blocks

Localization via unique IDs:
- threadIdx.x = 0 … 11
- threadIdx.y = 0 … 1
- blockDim.x = 12
- blockDim.y = 2
- blockIdx.x = 0 … 2
- blockIdx.y = 0 … 1
- gridDim.x = 3
- gridDim.y = 2
CUDA execution scheme

gridDim.x = 3
gridDim.y = 2

blockDim.x = 4
blockDim.y = 3

blockIdx.x = 0 ... 2
blockIdx.y = 0 ... 1

threadIdx.x = 0 ... 3
threadIdx.y = 0 ... 2
CUDA kernel example

```c
__global__ void kernel(int* a) {
    a[blockIdx.x*blockDim.x + threadIdx.x]=0;
}

int main() {
    . . .
    dim3 block(4);
    dim3 grid(n/block.x);
    kernel<<<grid,block>>>(d_a);
    . . .
    return 0;
}
```
CUDA kernel example: hands-on

```c
__global__ void kernel(int* a)
{ a[blockIdx.x*blockDim.x + threadIdx.x]=7;}
Output: 7777777777777777

__global__ void kernel(int* a)
{ a[blockIdx.x*blockDim.x + threadIdx.x]=blockIdx.x;}
Output: 0000111122223333

__global__ void kernel(int* a)
{ a[blockIdx.x*blockDim.x + threadIdx.x]=threadIdx.x;}
Output: 0123012301230123
thread-block and grid configuration

- **1D data of size N**: linear arrangement of threads
  - dim3 block(b,1,1), grid(g,1,1) with b*g=N
  - idx = blockIdx.x*blockDim.x + threadIdx.x
  - kernel<<<grid, block>>>
  - kernel<<<g, b>>>

- **2D data of size NxM**: rectangular arrangement of threads
  - dim3 block= (b,c,1), grid=(g,h,1) with b*g=N und c*h=M
  - idx = blockIdx.x*blockDim.x + threadIdx.x (global 2D)
  - idy = blockIdx.y*blockDim.y + threadIdx.y (global 2D)
  - id = idy*N + idx (global 1D)
  - kernel<<<grid, block>>>>
CUDA kernel example: saxpy

• sequential / CPU

```c
void saxpy_serial(int n, float a, float *x, float *y){
    for (int i = 0; i < n; i++)
        y[i] = a*x[i] + y[i];
}
saxpy_serial(n, 2.0, x, y);
```

• parallel / CUDA

```c
__global__ void saxpy_parallel(int n, float a, float *x, float *y){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```
CUDA kernel example: saxpy

- high parallelism
  - no dependencies
  - no communication
  - every thread handles one vector component

- good GPU usage?
  - yes, uniform operation on large data
  - yes, no branching (if…)
  - yes, no synchronization / no communication
  - no, few computations on data
Performance Bounds

execution time $T_R \geq \max \{ T_C , T_T \}$

- $T_C$ … computation time
- $T_T$ … data transfer / communication time
- classification between computation-bound and communication-bound algorithms

• algorithm characteristics
  - $f$ … number of floating point operations = 2N for saxpy
  - $w$ … data size (words) = 3N+1 for saxpy (R/W)
  - $f / w$ … compute intensity = 2/3 for saxpy

• hardware characteristics
  - $L$ … max compute performance (in GFlop/s) = 1040 GFlop/s (GT750 M)
  - $B$ … max bandwidth (in GByte/s) = 80 GB/s (GT750 M)
Performance Bounds

• lower bounds
  - \( T_c \geq \frac{f}{L} \) and \( T_T \geq \frac{4w}{B} \) (single precision)

• actual performance
  - \( L_{eff} = \frac{f}{T_R} \leq \min \{ L, \frac{f B}{4w} \} \)

• for high performance: \( L_{eff} \leq \frac{f B}{4w} \)
  - ratio \( f/w \) defines compute intensity

• for BLAS 3 (matrix matrix multiplication)
  - \( f/w \) is in order \( O(N)! \) (matrix dimension \( N \times N \))
  - bandwidth not an issue for large problems!

• for saxpy
  - \( L_{eff} \leq \frac{B}{4 \times 3} \times 2 \text{ flop/byte} = 13 \text{ Gflop/s} = 1.3 \% \text{ Peak}! \)
  - experimental results … !
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

cudaEventRecord(start);
    saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);

float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
__global__ void
sgemv_rowmajor(int n, float a, float *m, float *x, float *y){

    int row = blockIdx.x*blockDim.x + threadIdx.x;
    float sum = 0.0;

    if (row < n){
        for( int col=0; col<n; col++){
            sum += m[row*n+col] * x[i];
        }
        y[row] = a*sum;
    }
}

int nblocks = (n + 255) / 256;
sgemv_rowmajor<<<nblocks, 256>>>(n, 2.0, x, y);
___global__ void
sgemv_colmajor(int n, float a, float *m, float *x, float *y){

    int row = blockIdx.x*blockDim.x + threadIdx.x;
    float sum = 0.0;

    if (row < n){
        for( int col=0; col<n; col++){
            sum+= m[col*n+row] * x[i];
        }
        y[row] = a*sum;
    }
}

int nblocks = (n + 255) / 256;
sgemv_colmajor<<<nblocks, 256>>>(n, 2.0, x, y);
example program

• 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array
example program

• 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

• initial approach
  • 7 kernels, each adding one element to the sum
  • data always read from main memory
example program

__global__ void kernel_add(int n, int offset, int *a, int *b)
{
    int i = blockDim.x*blockIdx.x+threadIdx.x;
    int j = i + offset;
    if( j>-1 && j<n ){
        b[i]+=a[j];
    }
}

int nbblocks = (n + 255) / 256;
kernel_add<<<nbblocks, 256>>>(n, -3, a, b);
kernel_add<<<nbblocks, 256>>>(n, -2, a, b);
kernel_add<<<nbblocks, 256>>>(n, -1, a, b);
kernel_add<<<nbblocks, 256>>>(n, -0, a, b);
kernel_add<<<nbblocks, 256>>>(n, 1, a, b);
kernel_add<<<nbblocks, 256>>>(n, 2, a, b);
kernel_add<<<nbblocks, 256>>>(n, 3, a, b);
example program

• 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

• initial approach
  • 7 kernels, each adding one element to the sum
  • data always read from main memory

• better approach
  • merge the 7 kernels into one
__global__ void kernel2(int n, int *a, int *b)
{
    int i = blockDim.x*blockIdx.x+threadIdx.x;
    if( i<n ){
        if(i>2)
            b[i]+=a[i-3];
        if(i>1)
            b[i]+=a[i-2];
        if(i>0)
            b[i]+=a[i-1];
        b[i]+=a[i];
        if(i<n-3)
            b[i]+=a[i+3];
        if(i<n-2)
            b[i]+=a[i+2];
        if(i<n-1)
            b[i]+=a[i+1];
    }
}
example program

• 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

• initial approach
  • 7 kernels, each adding one element to the sum
  • data always read from main memory

• better approach
  • merge the 7 kernels into one

• even better
  - one thread reads needed data into shared memory
  - every thread-block computes blockDim.x partial sums
  - data read from shared memory
example program

• 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

• initial approach
  • 7 kernels, each adding one element to the sum
  • data always read from main memory

• better approach
  • merge the 7 kernels into one

• even better
  - one thread reads needed data into shared memory
  - every thread-block computes blockDim.x partial sums
  - data read from shared memory

• even better
  - every thread reads one entry into shared memory
  - every thread-block computes blockDim.x-6 partial sums
  - data read from shared memory
```c
__global__ void kernel3(int n, int *a, int *b)
{
    int i = (blockDim.x-6)*blockIdx.x+threadIdx.x-3;
    int idx = threadIdx.x;
    __shared__ int values[256];
    int sum = 0;

    values[idx] = (i>-1 && i<n) ? a[i] : 0;

    if( idx>2 && idx<256-3 ){
        for( int j=-3; j<4; j++ )
            sum += values[idx+j];

        b[i]=sum;
    }
}
```
```c
__global__ void
sgemv_rowmajor(int n, float a, float *m, float *x, float *y){
    int row = blockIdx.x*blockDim.x + threadIdx.x;
    float sum = 0.0;
    if (row < n){
        for( int col=0; col<n; col++){
            sum+= m[row*n+col] * x[col];
        }
        y[row] = a*sum;
    }
}

int nblocks = (n + 255) / 256;
sgemv_rowmajor<<<nblocks, 256>>>(n, 2.0, x, y);
```
__global__ void
sgemv_colmajor(int n, float a, float *m, float *x, float *y){

    int row = blockIdx.x*blockDim.x + threadIdx.x;
    float sum = 0.0;

    if (row < n){
        for( int col=0; col<n; col++){
            sum+= m[col*n+row] * x[i];
        }
        y[row] = a*sum;
    }
}

int nblocks = (n + 255) / 256;
sgemv_colmajor<<<nblocks, 256>>>(n, 2.0, x, y);
CUBLAS library

• BLAS-library for NVIDIA CUDA GPUs
  - BLAS= Basic Linear Algebra Subroutines

• typical workflow
  - matrices and vectors are managed on host, then copied to device
  - CUBLAS-functions execute computations on GPU
  - results are copied back to host

• advantages:
  - optimal kernel implementation
  - easy out-of-the-box usage

• disadvantages:
  - multiple data transfers / communication
CUBLAS library

- level 1 BLAS (vector-vector-operations)
  - cublasSaxpy, cublasScopy, cublasSdot, cublasSnrm2
  - cublasSscal, cublasIsamax, …

- level 2 BLAS (matrix-vector-operations)
  - cublasSgemv, cublasSsymv, …

- level 3 BLAS (matrix-matrix-operations)
  - cublasSgemm, cublasSsymm, …
#include<cublas.h>

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

    // restart residual after 50 iterations
    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);
    ...
}
Key Features of MAGMA 1.6

HYBRID ALGORITHMS

MAGMA uses hybrid algorithms where the computation is split into tasks of varying granularity and their execution scheduled over the hardware components. Scheduling can be static or dynamic. In either case, small non-parallelizable tasks, often on the critical path, are scheduled on the CPU, and larger more parallelizable ones, often Level 3 BLAS, are scheduled on the MICs.

PERFORMANCE & ENERGY EFFICIENCY
**MAGMA SPARSE**

**ROUTINES**  
CG, GMRES, BiCGSTAB, LOBPCG, Iterative refinement

**PRECONDITIONERS**  
ILU / IC, Jacobi

**KERNELS**  
SpMV, SpMM

**DATA FORMATS**  
CSR, ELL, SELL-P

Sparse matrix - vector product (SpMV) in double precision arithmetic

SpMM on block of 32 vectors in double precision

---

**The University of Florida Sparse Matrix Collection**  
http://www.cise.ufl.edu/research/sparse/matrices/

**GPU**  
NVIDIA K40 (Atlas)  
15 MP x 192 @ 0.88 GHz

**CPU**  
Intel Xeon ES-2670 (Sandy Bridge)  
2 x 8 cores @ 2.60 GHz
sparse matrices

- very few non-zeros
- storage and computational effort can be decreased by formats storing only non-zeros explicitly
- CSR format

\[
A = \begin{pmatrix}
5.4 & 1.1 & 0 & 0 & 0 & 0 \\
0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\
0 & 0 & 1.1 & 0 & 0 & 0 \\
0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\
9.0 & 0 & 0 & 1.1 & 4.5 & 0 \\
1.1 & 0 & 2.9 & 3.7 & 0 & 1.1
\end{pmatrix}
\]

rowptr: (0 2 5 6 9 12 16)

colind: (0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5)

values: (5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1)
matrix-vector multiplication

```c
__global__ void
sgemv_colmajor(int n, float a, float *m, float *x, float *y){
    int row = blockIdx.x*blockDim.x + threadIdx.x;
    float sum = 0.0;
    if (row < n){
        for( int col=0; col<n; col++){
            sum+= m[col*n+row] * x[i];
        }
        y[row] = a*sum;
    }
}

int nblocks = (n + 255) / 256;
sgemv_colmajor<<<nblocks, 256>>>(n, 2.0, x, y);
```

aligned memory access important for performance
sparse matrices

- very few non-zeros
- storage and computational effort can be decreased by formats storing only non-zeros explicitly
- CSR format

- conversion pays off if many matrix-vector multiplications are needed - e.g. an iterative solver
- conversion should be implemented on the GPU to avoid data transfers via PCI
conversion to CSR

- count non-zeros in the matrix
- allocate memory
- copy non-zeros into new data structures
- fill column-indices
- fill row pointer
conversion to CSR

- count non-zeros in the matrix
- allocate memory
- copy non-zeros into new data structures
- fill column-indices
- fill row pointer
conversion to CSR

- count non-zeros in the matrix
  - first approach: one thread counts all non-zeros
conversion to CSR

- **count non-zeros in the matrix**
  - first approach: one thread counts all non-zeros
  - better: non-zeros in the different rows are counted in parallel, then one thread adds the partial sums
conversion to CSR

- count non-zeros in the matrix
  - first approach: one thread counts all non-zeros
  - better: non-zeros in the different rows are counted in parallel, then one thread adds the partial sums
  - even better: non-zeros in the different rows are counted in parallel, then a global reduction phase forms overall sum
conversion to CSR

- count non-zeros in the matrix
  - first approach: one thread counts all non-zeros
  - better: non-zeros in the different rows are counted in parallel, then one thread adds the partial sums
  - even better: non-zeros in the different rows are counted in parallel, then a global reduction phase forms overall sum

Map-Reduce scheme

- map the data to the computing resources
- collect the results in a reduction operation
conversion to CSR

- count non-zeros in the matrix
  - first approach: one thread counts all non-zeros
  - better: non-zeros in the different rows are counted in parallel, then one thread adds the partial sums
  - even better: non-zeros in the different rows are counted in parallel, then a global reduction phase forms overall sum

Map-Reduce scheme

- map the data to the computing resources
- collect the results in a reduction operation

Tutorial:
Parallel reduction on GPUs

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
different parallel reduction schemes I

Problem: Warps are divergent
different parallel reduction schemes II

problem: bank conflicts
different parallel reduction schemes III
example program
example program