MD-CUDA

Presented by
Wes Toland
Syed Nabeel
Outline

- Objectives
- Project Organization
- CPU
- GPU
- GPGPU
- CUDA
- N-body problem
- MD on CUDA
- Evaluation
- Future Work
Objectives

- Understand molecular dynamics (MD) simulations
- Analyze various algorithms for parallelizing MD simulations
- Study GPGPU architectures
- Learn the CUDA programming model and API
- Port existing parallel MD code to parallel GPGPU code
- Evaluate the scalability of our parallel GPGPU MD application
Objectives

Project Organization

CPU

GPU

GPGPU

CUDA

N-body problem

MD on CUDA

Evaluation

Future Work
Project Organization

- Research parallel MD algorithms
- Research parallel GPGPU MD algorithms
- Install CUDA drivers
- Install CUDA API
- Use N-body work-distributing framework to make a parallel MD code
- Evaluation
Problems encountered (1/2)

- Installing the CUDA API took some time because it required several packages that had specific glibc dependencies.
- This was solved by installing the following modules via YaSt (for SuSe platforms):
  - gcc
  - gcc-++
  - freeglut-devel
  - glibc-devel-32bit
  - kernel-source
Problems Encountered (2/2)

- Once the CUDA APIs were successfully installed, we encountered runtime errors when attempting to run Nvidia benchmarks.
- The correct Nvidia GPU kernel module was not installed.
- **Solution**: Download and install the kernel module for the Nvidia Quadro FX 5600 GPU driver.
- Objectives
- Project Organization
- **CPU**
- GPU
- GPGPU
- CUDA
- N-body problem
- MD on CUDA
- Evaluation
- Future Work
CPU Instruction Level Parallelism (ILP)

- Instructions are re-ordered and combined into groups
- The groups of instructions are then executed in parallel without changing the result of the program
- Modern processors have multi-stage instruction pipelines
- Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage
CPU Limitations

- ILP is increasingly difficult to extract from the instruction stream
- Control hardware dominates microprocessors
  - Complex, difficult to build and verify
  - Takes substantial fraction of die
  - Scales poorly
  - Control hardware does not do any math
Objectives

Project Organization

CPU

GPU

GPGPU

CUDA

N-body problem

MD on CUDA

Evaluation

Future Work
GPU

- Type of CPU attached to a Graphics card dedicated to calculating floating point operations
- Incorporates custom microchips which contain special mathematical operations
- Stream Processing: applications can use multiple computational units without explicitly managing allocation, synchronization, or communication among those units.
GPU Limitations

- High learning curve
- Programming model of most graphics processors is inadequate for most non-graphics applications/application programmers
- Limitations in writing and scattering information
- DRAM memory bandwidth bottleneck
Objectives
Project Organization
CPU
GPU
GPGPU
CUDA
N-body problem
MD on CUDA
Evaluation
Future Work
GPGPU

- General-purpose computing on GPU
- GPU is viewed as a compute device capable of executing a large number of threads in parallel
- GPU operates as a co-processor to the main CPU
- Portions of code that can be parallelized and are computationally intensive can be off-loaded onto the GPU
GPGPU Applications

- Physical modeling
- Computational engineering
- Game effects (FX) physics
- Image processing
- Matrix algebra
- Convolution
- Correlation
- Sorting
GPGPU Challenges

- Unintuitive graphics API
- Confusing addressing modes
- Shader capabilities are limited
- Limited instruction sets
- Limited communication
  - Explicit data transfer is typically required
Objectives
Project Organization
CPU
GPU
GPGPU
CUDA
N-body problem
MD on CUDA
Evaluation
Future Work
CUDA to the Rescue

- **CUDA**: Compute Unified Device Architecture
- **CUDA** manages GPU computations with a parallel model similar to certain CPU paradigms.
  - User does not need to map to a graphics API
- **Layers**
  - Hardware driver
  - Application programming interface (API)
  - CUDA runtime
  - CUFFT
  - CUBLAS
CUDA Layers
GPU Architecture in CUDA

Memory Addressing Modes

- General DRAM Memory Addressing
- Shared Memory Addressing
General DRAM Memory Addressing

- More programming flexibility than GPUs
- Ability to read and write data at any location in DRAM, just like on a CPU
**Shared Memory**

- Parallel data cache
- Very fast general read and write access
- Minimized over-fetch and round-trips to DRAM
GPU as a Computation Device

- Certain parallelizable or computationally intensive portions of code are executed on multi-threaded GPU
- **Kernel**: the common function compiled to the instruction set of the device and downloaded to it
- **Separate DRAM for host and device**
  - Host Memory
  - Device Memory
A batch of threads that operates with a limited amount of shared memory

Synchronization points are used to coordinate shared memory access

Each thread knows its 1D, 2D, or 3D thread id
Grid Of Thread Blocks

- A set of blocks of the same size executing the same kernel
- Single block has a limited number of threads
- Much more threads available in the grid
- Inter-thread is avoided
- Each block is identified by its 1D, 2D, or 3D block id
CUDA Memory Model

- Read-write per-thread registers
- Read-write per-thread local memory
- Read-write per-block shared memory
- Read-write per-grid global memory
- Read-only per-grid constant memory
- Read-only per-grid texture memory
A set of SIMD multi-processors with on-chip shared memory
SIMD behavior through groups of threads called **warps**

One or more thread blocks are executed on each multi-processor using **time-slicing**

The **issue order of the warps** within a block is undefined

The **issue order of the blocks** within a grid of thread blocks is undefined
Objectives

Project Organization

CPU

GPU

GPGPU

CUDA

N-body problem

MD on CUDA

Evaluation

Future Work
N-body Problem

Numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body

- **Astrophysical** simulation-the bodies attract each other through the gravitational force
- **Protein folding** simulations-to calculate electrostatic and van der Waals forces
- **Turbulent fluid flow** simulation
- **Global illumination computation** in computer graphics
All-pairs Approach to N-body Simulation

- A brute-force technique to evaluate all pair-wise interactions among N bodies
  - Relatively simple method
  - $O(N^2)$ computational complexity
- Typical example of routine that is used as a kernel to determine the forces in close-range interactions
Accelerating All-pairs Approach

- All-pairs component requires substantial time to compute → target for acceleration
- Optimal reuse of data
  - Computation arranged in tiles
  - Interactions in each row are evaluated in sequential order, updating the acceleration vector
  - Separate rows are evaluated in parallel
Given $N$ bodies with an

- Initial position $x_i$
- Velocity $v_i$ for $1 \leq i \leq N$
- Force vector $f_{ij}$ on body $i$ caused by its gravitational attraction to body $j$

\[
f_{ij} = G \frac{m_i m_j}{\|r_{ij}\|^2} \cdot \frac{r_{ij}}{\|r_{ij}\|},
\]

- $m_i$ and $m_j$ --> the masses of bodies $i$ and $j$
- $r_{ij} = x_j - x_i$ is the vector from body $i$ to body $j$
- $G$ is the gravitational constant
Total Force

- The total force $F_i$ on body $i$, due to its interactions with the other $N - 1$ bodies, is obtained by summing all interactions:

$$F_i = \sum_{1 \leq j \leq N, j \neq i} f_{ij} = Gm_i \cdot \sum_{1 \leq j \leq N, j \neq i} \frac{m_j r_{ij}}{\|r_{ij}\|^3}.$$
As bodies approach each other, the force between them grows without bound, which is an undesirable situation for numerical integration.

Softening factor $\varepsilon^2 > 0$ is added.

Softening factor enforces a limit to the magnitude of the force between the bodies.
Acceleration Calculation

- To integrate over time, we need the acceleration $a_i = F_i/m_i$ to update the position and velocity of body $i$.

$$a_i \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{\left\| \mathbf{r}_{ij} \right\|^2 + \varepsilon^2}^{3/2}.$$  

- The integrator used to update the positions and velocities is a **Leapfrog-Verlet** integrator.
Objectives
Project Organization
CPU
GPU
GPGPU
CUDA
N-body problem
MD on CUDA
Evaluation
Future Work
Code Organization (1/3)

- **C++ Files (CPU)**
  - `md.cpp`
    - Main function
    - Instantiate `mdsystemcuda` object
    - Use `mdsystemcuda` object to make CUDA calls to initialize CPU and GPU memory and run CUDA kernel
  - `mdsystemcuda.cpp`
    - Provides functionality to initialize CPU and GPU memory
    - Has several wrapper functions for calls to the CUDA kernel
CUDA Files (GPU)

- mdsystemcuda.cu
  - Provides functions to copy the position and velocity arrays to GPU memory
  - integrateMDSystem is the top-level kernel function called by an mdsystemcuda object. This function calls the lower-level kernel function, integrateBodies.

- md_kernel.cu
  - Contains the integrateBodies function, which updates the position and velocity arrays
Code Organization (3/3)

Input Files
- md.in
  - InitUcell[3]
    - Determines N, the number of bodies
  - StepLimit StepAverage
    - Determines # of iterations
- p
  - N/p is the number of thread blocks
- q
  - Threads per body
Compiling MD-CUDA (1/2)

- Standard Libraries: FFT, BLAS, ...
- Integrated CPU and GPU C Source Code
- NVIDIA C Compiler
- NVIDIA Assembly for Computing
- CPU Host Code
- CUDA Runtime & Driver
- Profiler
Compiling MD-CUDA (2/2)

 Compile C++ files:

 - `g++ $(CFLAGS)-o mdsystemcuda.cpp_o -c mdsystemcuda.cpp`
 - `g++ $(CFLAGS)-o md.cpp_o -c md.cpp`

 Compile CUDA files:

 - `nvcc $(CFLAGS)-o mdsystemcuda.cu_o -c mdsystemcuda.cu`

 Compile final executable:

 - `g++ -fPIC -o MD-CUDA mdsystemcuda.cpp_o md.cpp_o mdsystemcuda.cu_o $(LDFLAGS) -lcudart -lcutil`
CUDA Implementation of N-body

- All-pairs algorithm calculates each entry $f_{ij}$ in an $N \times N$ grid of all pair-wise forces.
- The total force $F_i$ (or acceleration $a_i$) on body $i$ is obtained from the sum of all entries in row $i$.
- Each entry can be computed independently.
- $O(N^2)$ available parallelism.
- Requires $O(N^2)$ memory.
- Would be substantially limited by memory bandwidth.
Serialize some of the computations

- Achieves the data reuse needed to reach peak performance of the arithmetic units
- Reduces the memory bandwidth required

Computational tile: a square region of the grid of pair-wise forces consisting of $p$ rows and $p$ columns
Computational Tile(2/3)

- Only $2p$ body descriptions are required to evaluate all $p^2$ interactions in the tile
  - $p$ descriptions can be reused later
- These body descriptions can be stored in shared memory or in registers
- The total effect of the interactions in the tile on the $p$ bodies is captured as an update to $p$ acceleration vectors
For optimal data reuse, tile computation is organized so that:

- The interactions in each row are evaluated in sequential order, updating the acceleration vector.
- The separate rows are evaluated in parallel.
Body-Body Force Calculation

- The interaction between a pair of bodies is implemented as an entirely serial computation.
- bodyBodyInteraction function does the following:
  - Computes the force on body i from its interaction with body j
  - Updates acceleration $a_i$ of body i as a result of this interaction.
- FLOPS in bodyBodyInteraction: 20
  - Additions
  - Multiplications
  - Sqrtf() call
  - Division (or reciprocal)
```c
__device__ float3 bodyBodyInteraction(float3 ai, float4 bi, float4 bj)
{
    float3 r;
    r.x = bi.x - bj.x;
    r.y = bi.y - bj.y;
    r.z = bi.z - bj.z;

    float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
    distSqr += softeningSquared;

    float invDist = 1.0f / sqrtf(distSqr);
    float invDistCube = invDist * invDist * invDist;

    float s = bj.w * invDistCube;
    ai.x += r.x * s;
    ai.y += r.y * s;
    ai.z += r.z * s;

    return ai;
}
```

**float4 Data Type**

- Data type is for accelerations stored in GPU device memory
- Allows **coalesced** memory to access arrays of data in device memory.
  - Results in more efficient memory requests and transfers.
- Each body’s mass is stored in the w field of the body’s float4 position
- 3D vectors are stored as float3 variables
  - Register space is an issue and coalesced access is not
Tile Calculation

- A tile is evaluated by $p$ threads performing the same sequence of operations on different data.
- Each thread updates the acceleration of one body as a result of its interaction with $p$ other bodies.
- Load $p$ body descriptions from the GPU device memory into the shared memory provided to each thread block.
- Each thread in the block evaluates $p$ successive interactions.
- The result of the tile calculation is $p$ updated accelerations.
Tile Calculation

```c
__device__ float3
tile_calculation(float4 myPosition, float3 accel)
{
    int i;
    extern __shared__ float4[] shPosition;

    for (i = 0; i < blockDim.x; i++) {
        accel = bodyBodyInteraction(myPosition, shPosition[i], accel);
    }
    return accel;
}
```

- Each of the \( p \) threads
  - executes the function body in parallel
  - iterates over the same \( p \) bodies
  - computes the acceleration of its individual body as a result of interaction with \( p \) other bodies
Clustering Tiles into Thread Blocks

- **Thread block** has $p$ threads that execute some number of tiles in sequence
  - Sized to balance parallelism with data reuse
  - Degree of parallelism (i.e. the number of rows) must be sufficiently large to interleave multiple warps (to hide latencies in interaction evaluation)

- Amount of data reuse grows with the number of columns
  - It also governs the size of the transfer of bodies from device memory into shared memory

- The **size of the tile** also determines the register space and shared memory required
Thread Blocks for N-Body Implementation (1/3)

- For this implementation tiles are square of size $p$ by $p$
- Before executing a tile:
  - Each thread fetches one body into shared memory
  - Threads synchronize after the fetch
- Consequently, each tile starts with $p$ successive bodies in the shared memory
Thread Blocks for N-Body Implementation (2/3)

- Time spans the horizontal direction
- Parallelism spans the vertical direction
- Heavy lines demarcate the tiles of computation showing
  - Where shared memory is loaded
  - Where a barrier synchronization is performed
- In a thread block:
  - There are $N/p$ tiles
  - $p$ threads computing the forces on $p$ bodies (one thread per body).
  - Each thread computes all $N$ interactions for one body
Multiple threads work from left to right

Synchronization takes place at the end of each tile of computation
CUDA Kernel Code To Calculate N-body Forces For A Thread Block

```c
__global__ void calculate_forces(void *devX, void *devA)
{
    extern __shared__ float4[] shPosition;

    float4 *globalX = (float4 *)devX;
    float4 *globalA = (float4 *)devA;
    float4 myPosition;
    int i, tile;
    float3 acc = {0.0f, 0.0f, 0.0f};
    int gtid = blockIdx.x * blockDim.x + threadIdx.x;

    myPosition = globalX[gtid];

    for (i = 0, tile = 0; i < N; i += p, tile++) {
        int idx = tile * blockDim.x + threadIdx.x;
        shPosition[threadIdx.x] = globalX[idx];
        __syncthreads();
        acc = tile_calculation(myPosition, acc);
        __syncthreads();
    }

    // Save the result in global memory for the integration step.
    float4 acc4 = {acc.x, acc.y, acc.z, 0.0f};
    globalA[gtid] = acc4;
}
```

- Pointers to global device memory for the positions `devX` and the accelerations `devA` of the bodies.
- Input parameters assigned to local pointers with type conversion so they can be indexed as arrays.
- Ensures that all shared memory locations are populated before the gravitation computation proceeds.
- Ensures that all threads finish their gravitation computation before advancing to the next tile.
Defining a Grid of Thread Blocks

- The kernel program in previous section calculates the acceleration of $p$ bodies in a system, caused by their interaction with all $N$ bodies in the system.
- The kernel invoked on a grid of thread blocks to compute the acceleration of all $N$ bodies.
- There are $p$ threads per block and one thread per body.
- Number of thread blocks needed to complete all $N$ bodies is $N/p$.
- Define a 1D grid of size $N/p$.
- Result is a total of $N$ threads that perform $N$ force calculations each, for a total of $N^2$ interactions.
Evaluation of the Full Grid of Interactions

- **Vertical** dimension shows the parallelism of the 1D grid of $N/p$ independent thread blocks with $p$ threads each.

- **Horizontal** dimension shows the sequential processing of $N$ force calculations in each thread.

- A thread block reloads its shared memory every $p$ steps to share $p$ positions of data.
void runBenchmark(int iterations, bool benchmark)
{
    /* Prime the GPU */
    md->update(activeParams.m_timestep);

    CUT_SAFE_CALL(cutStartTimer(timer));
    for (int i = 0; i < iterations; ++i)
    {
        md->update(activeParams.m_timestep);

        if(!benchmark) {
            float *pos = md->getArray(MDSystem::MDSYSTEMPOSITION);
            float *vel = md->getArray(MDSYSTEMVELOCITY);
        }
    }

    md->synchronizeThreads();
    CUT_SAFE_CALL(cutStopTimer(timer));

    float milliseconds = cutGetTimerValue(timer);
    double interactionsPerSecond = 0;
    double gflops = 0;
    computePerfStats(interactionsPerSecond, gflops, milliseconds, iterations);
    displayPerfStats(iterations, milliseconds, interactionsPerSecond, gflops, 20);
}
```cpp
void MDSystemCUDA::update(float deltaTime)
{
    assert(m_bInitialized);

    CUT_SAFE_CALL( cutStartTimer(m_timer));

    integrateMDSystem(m_dPos[m_currentWrite], m_dVel[m_currentWrite],
                      m_dPos[m_currentRead], m_dVel[m_currentRead],
                      deltaTime, m_damping, m_numBodies, m_p, m_q);

    CUT_SAFE_CALL( cutStopTimer(m_timer));

    std::swap(m_currentRead, m_currentWrite);

    CUT_SAFE_CALL( cutResetTimer(m_timer));
}
```
Calls to a **global** function must specify the execution configuration

There is an expression with the following form between the function name and the args list:

\[ <<< \text{Dg, Db, Ns, S} >>> \]

- \( \text{Dg} \) is of type `dim3` and specifies the dimension and size of the grid
- \( \text{Db} \) is of type `dim3` and specifies the dimension and size of each block such that
  - \( \text{Db.x} \times \text{Db.y} \times \text{Db.z} \) equals the number of threads per block
- \( \text{Ns} \) is of type `size_t`. It specifies the number of bytes of shared memory that is dynamically allocated per block.
  - \( \text{Ns} \) is an optional argument which defaults to 0
- \( \text{S} \) is of type `cudaStream_t` and specifies the associated stream
```
void integrateMDSystem(float* newPos, float* newVel, float* oldPos, float* oldVel,
    float deltaTime, float damping, int numBodies, int p, int q)
{
    int sharedMemSize = p * q * sizeof(float4); // 4 floats for pos
dim3 threads(p,q,1); // 256 x 1 x 1 threads per block
dim3 grid(numBodies/p, 1, 1); // 4 x 1 x 1 blocks

    if (grid.y == 1)
    {
        integrateBodies<false><<< grid, threads, sharedMemSize >>>
        ((float4*)newPos, (float4*)newVel,
         (float4*)oldPos, (float4*)oldVel,
         deltaTime, damping, numBodies);
    }
    else
    {
        integrateBodies<true><<< grid, threads, sharedMemSize >>>
        ((float4*)newPos, (float4*)newVel,
         (float4*)oldPos, (float4*)oldVel,
         deltaTime, damping, numBodies);
    }
    CUT_CHECK_ERROR("Kernel execution failed");
}
```
Objectives

Project Organization

CPU

GPU

GPGPU

CUDA

N-body problem

MD on CUDA

Evaluation

Future Work
Testbed

CPU
- 8 Intel Xeon processors
- 4 cores @ 3.00 GHz per processor
- 4 GB RAM
- SUSE 10.2 (X86-64)

GPU
- Nvidia Quadro FX 5600
Quadro FX 5600 Specifications

- Memory Size: 1.5GB GDDR3
- Memory Interface: 384-bit
- Memory Bandwidth: 76.8 GB/sec.
- Max Power Consumption: 171W
- Number of Slots: 2
- Display Connectors: DVI-I DVI-I Stereo
- Dual-Link DVI: 2
Performance Testing

3 tests are performed to understand the performance of MD-CUDA:

- Test 1: find the number of iterations that achieve a near maximum utilization
- Test 2: find a value of N (# of bodies) that achieves the highest performance at a reasonable execution time
- Test 3: find the optimal p and q values
  - Full utilization occurs for p*q = 256
Test 1: GFLOPS/s for different # of Iterations

- N = 16384, p = 256, q = 1
- Standard deviation: Min = 0.0008367, Max = 0.0139
- **# of iterations = 1000** produced an average performance of **214.6474 GFLOPS/s**, compared to 214.7892 GFLOPS/s when # of iterations = 3000
- Runtime of 1000 iterations was 2.998x faster than 3000 iterations
Test 2: GFLOPS/s for different Problem Sizes

- p=256, q = 1, # iter = 1000
- Varied N to analyze scalability
- Standard deviation: Min = 0.000859, Max = 0.0244
- **N = 16384** performed an average of 214.6454 GFLOPS/s, where N = 32768 produced an average of 216.602 GFLOPS/s
- Runtime of N = 16384 was 3.963x faster than N = 32768
Test 3: GFLOPS/s for different values of $p$ and $q$
Test 3: GFLOPS/s for different $p$ and $q$

- Higher FLOPS/s when $q = 1$ and $p > 1$
Objectives

Project Organization

CPU

GPU

GPGPU

CUDA

N-body problem

MD on CUDA

Evaluation

Future Work
Future Work

- The accumulation of the potential energy is not an easy task to perform in parallel.
- The naïve solution would be to have every thread update a single shared memory location with the potential for a single body-body interaction in a synchronized fashion.
  - This will serialize a portion of the code and is not acceptable.
- Another naïve solution would be to copy the position and velocity arrays off the GPU and serially compute the potential energy on the CPU.
  - This also has poor performance and requires some redundant computations to obtain the total potential energy.
- We need to develop a parallel summation algorithm that possibly uses partial summation within thread blocks.
References
Test 3 (2/5)

\[ \text{p,q VS. GFLOPS/s} \]

\[ \text{p,q VS GLOPS/s} \]
Test 3 (3/5)

p,q VS GFLOPS/s

GFLOPS/s

8  1  8  2  8  4  8  16  8  32

p,q

p,q VS GFLOPS/s

GFLOPS/s

4  1  4  2  4  16  4  32  4  64

p,q
Test 3 (4/5)