GPU COMPUTING
LECTURE 08 - N-BODY METHODS

Kazem Shekofteh
Kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg
Inspired from lectures by Holger Fröning

Partially based on "CUDA Handbook" by Nicholas Wilt

(MORE) CODE OPTIMIZATIONS

Case study: n-Body Particle Computations in Physics

ARRAYS OF STRUCTURES (AOS)

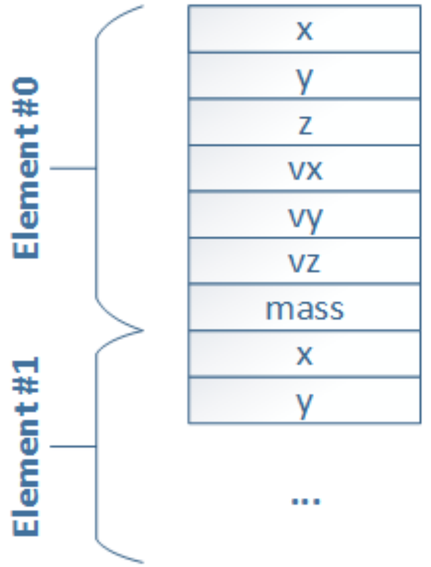
```
struct {
  float x, y, z;
  float vx, vy, vz;
  float mass;
} p_t;

p_t particles [MAX_SIZE];
```

Memory access is one of the most expensive operations

Data is grouped per element index, different types next to each other

Typical in most applications



STRUCTURE OF ARRAYS (SOA)

Data is grouped per element type, elements distributed among different arrays with the same index

- Many pointers required
- => Register use

Typical for GPU applications

- Multiple threads are accessing memory concurrently
- Thread organization affects memory performance
- SOA better for regular memory access pattern

```
struct {
  float   x   [MAX_SIZE],
          y   [MAX_SIZE],
          z   [MAX_SIZE];
  float   vx  [MAX_SIZE],
          vy  [MAX_SIZE],
          vz  [MAX_SIZE];
  float   mass[MAX_SIZE];
} p_t;

p_t particles;
```

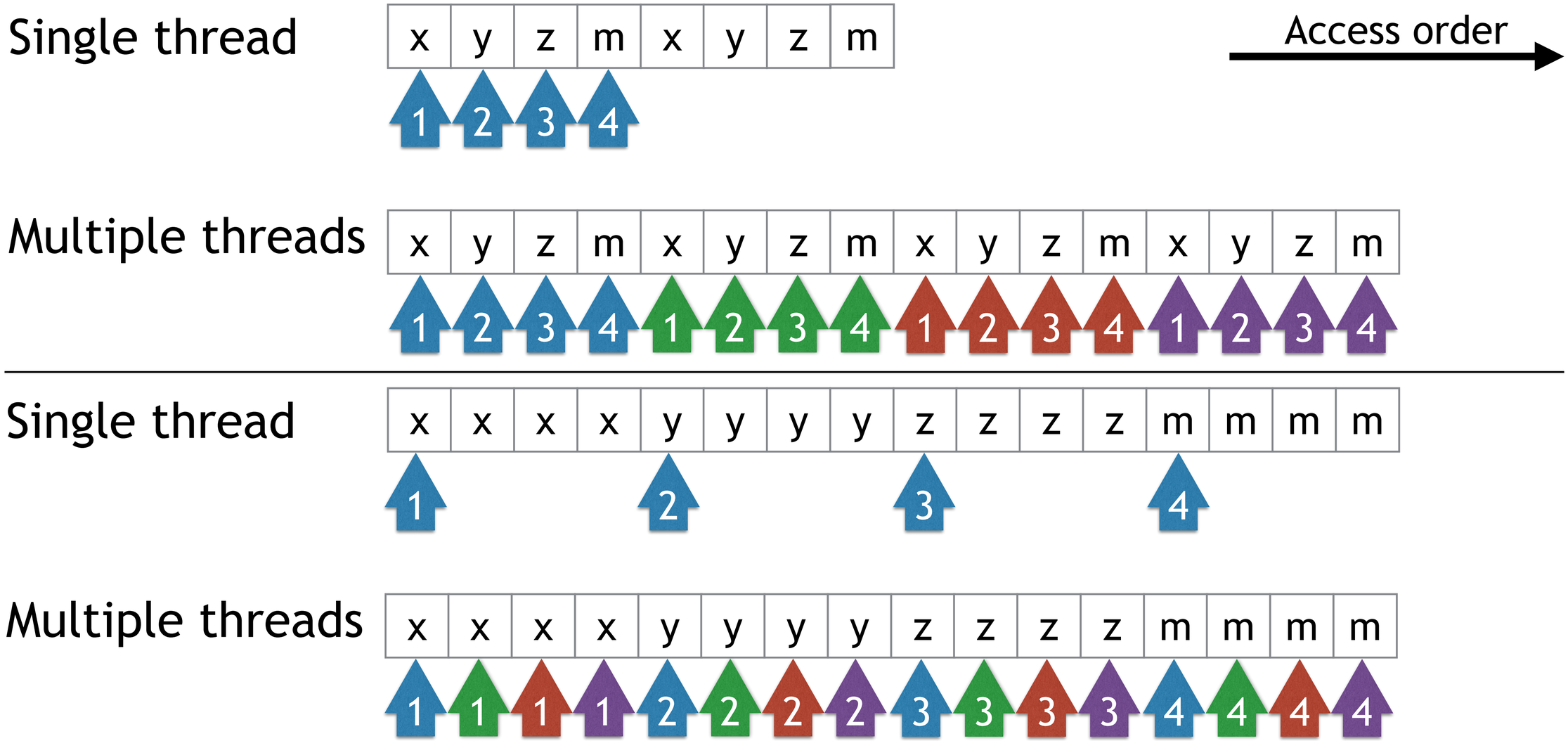
REMINDER: COALESCING

Coalesced: when the memory access issued by multiple threads is ordered in a way that consecutive addresses are used, so that multiple accesses can be translated into one single memory transaction

Non-coalesced: when multiple memory transactions are required to fulfill the memory access requests, or when a transaction is not completely utilized

Performance penalty highly depends on access pattern

NON-COALESCED AOS VS COALESCED SOA

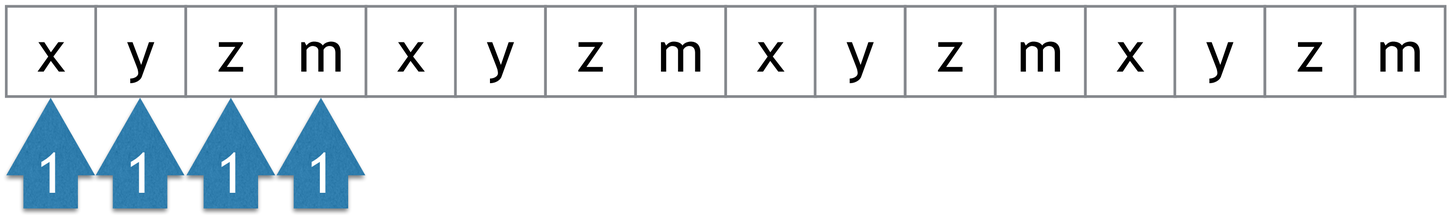


COALESCED AOS – PACKED VALUES

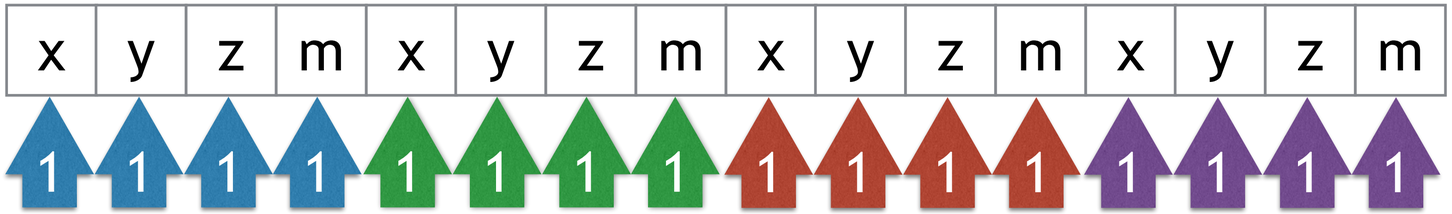
Access order

$$float4 = \begin{pmatrix} x \\ y \\ z \\ m \end{pmatrix}$$

Single thread



Multiple threads



TRADING MEMORY VS COMPUTE

- Using more memory to reduce the compute pressure
  - Look-up tables
  - Precompute certain values for re-use
- Use more computing to reduce memory pressure
  - Reorder elements/computations to improve locality (increased control flow complexity)
  - Improve memory coalescing
  - No re-use of values, recomputing instead (GPU cycles are cheap)
- Optimizations targeting compute instructions useless for memory-bound applications and vice versa

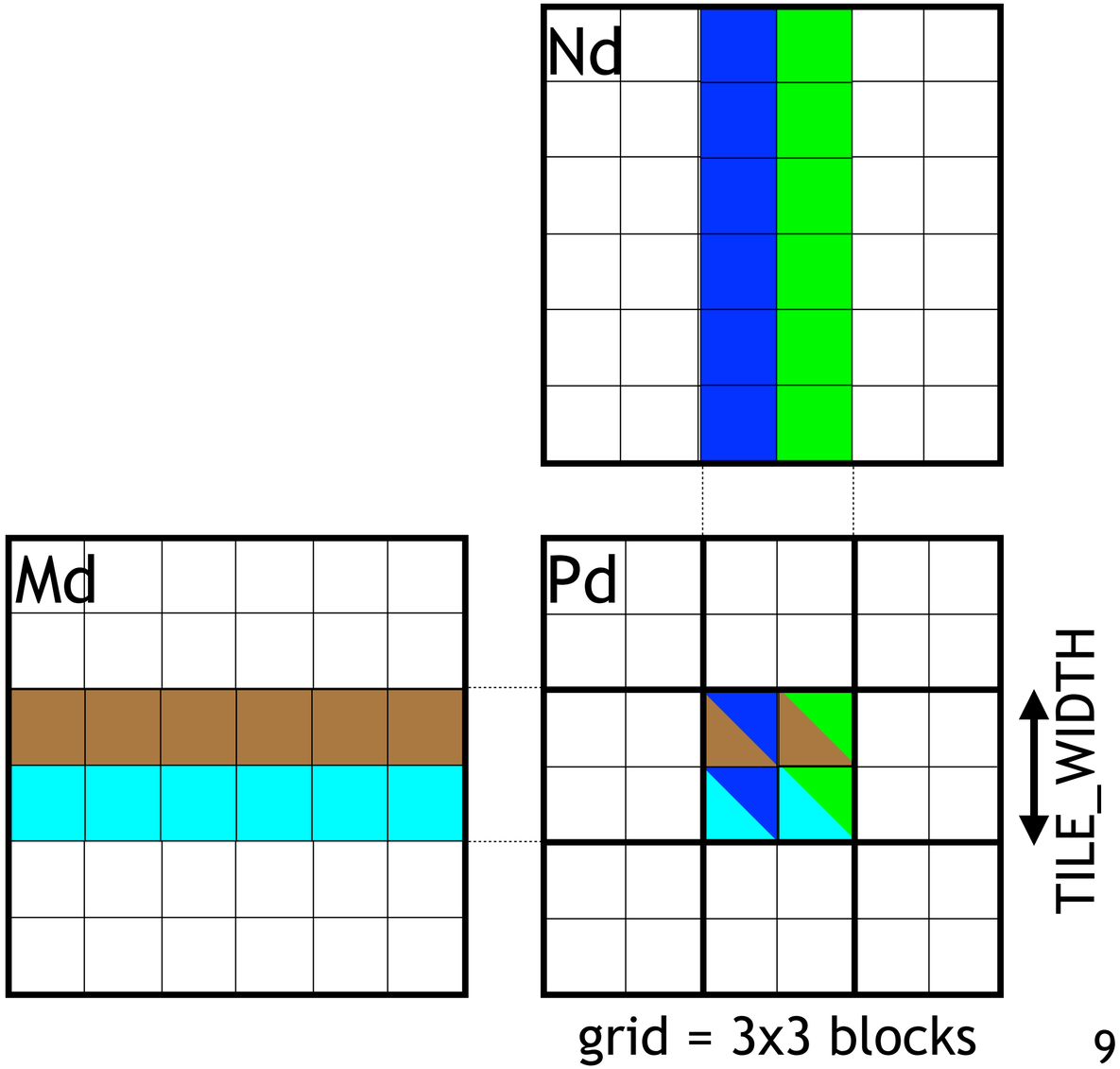
REMINDER: TILING

Divide a long repetitive process in regular iterative blocks

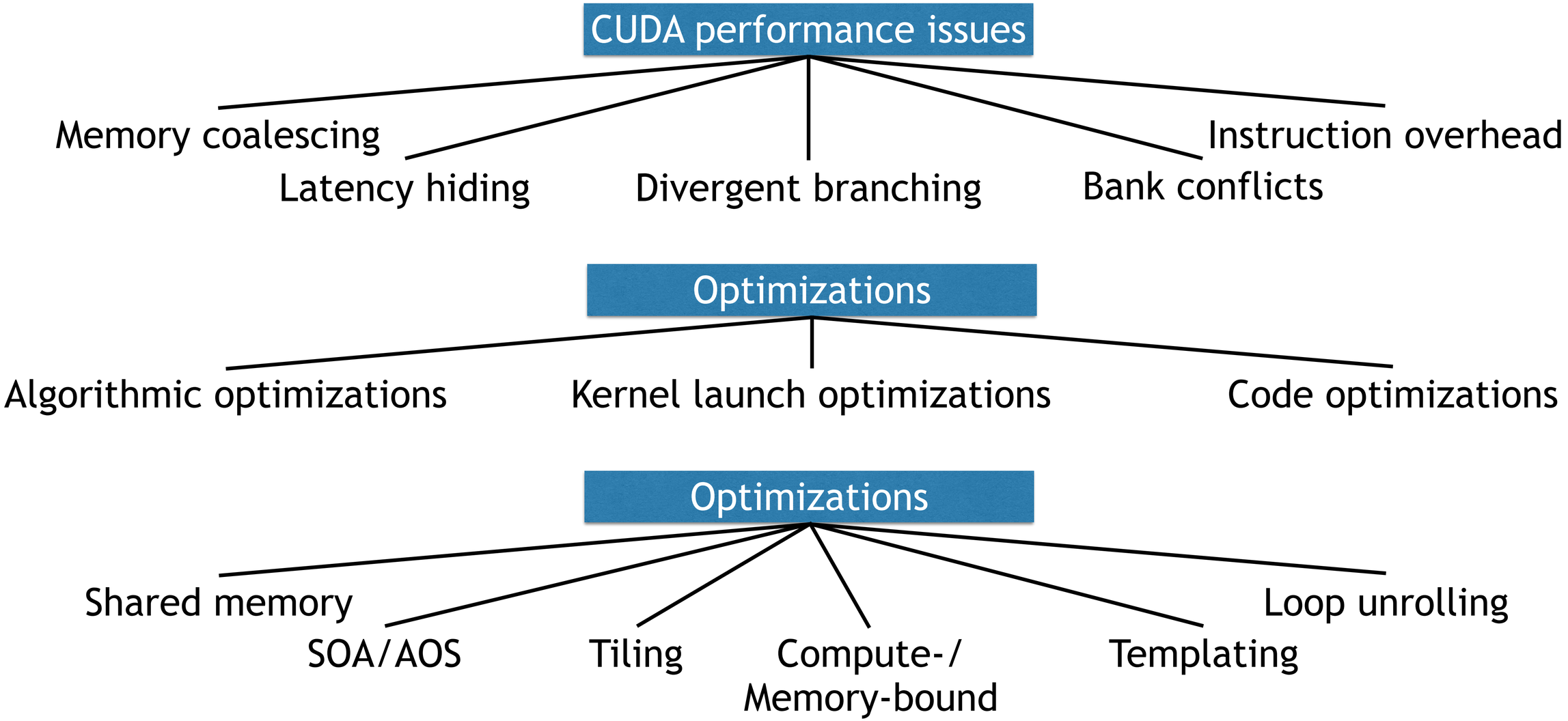
- Limiting the amount of resources required
- Making computation and memory access (more) regular
- Making the computation independent of the actual problem size (for many/most workloads)

Many operations are associative

- a + (b + c) = (a + b) + c
- So feel free to reorder multiply operations
- Goal: increase data reuse



OPTIMIZATIONS



N-BODY PARTICLE COMPUTATIONS

N-BODY SIMULATIONS

Modeling biomolecular systems

- Electrostatic and Van der Waals forces

Time scale of 1^{fs} (10^{-15}s)

Example workload: Satellite Tobacco Mosaic Virus (STMV)

- 100M atoms, 160 genes
- Petascale-class: 100ns/day of simulation time

Complex structures like parasites: ~ 6k genes

- 1400x runtime increase

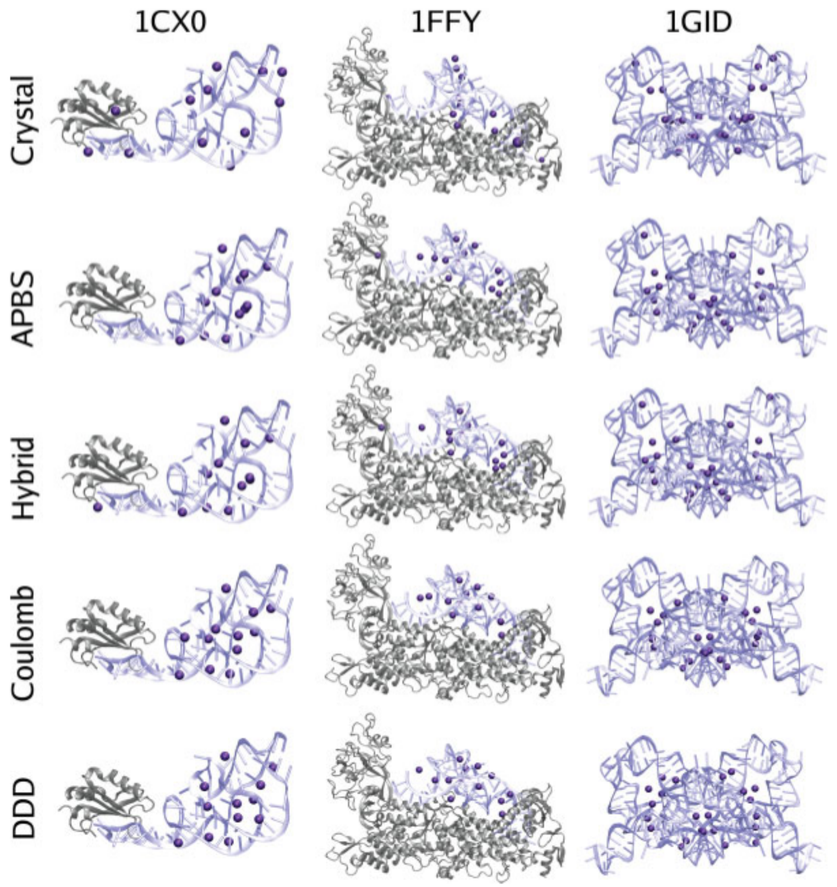


Figure 8. Comparison of ion placement test cases used in this study with crystal ions. Structures are arranged in columns, from left to right, hepatitis delta virus ribozyme (1CX0), tRNA-Ile/synthetase (1FFY), and P4-P6 ribozyme domain (1GID). The crystal structure is shown at the top of each column, followed by placement with the APBS, Hybrid, Coulomb, and DDD methods (see Table 8 for abbreviations). [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

Stone JE, Phillips JC, Freddolino PL, Hardy DJ, Trabuco LG, Schulten K. 2007. Accelerating molecular modeling applications with graphics processors. J Comput Chem. 2007 Dec;28(16):2618-40.

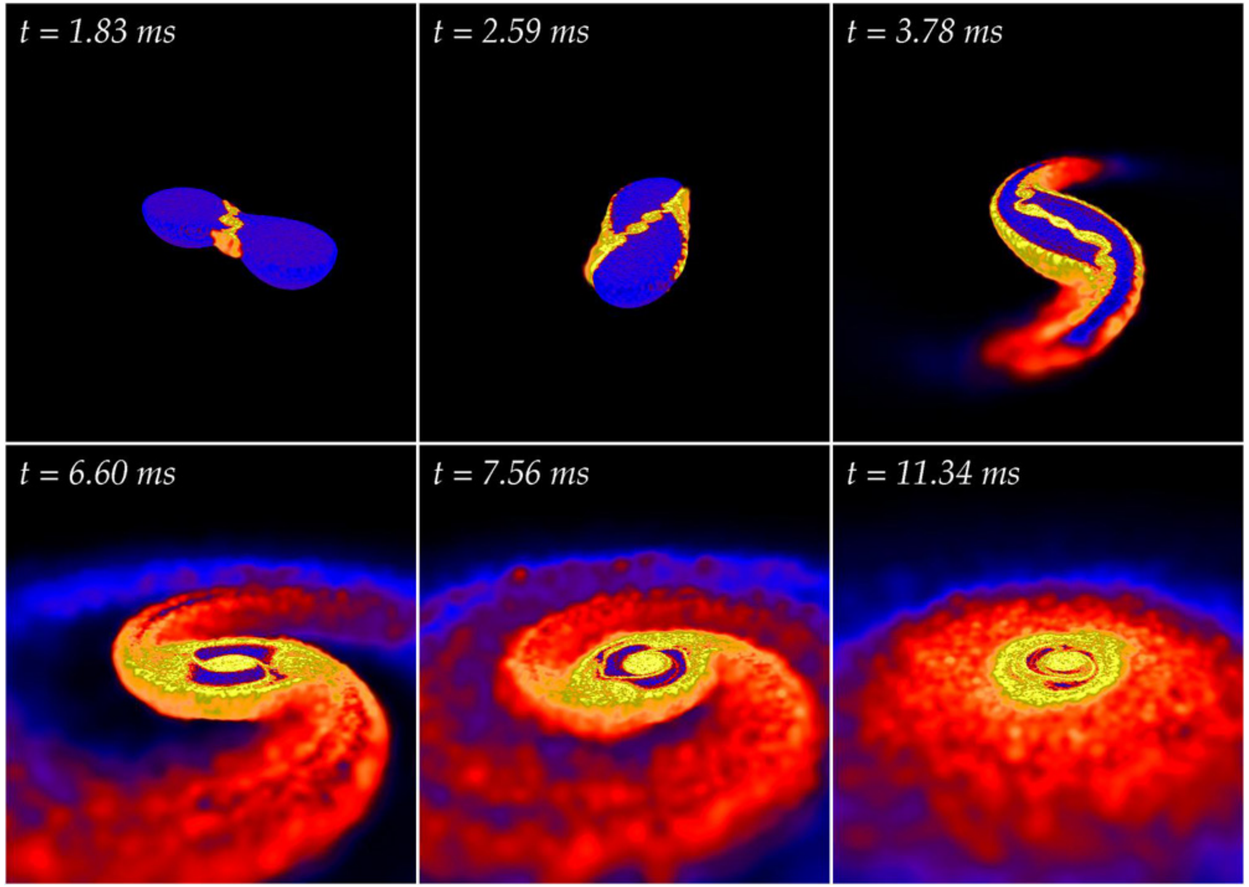
N-BODY SIMULATIONS

Astrophysics: galaxy simulations

- Gravitational forces
- Smoothed particle hydrodynamics (fluids)

Contributed to the discovery of dark energy

- 1990‘s at LBNL



Gravitational waves of the collision of two neutron stars

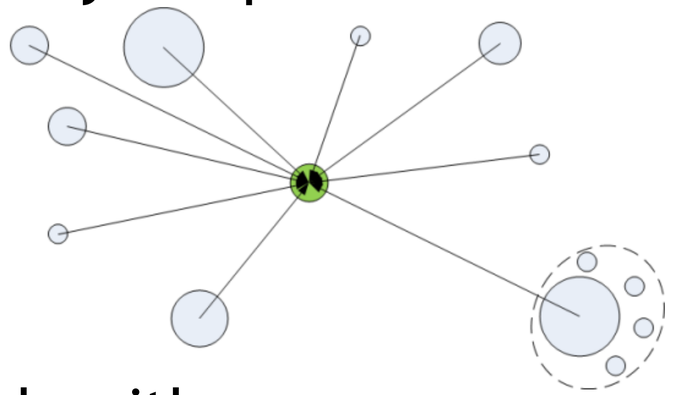
Source: ucsc.edu

N-BODY SIMULATIONS

Newton's second law of motion

$$F(x(t)) = m\frac{d^2x(t)}{dt^2}$$

Approximating the solution of the differential equation by temporal discretization



-> Simulation based on time steps

- Each time step costs $O(N^2)$ operations, N = body count
- Computational bound as memory costs are O(N)

Forces decrease quickly with distance -> hierarchical algorithms

- Within clusters all-pair methods; for k bodies: $O(k^2)$ -> very suitable for GPUs

Far-field approximations

- Barnes-Hut algorithm based on spatial hierarchy between clusters of objects: O(N*log(N))
- Fast multipole method

N-BODY EXAMPLE FORMULA

N bodies with positions x and velocities v, forces caused by gravity are

$$f_{ij} = G \cdot \frac{m_i m_j}{\|{d_{ij}}^2\|} \cdot \frac{d_{ij}}{\|d_{ij}\|}$$

To avoid divide overflow, introduce softening factor

$$f_{ij} = G \cdot \frac{m_i m_j d_{ij}}{(\|{d_{ij}}^2\| + \varepsilon^2)^{\frac{3}{2}}}$$

Total force

O(N^2)

$$F_i = \sum_{j=1}^{N} f_{ij} = G m_i \sum_{j=1}^{N} \frac{m_j d_{ij}}{(\|{d_{ij}}^2\| + \varepsilon^2)^{\frac{3}{2}}}$$

Leapfrog verlet algorithm updates velocity, then position

O(N)

See also Noether's theorem (but only for continuous settings) (https://en.wikipedia.org/wiki/Noether%27s_theorem)

$$v_i(t + \frac{1}{2}\delta t) = v_i(t - \frac{1}{2}\delta t) + \delta t \frac{F_i}{m_i}$$

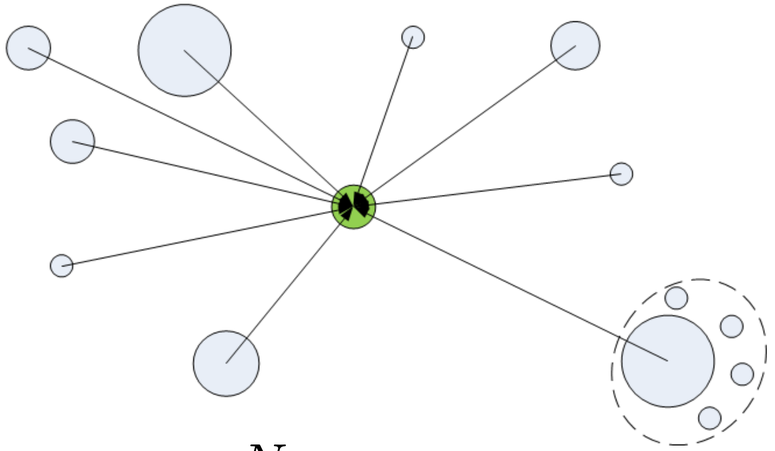
$$x_i(t + \delta t) = x_i(t) + \delta t \cdot v_i(t + \frac{1}{2}\delta t)$$

GPU IMPLEMENTATION OPTIONS

Partitioning: one thread per body

Communication

- Optimize for data re-use



2 approaches here:

- Naive implementation
- Tiled implementation (shared memory)
- (Constant memory implementation)
- (Warp shuffle implementation)

$$F_i = \sum_{j=1}^{N} f_{ij} = Gm_i \sum_{j=1}^{N} \frac{m_j d_{ij}}{(\|{d_{ij}}^2\| + \varepsilon^2)^{\frac{3}{2}}}$$

FORCE MATRIX

N x N grid of pair-wise forces

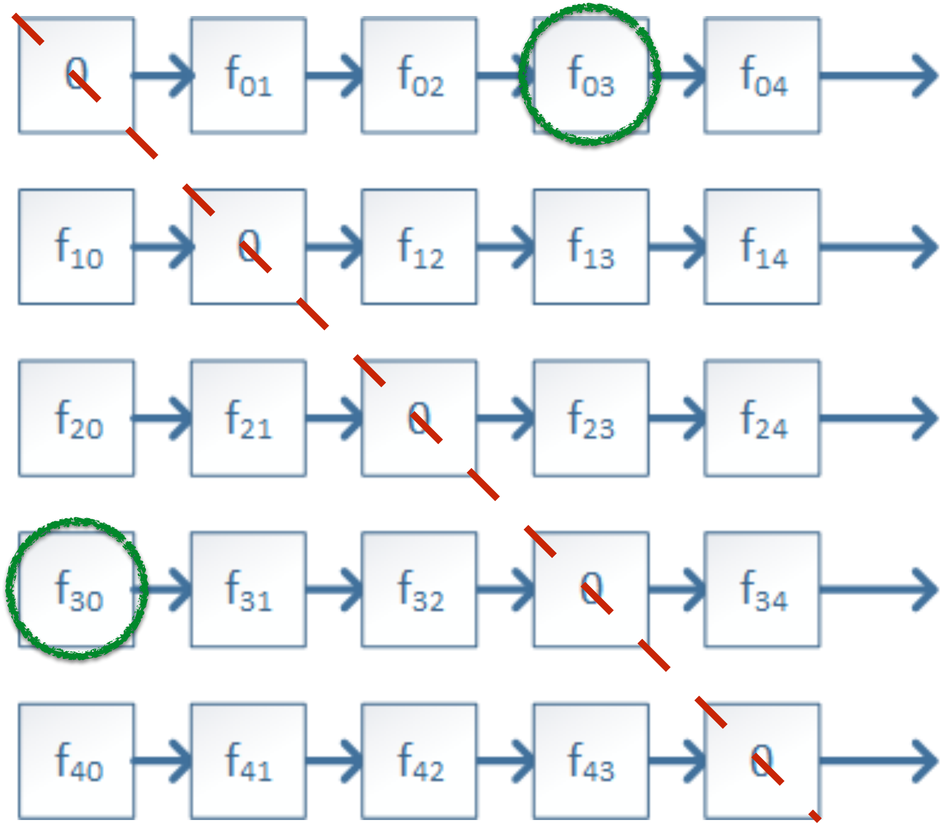
- Resp. accelerations

Symmetry reducing computations by half

- $f_{02} = -f_{20}$
- Partial sums required
- Temporary locations or mutual exclusions
- -> Overhead often overwhelms the benefits

Partitioning alternative: one thread per force?

- Global sum at the end -> synchronization



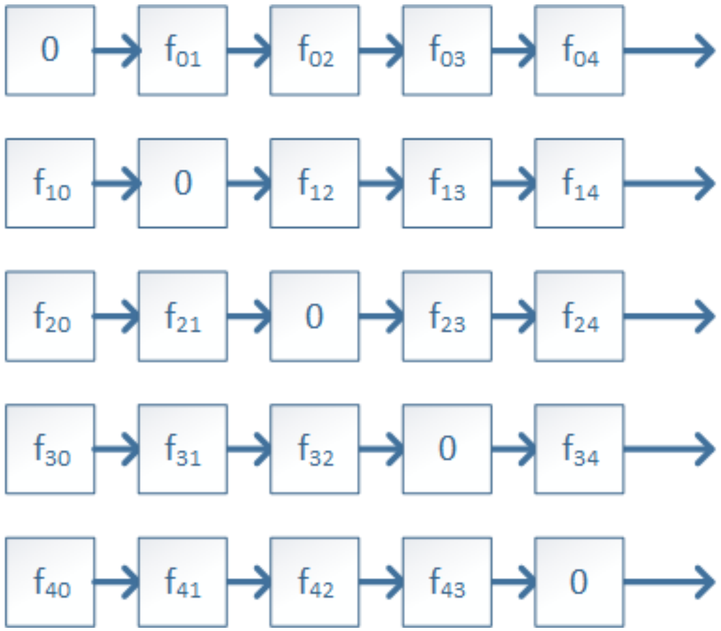
$$\sum_{j=0}^{N-1} f_{ij}$$

NAIVE GPU IMPLEMENTATION

Kernel for body-body interaction

Calculating f_{ij}

16 SP FLOPs




```
__host__ __device__ void bodyBodyInteraction(...)
{
    float dx = x1 - x0;
    float dy = y1 - y0;
    float dz = z1 - z0;

    float distSqr = dx*dx + dy*dy + dz*dz;
    distSqr += softeningSquared;

    float invDist = rsqrtf(distSqr);

    float invDistCube =  invDist * invDist * invDist;
    float s = mass1 * invDistCube;

    *fx = dx * s;
    *fy = dy * s;
    *fz = dz * s;
}
```

NAIVE GPU IMPLEMENTATION

- Code for gravitational forces for each body
  - AOS packed
  - float4 means 16B load instructions
  - Casting to ensure this
- Inner loop with good data re-use
  - How many threads access the same data?
  - Cache benefits for SM 2.x and later

```
__global__ void ComputeNBodyGravitation_GPU_AOS (float
*force, float *posMass, size_t N, float softeningSquared)
{
for ( int i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x ) {
    float acc[3] = {0};
    float4 me = ((float4 *) posMass)[i];
    float myX = me.x; float myY = me.y; float myZ = me.z;
    for ( int j = 0; j < N; j++ ) {
        float4 body = ((float4 *) posMass)[j];
        float fx, fy, fz;
        bodyBodyInteraction(
            &fx, &fy, &fz, myX, myY, myZ,
            body.x, body.y, body.z, body.w,
            softeningSquared);
        acc[0] += fx; acc[1] += fy; acc[2] += fz;
        }
    force[3*i+0] = acc[0];
    force[3*i+1] = acc[1];
    force[3*i+2] = acc[2];
    }
}
```

NAIVE GPU IMPLEMENTATION

Loop unrolling reduces branch overhead => insert unroll pragma

`#pragma unroll 1` will prevent the compiler from ever unrolling a loop.

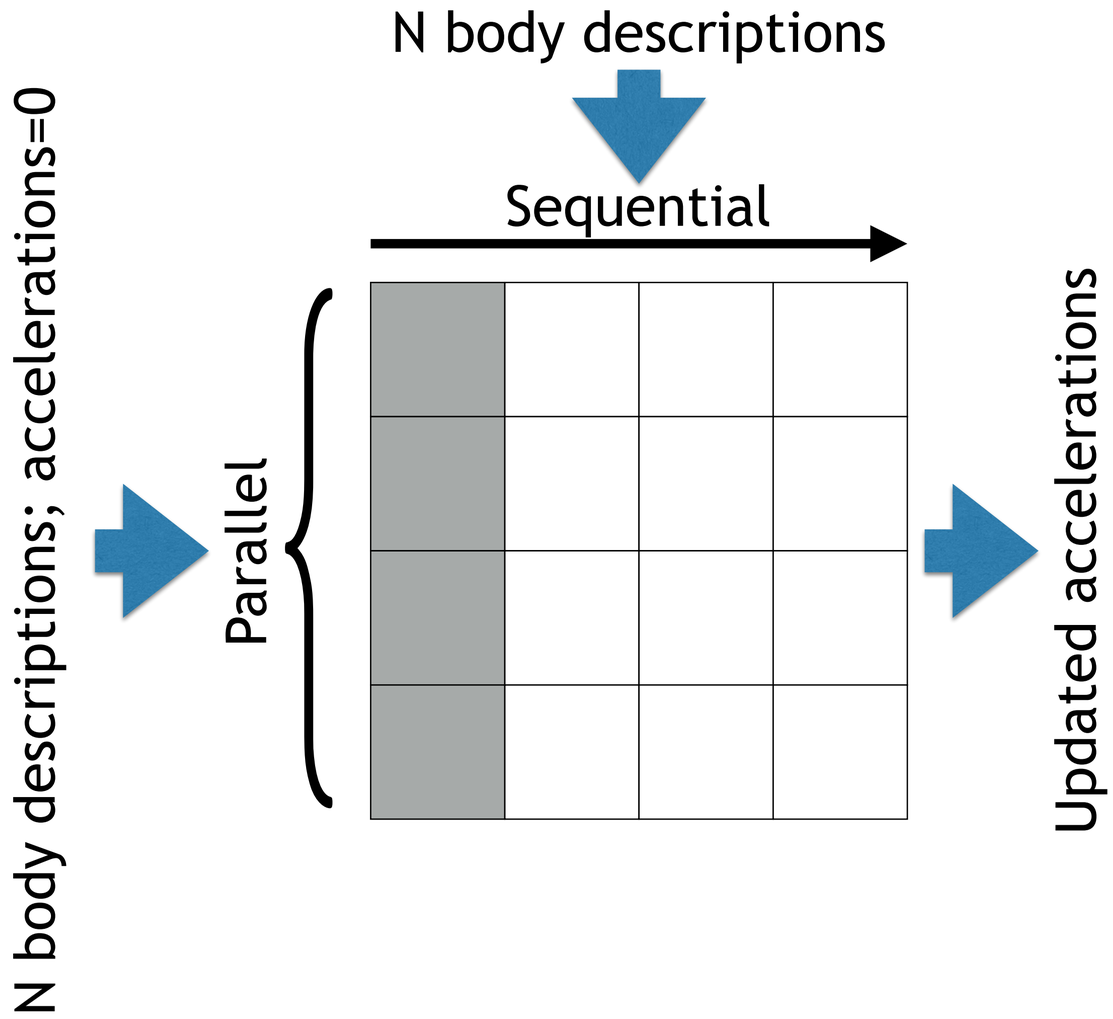
If no number is specified after `#pragma unroll`, the loop is completely unrolled if its trip count is constant, otherwise it is not unrolled at all.

Optimal unrolling factor has to be determined empirically

Version	Unroll factor	Body-body interactions per second [G]
GPU naive	1 (no unrolling)	25
	2	30
	16	34,3

OPTIMIZATIONS?

- Force matrix of NxN
  - Caches already exploit locality
  - Shared memory will further improve this
- Tiling as an optimization
  - Make computations and memory accesses more regular



TILING

Tile NxN matrix into sub matrices with height of p

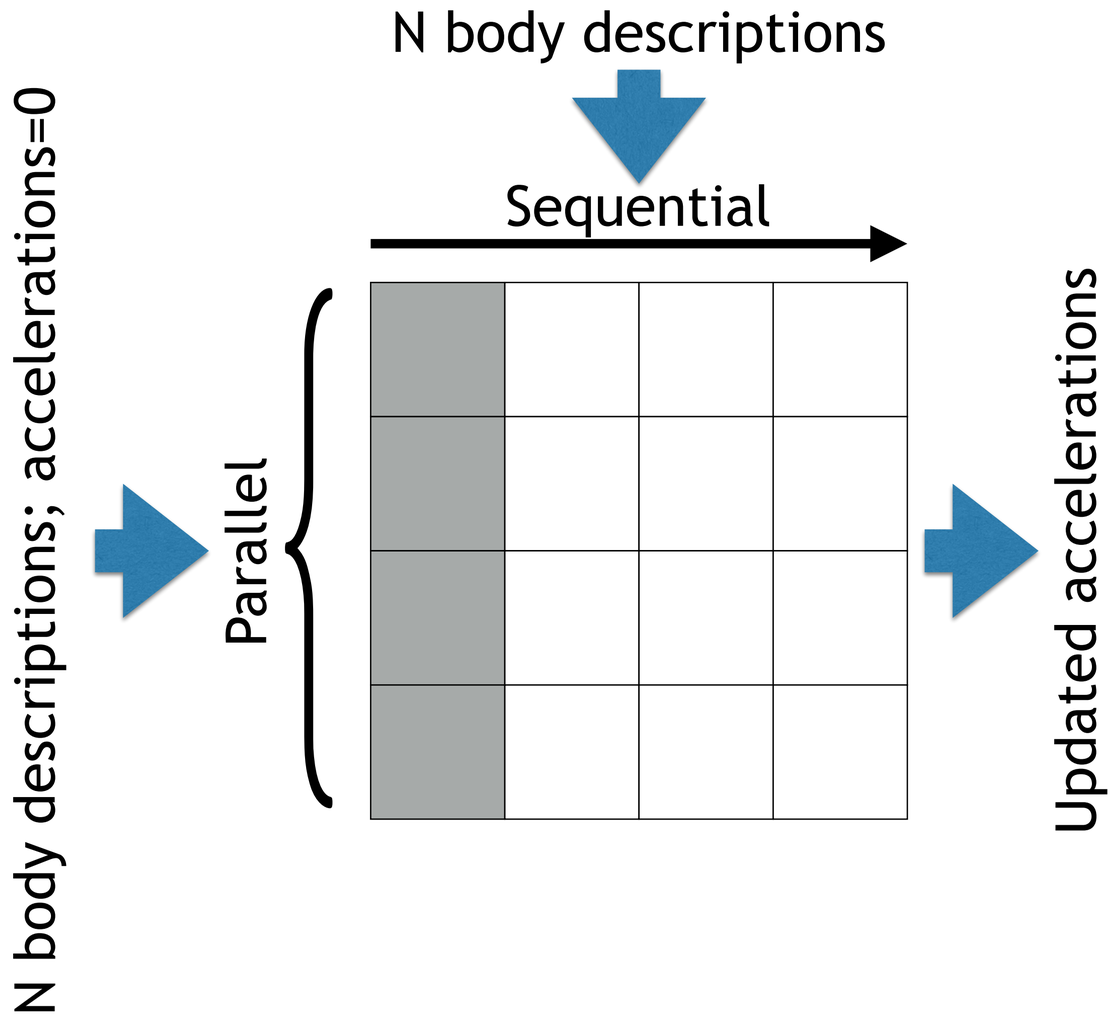
- Block size = p
- Each thread computes N interactions for one body

Submatrix width of p

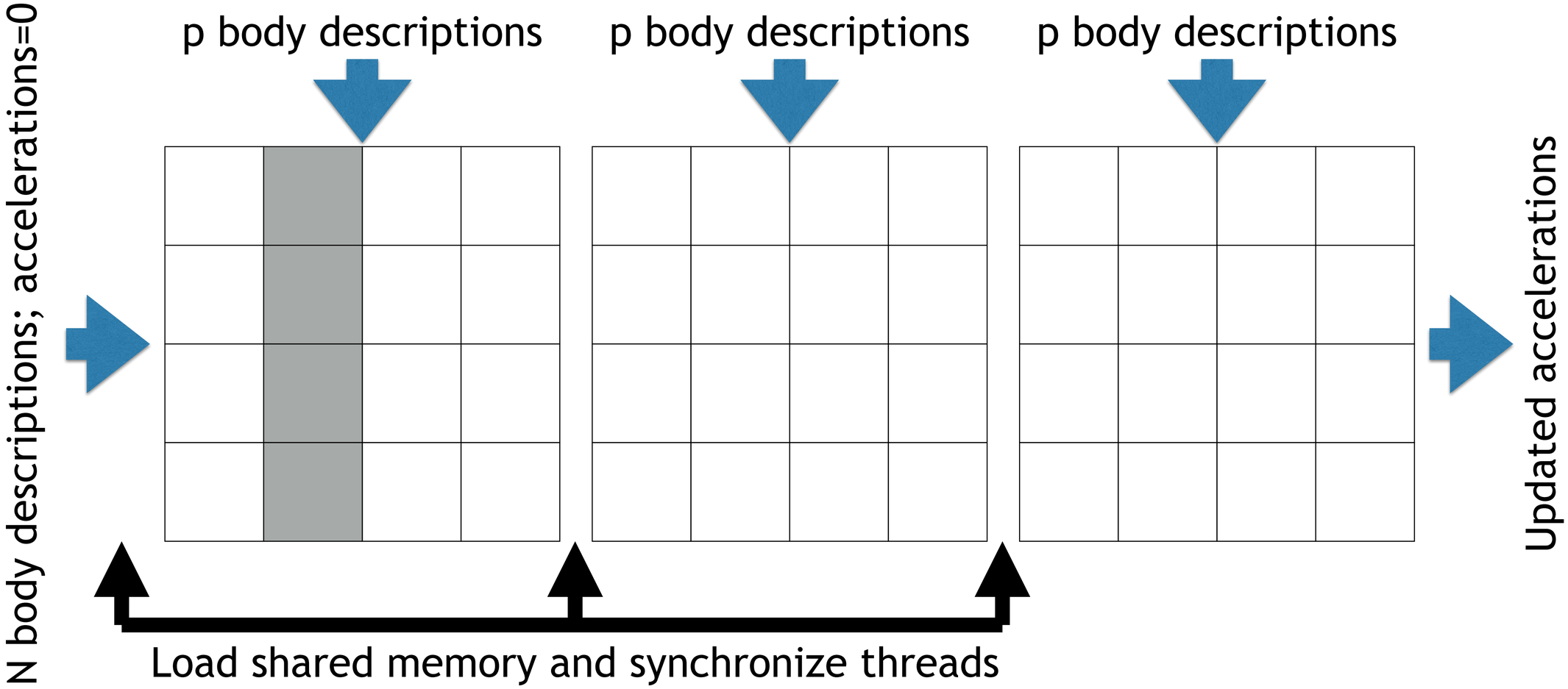
- After p steps, reload shared memory

Cooperative loads

Data re-use



TILING


p body descriptions
p body descriptions
p body descriptions
N body descriptions; accelerations=0
Updated accelerations
Load shared memory and synchronize threads

OPTIMIZED GPU IMPLEMENTATION

```
__global__ void ComputeNBodyGravitation_Shared ( float *force, float *posMass, float
softeningSquared, size_t N ) {
    extern __shared__ float4 shPosMass[];
    for (int i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i += blockDim.x*gridDim.x){
        float acc[3] = {0};
        float4 myPosMass = ((float4 *) posMass)[i];
#pragma unroll 32
        for ( int j = 0; j < N; j += blockDim.x ) {
            shPosMass[threadIdx.x] = ((float4 *) posMass)[j+threadIdx.x];
            __syncthreads();
            for ( size_t k = 0; k < blockDim.x; k++ ) {
                float fx, fy, fz;
                float4 bodyPosMass = shPosMass[k];
                bodyBodyInteraction(
                    &fx, &fy, &fz,
                    myPosMass.x, myPosMass.y, myPosMass.z, bodyPosMass.x, bodyPosMass.y,
                    bodyPosMass.z, bodyPosMass.w, softeningSquared );
                acc[0] += fx; acc[1] += fy; acc[2] += fz;
            }
            __syncthreads();
        }
        force[3*i+0] = acc[0]; force[3*i+1] = acc[1]; force[3*i+2] = acc[2];
    }
}
```

outer loop that strides through bodies

inner loop that iterates over body descriptions

OPTIMIZED GPU IMPLEMENTATION

- Loop unrolling helps again
  - Empirical determination
- Scales easily further with multiple GPUs
  - Strong scaling (scales with fixed problem size)

Version	Unroll factor	Body-body interactions per second [G]
GPU naive	1 (no unrolling)	25
	2	30
	16	34,3
GPU shmem	1 (no unrolling)	38,2
	2	44,5
	3	42,6
	4	45,2

OPTIMIZED CPU IMPLEMENTATION

```
inline void bodyBodyInteraction(__m128&
fx, __m128& fy, __m128& fz, const __m128&
x0, const __m128& y0, const __m128& z0,
const __m128& x1, const __m128& y1, const
__m128& z1, const __m128& mass1, const
__m128& softSquared )
{
// r_01  [3 FLOPS]
__m128 dx = _mm_sub_ps( x1, x0 );
__m128 dy = _mm_sub_ps( y1, y0 );
__m128 dz = _mm_sub_ps( z1, z0 );

// d^2 + e^2 [6 FLOPS]
__m128 distSq =
   _mm_add_ps(
       _mm_add_ps(
           _mm_mul_ps( dx, dx ),
           _mm_mul_ps( dy, dy )
        ),
        _mm_mul_ps( dz, dz )
   );
distSq =_mm_add_ps(distSq,softSquared );

// invDistCube =1/distSqr^(3/2)  [4 FLOPS]
__m128 invDist = rcp_sqrt_nr_ps( distSq );
__m128 invDistCube =
   _mm_mul_ps(
        invDist,
        _mm_mul_ps(
            invDist, invDist )
    );

// s = m_j * invDistCube [1 FLOP]
__m128 s = _mm_mul_ps(mass1,invDistCube);

// (m_1 * r_01) / (d^2 + e^2)^(3/2)  [6
FLOPS]
fx = _mm_add_ps( fx, _mm_mul_ps(dx, s) );
fy = _mm_add_ps( fy, _mm_mul_ps(dy, s) );
fz = _mm_add_ps( fz, _mm_mul_ps(dz, s) );
}
```

SSE: 128bit XMM registers -> SOA

Multi-threading not shown

SCALAR OR VECTOR VIEW?

1. Right vector size
2. Dynamic instruction selection

   Uniform, linear, varying
3. Semantic model for compiler

   Intra-thread model = dependencies (SC), optimizer aware, good code quality

   Inter-thread model = DRF memory, optimizer aware, good code quality

   => Scalar compiler sees only dependencies and DRF memory

   => Vector compiler sees mixture of SC and DRF
4. C++ is for scalars

   GPU: C++, scalar compiler, scalar ISA, thread virtualization, vector units

   CPU: C++/V-C++, scalar/vector compiler, scalar/vector ISA, scalar/vector units

SC only within one thread of control

DRF among threads (since C++11)

SIMT (change vectors for C++) vs SIMD (change C++ for vectors)

OPTIMIZED CPU IMPLEMENTATION

Expect a strong scaling for multiple cores

Performance comparison

- Intel E5-2670 CPUs
- GK104 GPUs

Version	Unroll factor	Body-body interactions per second [G]
GPU naive	1 (no unrolling)	25
	2	30
	16	34,3
GPU shmem	1 (no unrolling)	38,2
	2	44,5
	3	42,6
	4	45,2
CPU naive, single threaded		0,017
CPU SSE, single threaded		0,307
CPU SSE, 32 threads		5,650

WRAPPING UP

SUMMARY

N-Body computations a prime example for GPUs

- Different optimizations helpful
- Loop unrolling and shared memory

Skipped here: warp shuffle instruction

- Communication within thread warps: `__shfl()` intrinsic
- Latency of about a shared memory read
- Need to tile computation at warp size, rather than CTA size
- No performance benefits here (about 25% slower than shared memory)

Skipped here: constant memory

- Host-GPU context switches required per iteration to reload constant memory

N-Body is also a great example for CPU/GPU code complexity