Simple Molecular Dynamics

Gustavo Pereira PG 53723
Nelson Almeida PG 52697
Computação Paralela
2023/24

Escola de Engenharia
Universidade do Minho

Summary

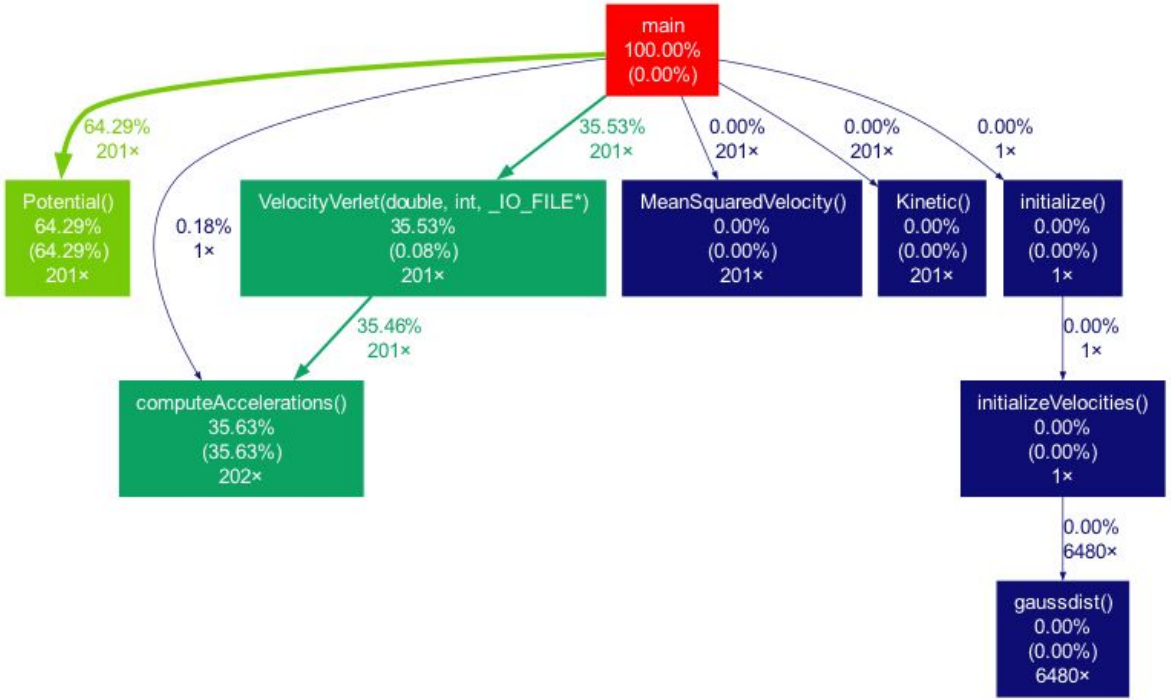
1. Single-Threaded Program Optimization
    - Profiling
    - Complexity analysis
    - Optimizations (I, II and III)
2. Shared Memory Parallelism
    - Profiling
    - Parallelization using OpenMP
    - Expectations and Results
3. GPU Acceleration with CUDA
    - Approach
    - Implementation
    - Result Analysis
4. OpenMP vs CUDA

Profiling and code analysis

Tools: gprof and gprof2dot

Hot spots encountered:

- Potencial()
- computeAccelerations()



Profiling and code analysis

```
double Potential() {
    double quot, r2, rnorm, term1, term2, Pot;
    int i, j, k;
    Pot=0.;

    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if (j!=i) {
                r2=0.;
                for (k=0; k<3; k++) {
                    r2 += (r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
                }
                rnorm=sqrt(r2);
                quot=sigma/rnorm;
                term1 = pow(quot,12.);
                term2 = pow(quot,6.);
                Pot += 4*epsilon*(term1 - term2);
            }
        }
    }
    return Pot;
}
```

Potencial() called 201 times

Nested *for* loops with complexity $O(N^2)$

N = 2,160, resulting in 4 665 600 iterations per call, in the worst case scenario

Profiling and code analysis

```
void computeAccelerations() {
        (...)

        for (i = 0; i < N; i++) {  // set all accelerations to zero
                for (k = 0; k < 3; k++) {
                        a[i][k] = 0;
                }
        }
        for (i = 0; i < N-1; i++) {    // loop over all distinct pairs i,j
                for (j = i+1; j < N; j++) {
                        (...)
                        for (k = 0; k < 3; k++) {
                                (...)
                        }
                }
        }
}
```

computeAccelerations() called 201 times

Nested *for* loops with complexity $O(N^2)$

N = 2,160, resulting in 4 665 600 iterations per call, in the worst case scenario

Optimizations I

Reduction the use of costly mathematical (division and exponentiation) operations.

```
f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4))
```

to

```
rSqd7 = rSqd3*rSqd3*rSqd;
f =  (1/rSqd7)*(48-24*rSqd3);
```


https://stackoverflow.com/questions/2940367/what-is-more-efficient-using-pow-to-square-or-just-multiply-it-with-itself

Optimizations II

Merge of *Potencial()* with *computeAccelerations()* functions combining their loops

Motivation:

- Similar logic
- Saves iterations
- Called simultaneously
- Potencial() had redundant calculations (calculating same pair twice)

Optimizations II

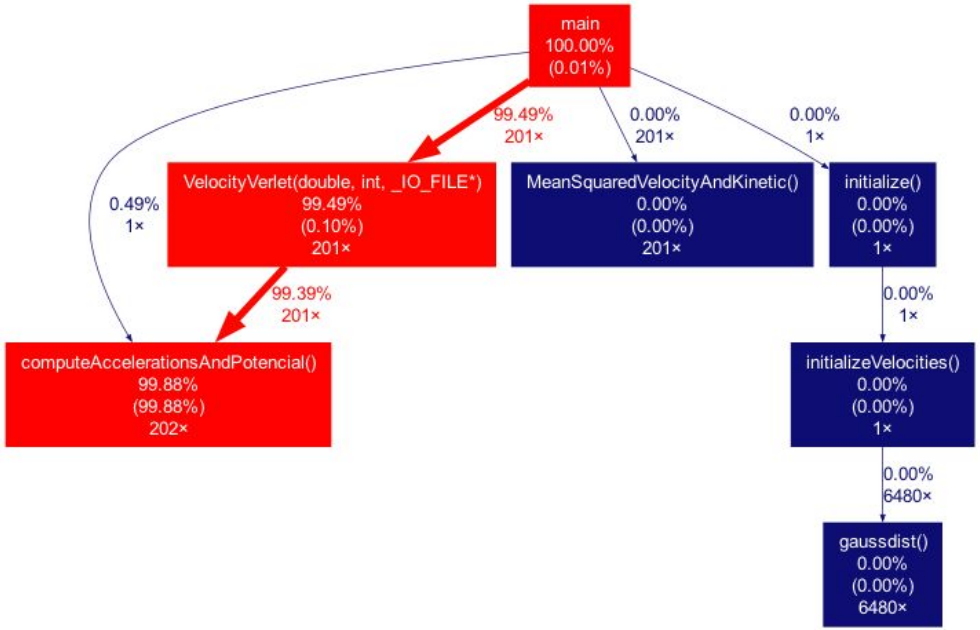
Merge of *Potencial()* with *computeAccelerations()* functions combining their loops

CYCLE AND INSTRUCTION METRICS COMPARISON

Total	Before	After
Instructions	1,236,991,944,534	260,326,290,217
Cycles	766,640,483,170	163,770,254,165

Optimizations II

Merge of *Potencial()* with *computeAccelerations()* functions combining their loops



Optimizations III

Exploration of compiler-driven math optimization using flags.

-O3
- Loop Unrolling
- Vectorization

-funroll-loops
- Loop Unrolling
- Enhanced Pipelining

-ffast-math
- Faster Math Functions
- Associative Math Transformations

Best performance achieved using the **-O3 -funroll-loops -ffast-math** flags combined

Optimizations III

Exploration of compiler-driven math optimization using flags.

Conversion of matrices into vectors to allow vectorization.

PERFORMANCE COUNTER COMPARISON

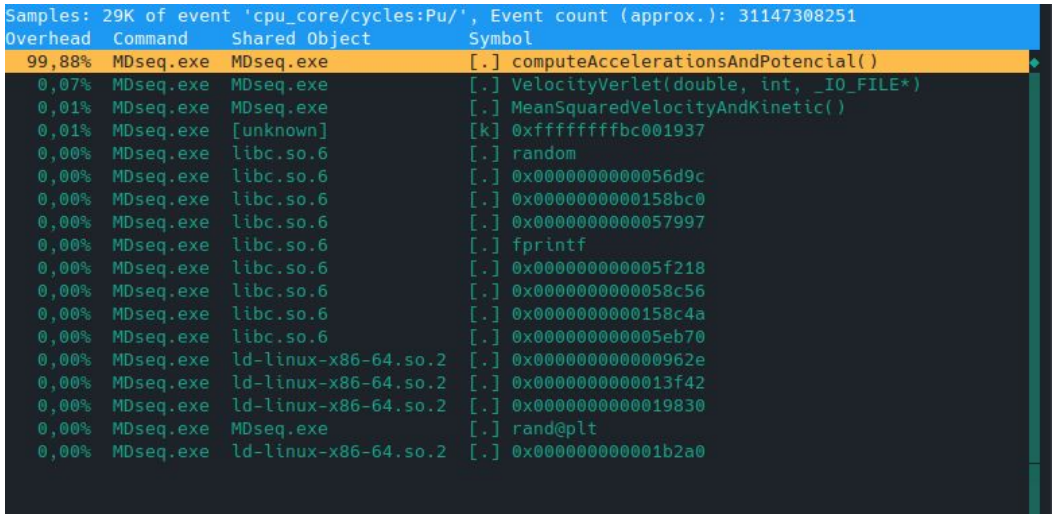
Metric	Source	Optimized
Cache misses	28473957	11795
Instructions	46255998416	32537845062
Cycles	782545448209	19379819430
Time Elapsed	269.721383703 seconds	6.015 seconds

Best performance achieved using the **-O3 -funroll-loops -ffast-math** flags combined

Shared Memory Parallelism

Profiling

perf record output:



hotspot

Preparation for parallelization

```
int numThreads = omp_get_max_threads();
double** aTemp;

aTemp=(double**)malloc(numThreads * sizeof(double*));

for (i = 0; i < numThreads; i++) {
       aTemp[i]=(double*)calloc(N * 3, sizeof(double));
}
```

why calloc:

- contiguous memory allocation -> shorter jumps in memory
- default value per index is 0 -> no need of initialization

Trivial data races:
access of ***a*** hence the use of ***aTemp*** for each thread
Pot hence the use of the reduction primitive

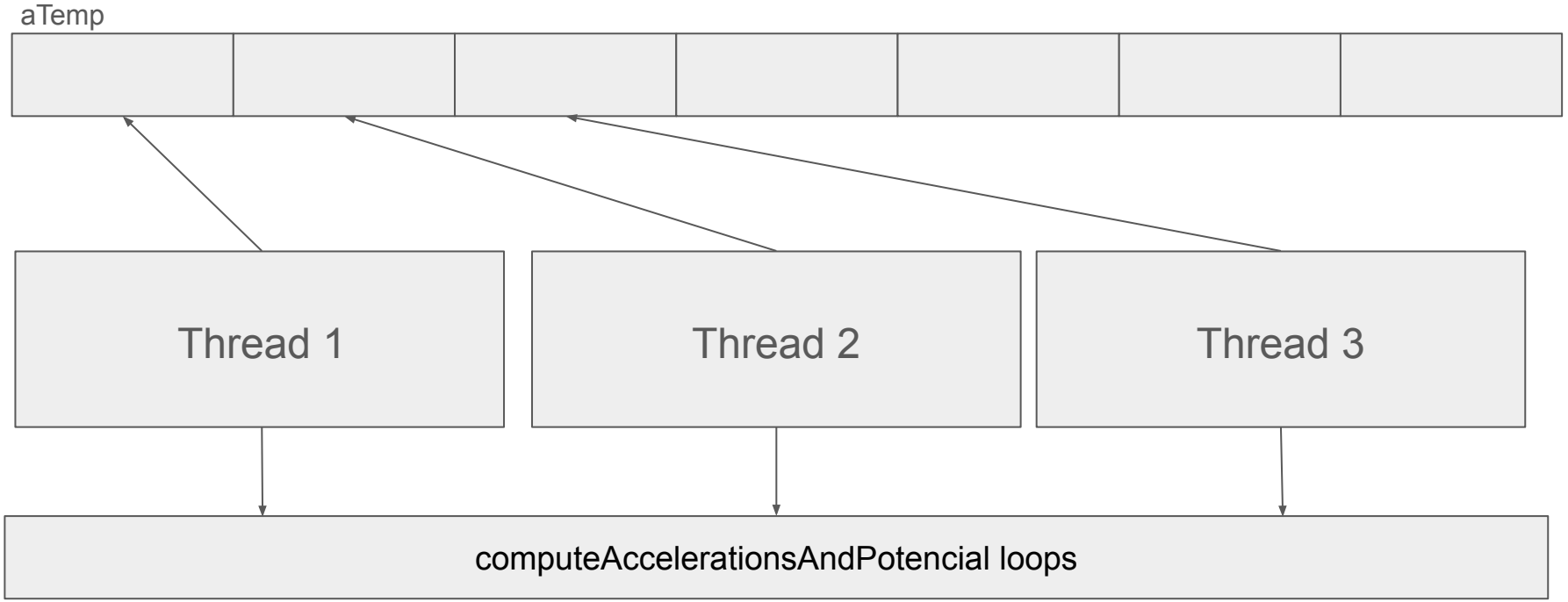
Parallelization using OpenMP

```
#pragma omp parallel num_threads(numThreads)\
private(privateVars)
{
	threadNum = omp_get_thread_num();

	#pragma omp for reduction(+:PEA)\
	schedule(dynamic, numThreads)

	for (i = 0; i < N * 3 - 4; i += 3) {
		for (j = i + 3; j < N * 3; j += 3) {
			// auxiliary computing
			// storing the values
			// in the auxiliary matrix
			aTemp[threadNum][i] += tempx * f;
			aTemp[threadNum][j] -= tempx * f;
		}
	}
}
```

Parallelization using OpenMP

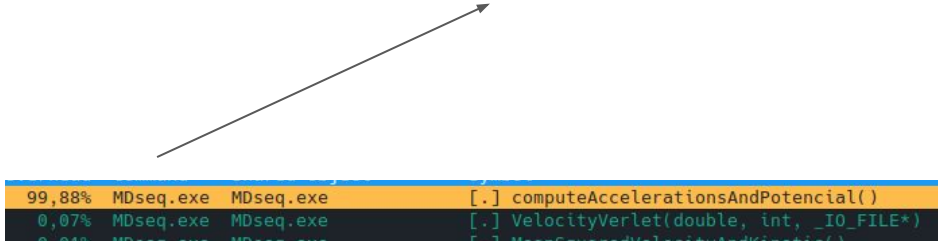


Expectations vs Results

recalling Amdahl's Law

$$\text{Speedup} \leq \frac{1}{(1-P) + \frac{P}{S}}$$

$$\text{Speedup} \leq \frac{1}{(1-0.9988) + \frac{0.9988}{S}}$$



Expectations

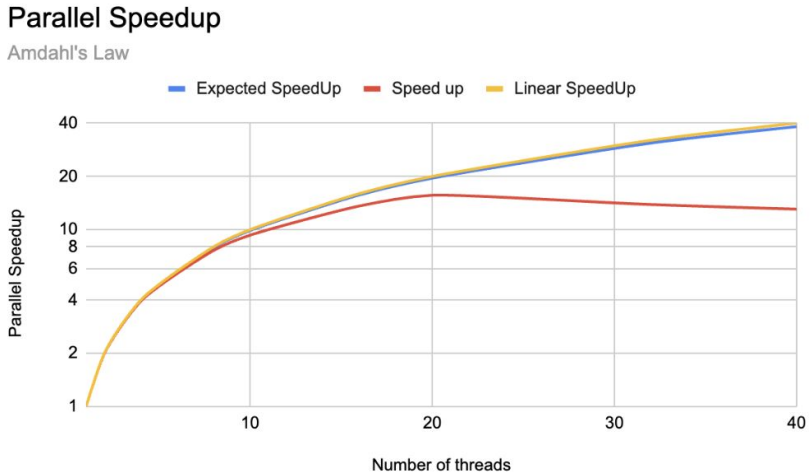
PERFORMANCE METRICS FOR DIFFERENT NUMBERS OF THREADS

Num. Threads	Expected Speedup
1	1
2	1.997
4	3.985
8	7.933
10	9.893
16	15.717
20	19.554
32	30.852
40	38.211

Results

PERFORMANCE METRICS MEASURED FOR DIFFERENT NUMBERS OF THREADS

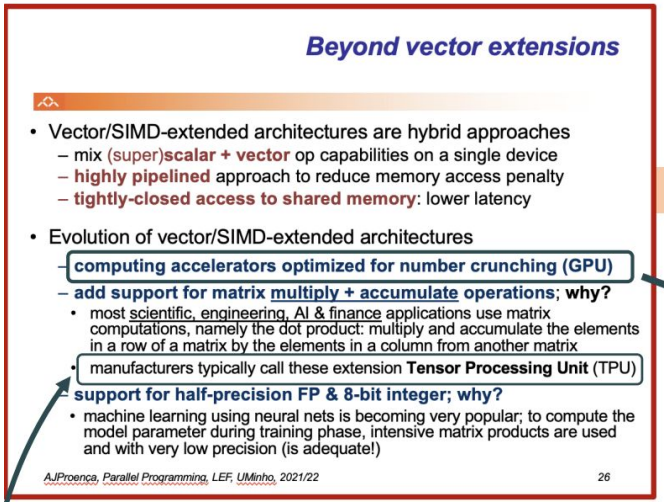
Num. Threads	Avg. execution time	Speed Up
1	35,071	1,00
2	17,633	1,99
4	8,902	3,94
8	4,595	7,63
10	3,767	9,31
16	2,577	13,61
20	2,240	15,65
32	2,528	13,87
40	2,684	13,06



GPU acceleration with CUDA

GPU acceleration with CUDA

Why CUDA?



Compute accelerators

Best accelerator for number crunching, namely intensive vector/matrix computing:
GPU

Approach

Trivial data races:

read and write in array ***a***
write to value of the potential energy

Solutions:

change ***a*** from vector to matrix
use of a vector to store the potential of each particle/atom

Implementation

Creation of CUDA Kernel(computeAccelerationsPotential)

Number of Blocks

```
const int NUM_BLOCKS = (N + NUM_THREADS_PER_BLOCK - 1)/ NUM_THREADS_PER_BLOCK;
```

Number of Threads per Block

trial and error between 128, 256 and 512

Implementation

Creation of CUDA Kernel(computeAccelerationsPotential)

Shared Memory
sharedRk - array dedicated to caching particle positions (rk) within each block.
- mitigates global memory access latency

Shared Memory Initialization
sharedRk[threadIdx.x][k] = rk_[i][k] - preloading, each thread loads a portion of the positions array rk into shared memory.
- facilitates faster access to particle positions during subsequent calculations.

Calculating Relative Positions
rij - stores relative positions between the current thread's particle instead of TempX, TempY and TempZ.

Implementation

Creation of CUDA Kernel(computeAccelerationsPotential)

Storing Results in Global Memory
Pot[i], ak[i][0], ak[i][1], ak[i][2] - store computed results in global memory arrays (Pot and ak)

Shared vs. Global Memory
Shared memory (sharedRk) optimization is directed towards enhancing access to particle positions within a block

Each particle calculates its potential energy using a local variable(vPot_local).
Sharing potentials would introduce synchronization and communication overhead, diminishing efficiency. Need manual sync hence the use of *POT[NUM_THREADS]*

Result Analysis

NORMALIZED PERFORMANCE METRICS FOR DIFFERENT PARTICLE QUANTITIES AND THREAD CONFIGURATIONS WITH GPU ACCELERATION

Num. Particles	Avg. exec. time (128 Threads)	Avg. exec. time (256 Threads)	Avg. exec. time (512 Threads)
5000	1.272	2.564	2.976
10000	3.153	3.586	3.5642
15000	6.945	7.025	6.786

Result Analysis - GPU usage

```
 NUMBER OF PARTICLES (unitless):          5000
==19876== Profiling application: ./bin/MDcuda
==19876== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.18%  753.81ms       202  3.7317ms  3.7202ms  3.7405ms  computeAccelerationsAndPotential(double[3]*, double[3]*, double*)
                    1.16%  8.9155ms       406  21.959us     928ns  22.688us  [CUDA memcpy HtoD]
                    0.66%  5.0823ms       404  12.579us  6.4960us  18.784us  [CUDA memcpy DtoH]
      API calls:   72.48%  832.14ms       808  1.0299ms  41.643us  3.8823ms  cudaMemcpy
                   20.35%  233.60ms         2  116.80ms  11.170us  233.59ms  cudaMemcpyToSymbol
                    3.62%  41.550ms       606  68.565us  3.8290us  623.80us  cudaMalloc
                    2.97%  34.113ms       606  56.291us  3.6900us  344.10us  cudaFree
                    0.38%  4.3895ms       202  21.730us  18.746us  83.775us  cudaLaunchKernel
                    0.10%  1.0929ms         2  546.45us  519.45us  573.45us  cuDeviceTotalMem
                    0.08%  927.69us       202  4.5920us     274ns  206.22us  cuDeviceGetAttribute
                    0.02%  217.75us         2  108.88us  46.352us  171.40us  cuDeviceGetName
                    0.00%  16.109us         2  8.0540us  3.7360us  12.373us  cuDeviceGetPCIBusId
                    0.00%  3.4450us         4     861ns     344ns  1.1260us  cuDeviceGet
                    0.00%  2.4510us         3     817ns     387ns  1.3600us  cuDeviceGetCount
                    0.00%  1.9500us         2     975ns     720ns  1.2300us  cuDeviceGetUuid
```

OpenMP vs CUDA comparison

PERFORMANCE METRICS MEASURED FOR DIFFERENT PARTICLE QUANTITIES: OPENMP VS. CUDA

Num. Particles	OpenMP (sec)	CUDA (sec)
5000	3.2	2.5245
10000	8.692	4.7488
15000	17.12	9.469

OpenMP vs CUDA comparison