OpenCL-based Approach to Heterogeneous Parallel TSP Optimization

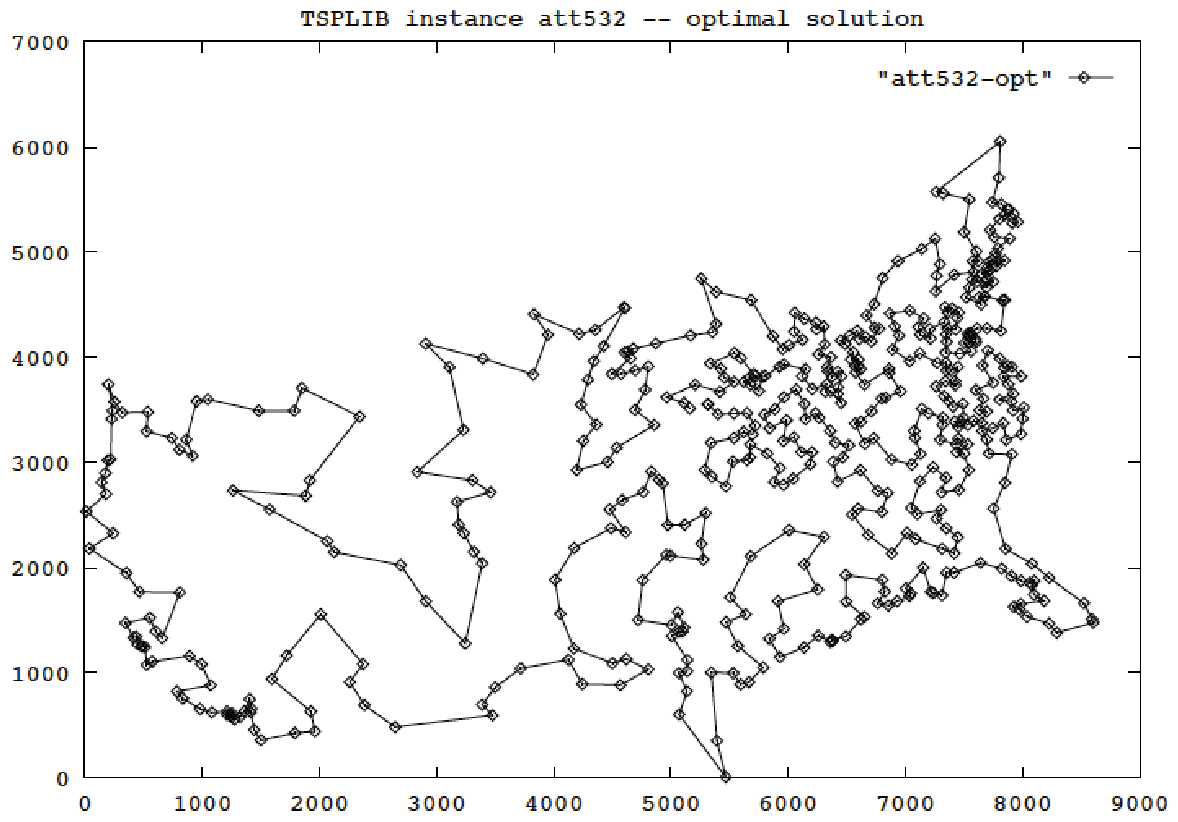
---

Kamil Rocki, PhD
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo

Why TSP?

- It's hard (NP-hard to be exact)
    - Especially to parallelize
- But easy to understand
    - A basic problem in CS
- TSPLIB - a set of standard problems
    - Allows comparisons of algorithms
- It's simple to adjust the problem size

TSP instance: shortest round trip through 532 US cities

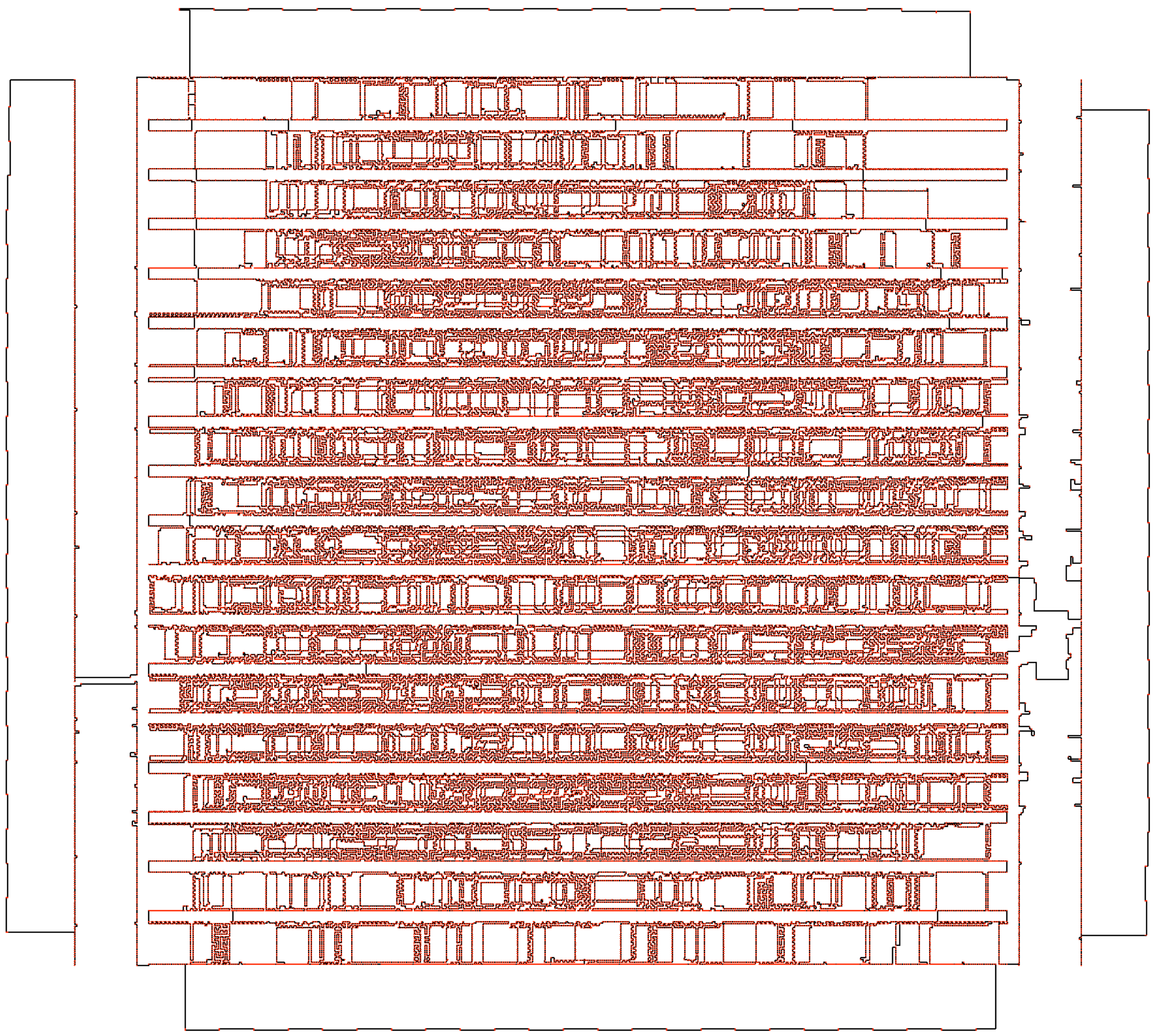


Platform for the study of general methods that can be applied to a wide range of discrete optimization problems

Not only salesmen’s headache

- Computer wiring
- Vehicle routing
- Crystallography
- Robot control

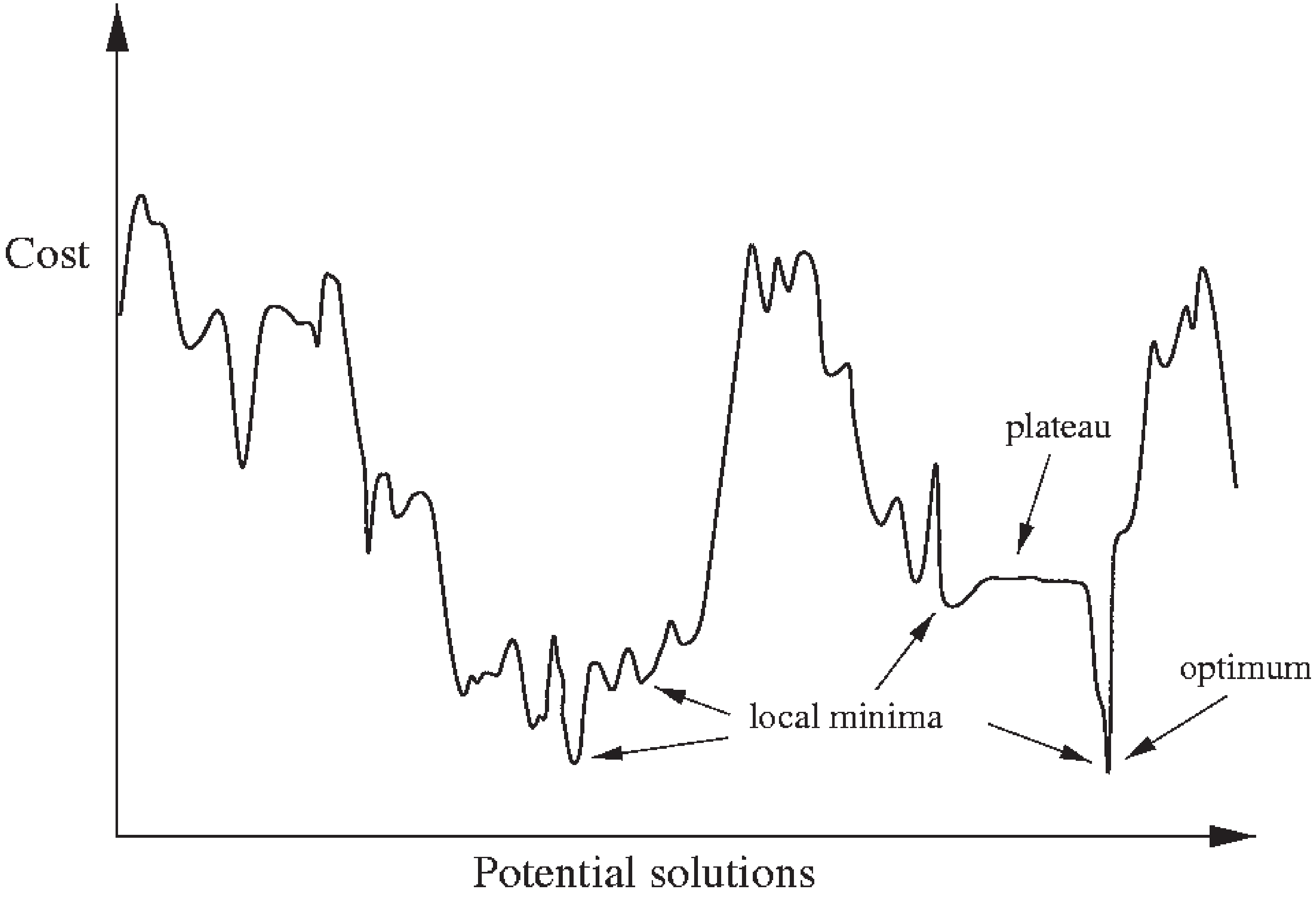
pla85900.tsp



It took 15 years to solve this problem (solved in 2006)

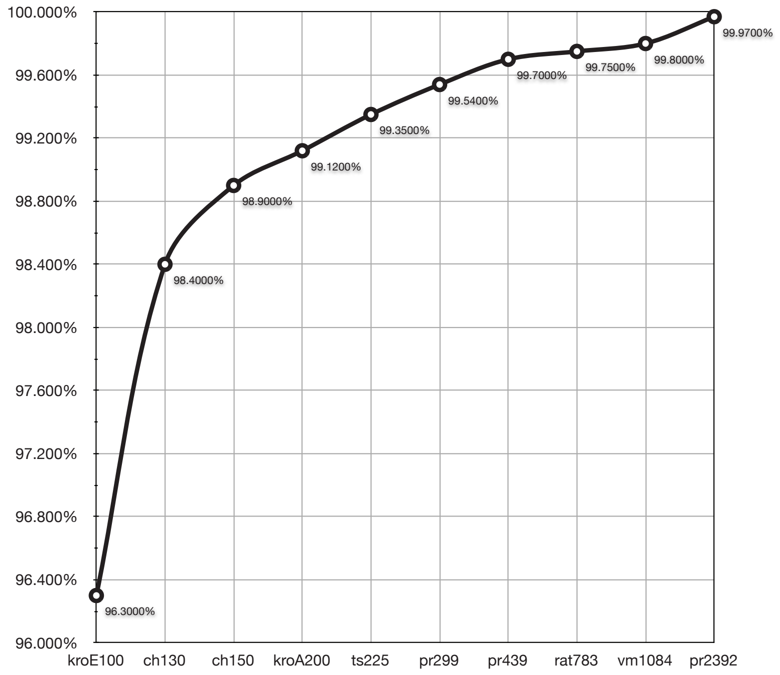
Local/Global Optimization

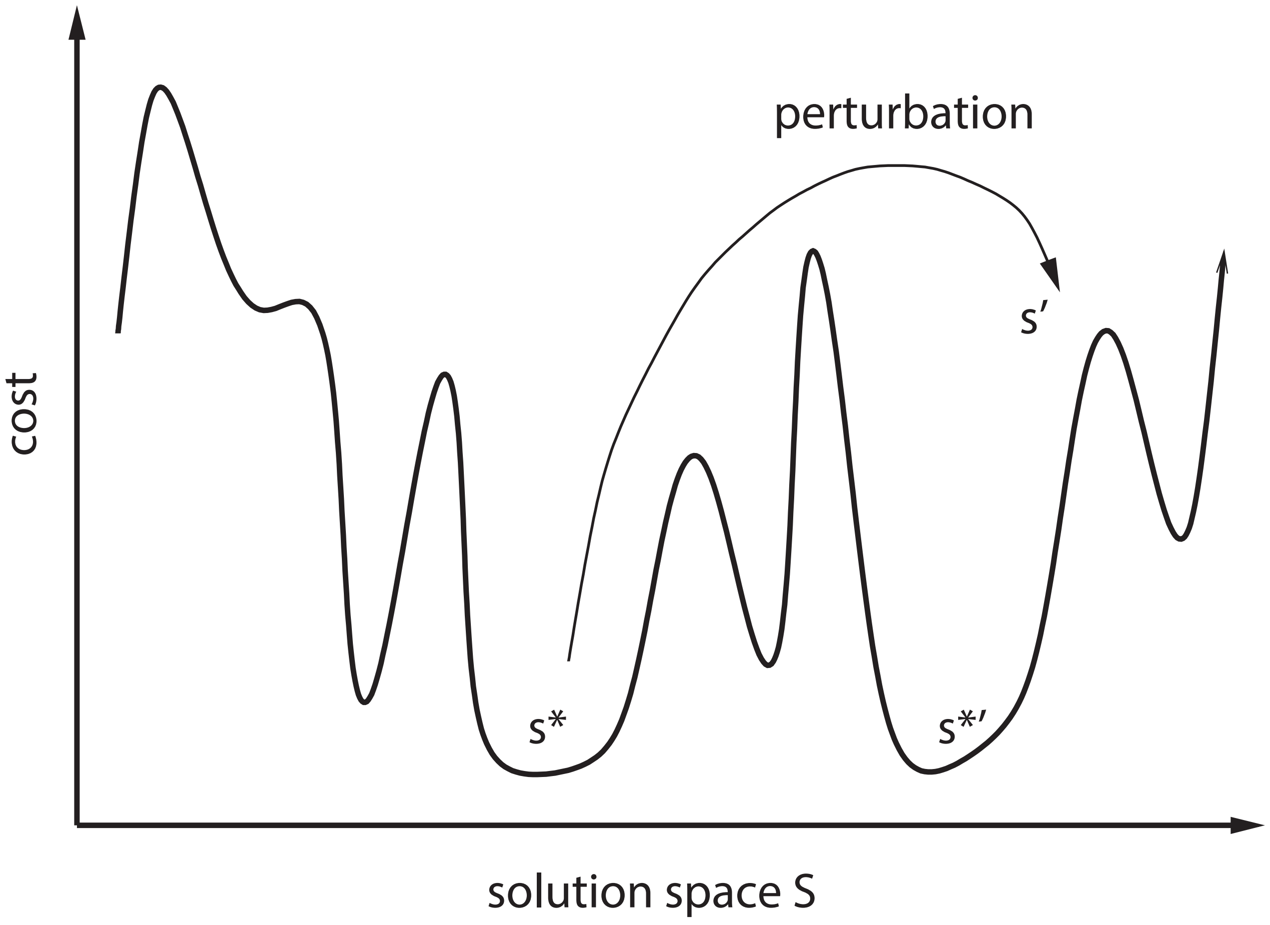
- Usually the space is huge
- There are many valleys and local minima
- Local search might terminate in one of the local minima
- Global search should find the best solution



Iterated Global Search - Global Optimization (Approximate algorithm)

- Local search is followed by some sort of tour randomization and it’s repeated (until there’s time left)
- Most of the time is spent in the ‘Local Search’ part

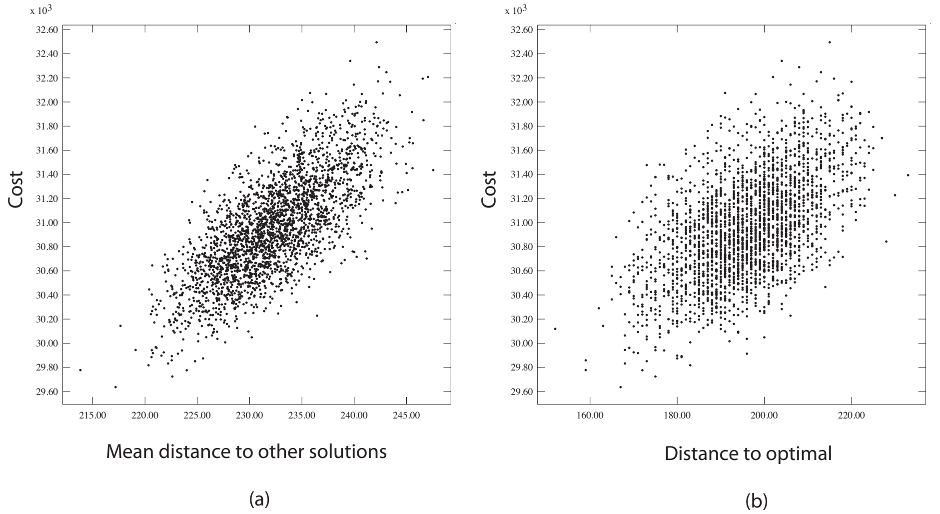




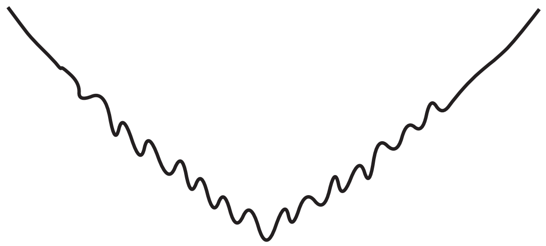
Thomas Stützle, Iterated Local Search & Variable Neighborhood Search — MN Summerschool, Tenerife, 2003 – p.9

TSP - Big valley theory (Why does ILS work?)

- Local minima form clusters
- Global minimum is somewhere in the middle
- The better the solution, the closer it is to the global optimum
- ILS worsens the tour only slightly
- Local Search is repeated close to the previous minimum



2,500 Random 2-Opt local minima for ATT532. Tour cost (vertical axis) is plotted against (a) mean distance to the other local minima and (b) distance to the global minimum.

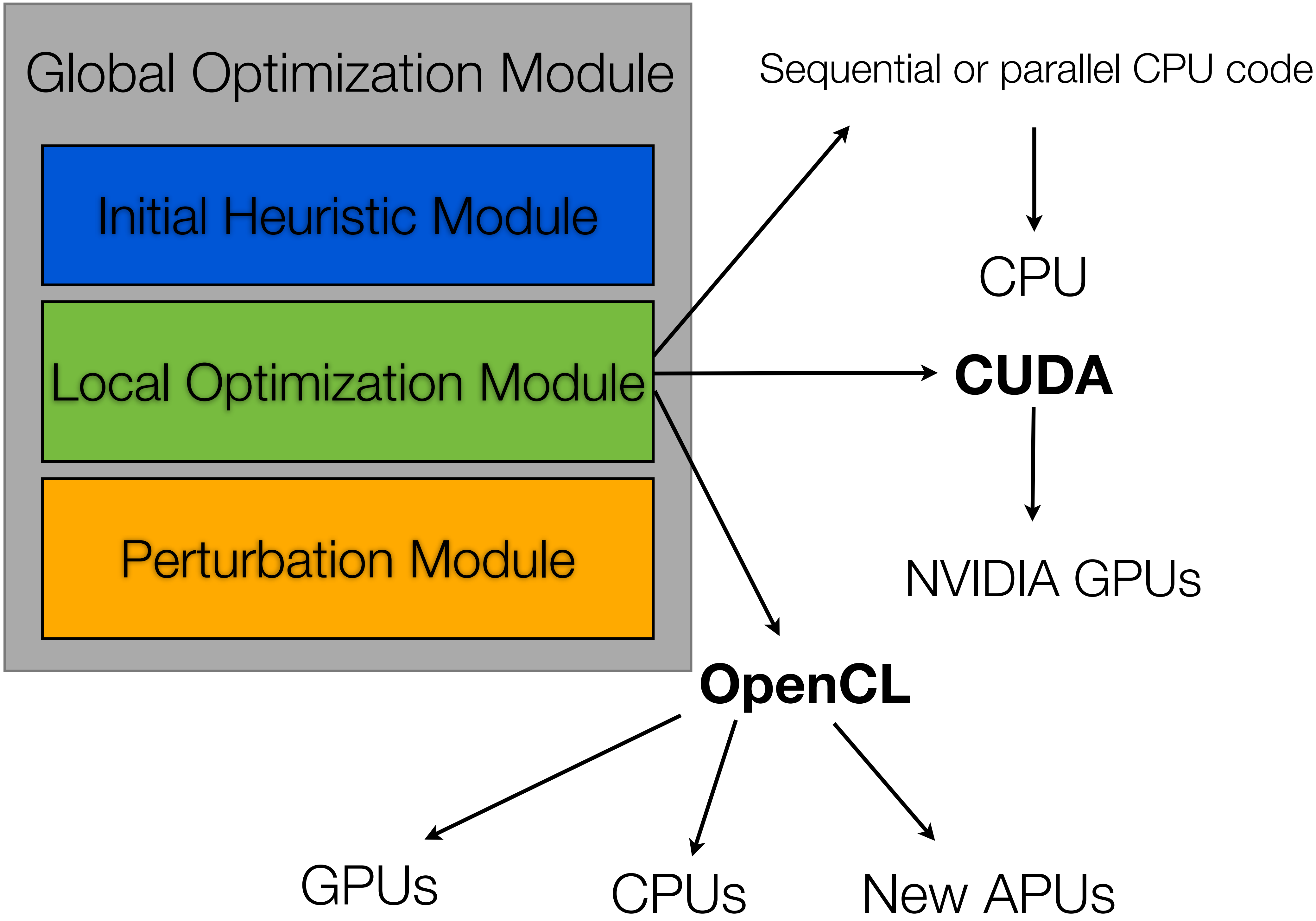


S. Baluja, A.G. Barto, K. D. Boese, J. Boyan, W. Buntine, T. Carson, R. Caruana, D. J. Cook, S. Davies, T. Dean, T. G. Dietterich, P. J. Gmytrasiewicz, S. Hazlehurst, R. Impagliazzo, A.K. Jagota, K. E. Kim, A. Mcgovern, R. Moll, A. W. Moore, L. Sanchis, L. Su, C. Tseng, K. Tumer, X. Wang, D. H. Wolpert, Justin Boyan, Wray Buntine, Arun Jagota: Statistical Machine Learning for Large-Scale Optimization, 2000

Iterated Global Search - Global Optimization (Approximate algorithm)

```
procedure ITERATED LOCAL SEARCH
    $s_0$ := GenerateInitialSolution()
    $s^*$ := 2optLocalSearch ($s_0$)        Parallel
    while (termination condition not met)
        $s'$ := Perturbation ($s^*$)
        $s^{*'}$ := 2optLocalSearch ($s'$)        Parallel
        $s^*$ := AcceptanceCriterion ($s^*, s^{*'}$)
    end while
end procedure
```

Global Optimization Module



Global Optimization Module

Initial Heuristic Module

Local Optimization Module

Perturbation Module

Sequential or parallel CPU code

CPU

CUDA

NVIDIA GPUs

OpenCL

GPUs

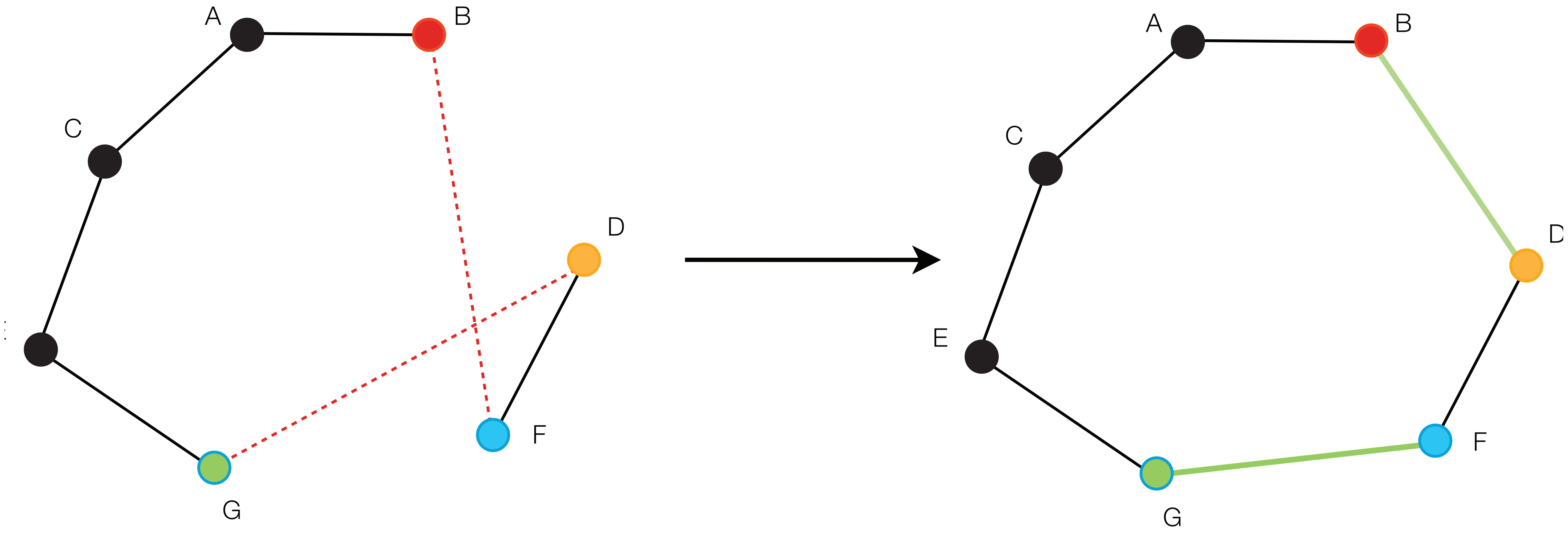
CPUs

New APUs

2-opt local optimization

- Exchange a pair of edges to obtain a better solution if the following condition is true

$$\text{distance}(B,F) + \text{distance}(G,D) > \text{distance}(B,D) + \text{distance}(G,F)$$



2-opt local optimization

- In order to find such a pair need to perform:

$$\binom{n-2}{2} = (n-2) * (n-3)/2 \text{ checks}$$

- There are some pruning techniques in more complex algorithms
- I analyze all possible pairs
    - 10000 city problem -> 49.9 Million pairs
    - 100000 city problem -> 5 Billion pairs

```
for (int i = 1; i < points - 2; i++)
    for (int j = i + 1; j < points - 1; j++)
    {
        if (distance(i, i-1) + distance(j+1, j) >
            distance(i, j+1) + distance(i-1, j))
                update best i and j;
    }

remove edges  (best i, best i-1)
          and (best j+1, best j)

add edges     (best i,best j+1)
          and (best i-1, best j)
```

Obtaining a distance

- ‘**Naive**’ approach - calculate it each time it’s needed
- ‘**Smart**’ way - reuse the results
    - For a 100-city problem it is approximately 5 times faster (CPU)

Problem (TSPLIB)	Number of cities (points)	Memory needed for LUT (MB)	Memory needed for coordinates (kB)
kroE100	100	0.038	0.78
ch130	130	0.065	1.02
ch150	150	0.086	1.17
kroA200	200	0.15	1.56
ts225	225	0.19	1.75
pr299	299	0.34	2.34
pr439	439	0.74	3.43
rat783	783	2.34	6.12
vm1084	1084	4.48	8.47
pr2392	2392	21.8	18.69
pcb3038	3038	35.21	23.73
fl3795	3795	54.9	29.65
fnl4461	4461	75.9	34.85
		n^2	n

Obtaining a distance

- ‘**Naive**’ approach - calculate it each time it’s needed
- ‘**Smart**’ way - reuse the results
  - For a 100-city problem it is approximately 5 times faster (CPU)
  - For a 100000-city problem it can be slower
    - Cache

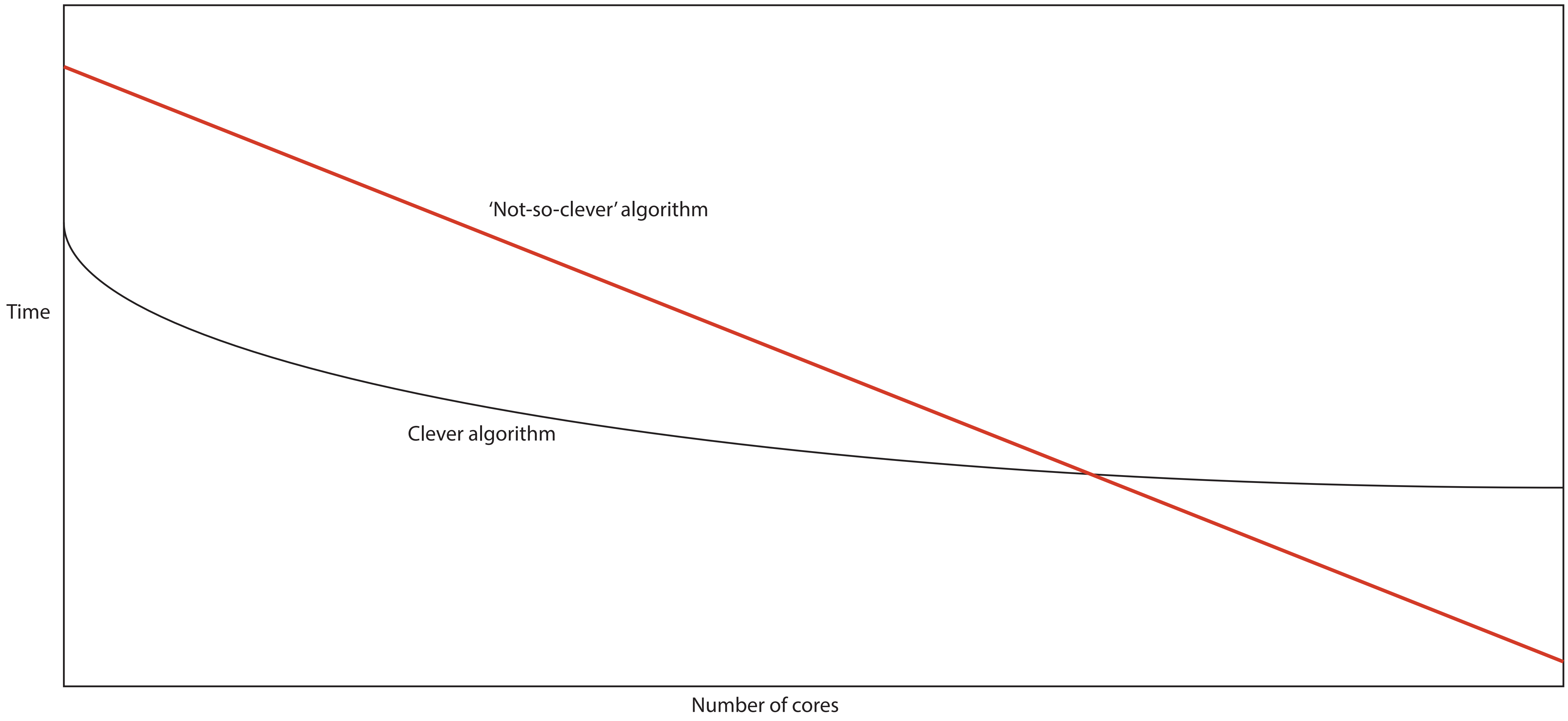
Problem (TSPLIB)	Number of cities (points)	Memory needed for LUT (MB)	Memory needed for coordinates (kB)
kroE100	100	0.038	0.78
ch130	130	0.065	1.02
ch150	150	0.086	1.17
kroA200	200	0.15	1.56
ts225	225	0.19	1.75
pr299	299	0.34	2.34
pr439	439	0.74	3.43
rat783	783	2.34	6.12
vm1084	1084	4.48	8.47
pr2392	2392	21.8	18.69
pcb3038	3038	35.21	23.73
fl3795	3795	54.9	29.65
fnl4461	4461	75.9	34.85
		n^2	n

Obtaining a distance

- ‘**Naive**’ approach - calculate it each time it’s needed
- ‘**Smart**’ way - reuse the results
    - For a 100-city problem it is approximately 5 times faster (CPU)
    - For a 100000-city problem it can be slower
        - Cache
    - With multiple cores, it becomes slower
        - Cache/memory

Problem (TSPLIB)	Number of cities (points)	Memory needed for LUT (MB)	Memory needed for coordinates (kB)
kroE100	100	0.038	0.78
ch130	130	0.065	1.02
ch150	150	0.086	1.17
kroA200	200	0.15	1.56
ts225	225	0.19	1.75
pr299	299	0.34	2.34
pr439	439	0.74	3.43
rat783	783	2.34	6.12
vm1084	1084	4.48	8.47
pr2392	2392	21.8	18.69
pcb3038	3038	35.21	23.73
fl3795	3795	54.9	29.65
fnl4461	4461	75.9	34.85
		n^2	n

LUT (Look Up Table) vs recalculation



A change in algorithm design - sequential

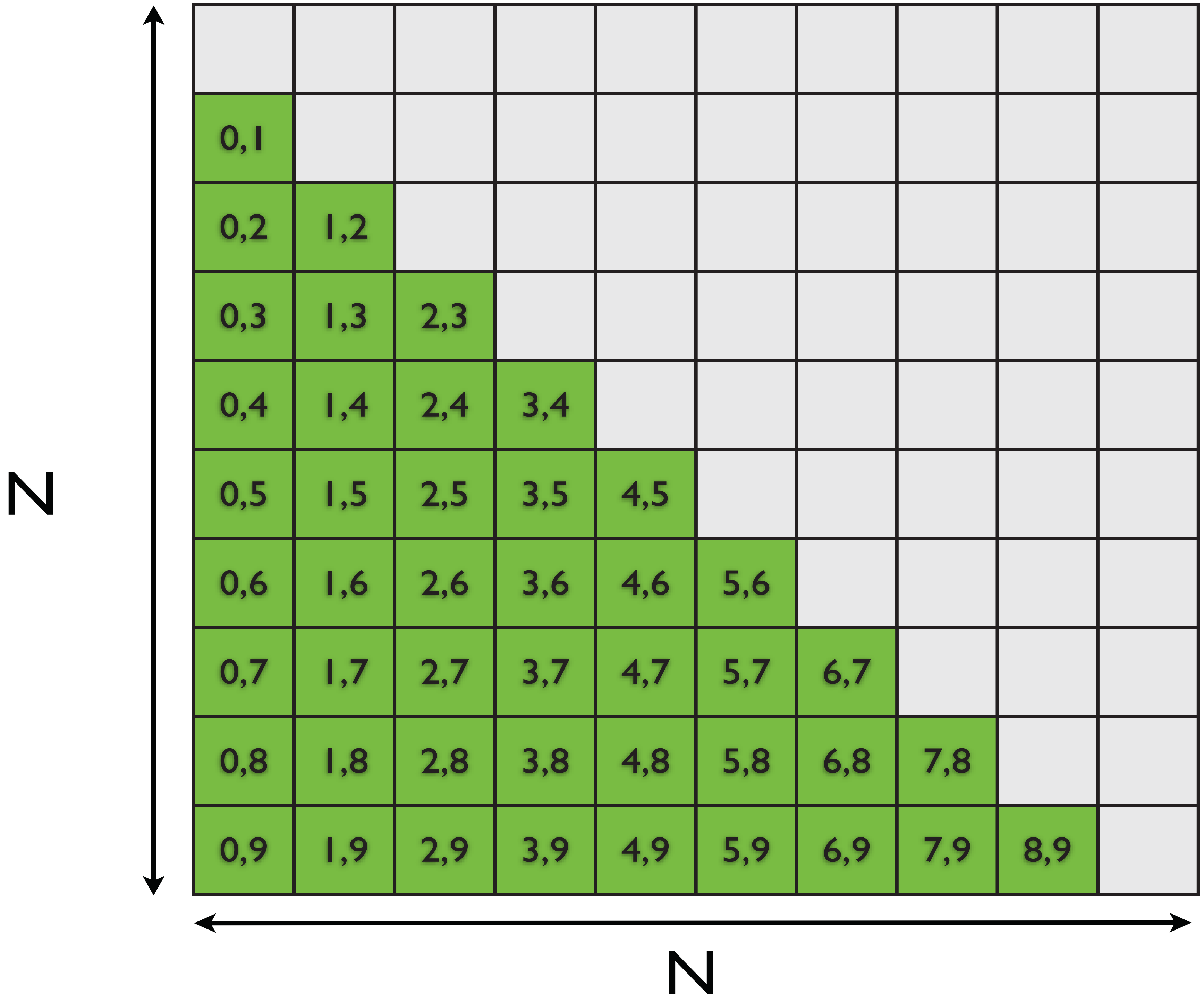
- Memory is free, reuse computed results
- Use sophisticated algorithms, prune the search space

A change in algorithm design - parallel

- ~~Memory is free, reuse computed results~~
- ~~Use sophisticated algorithms, prune the search space~~
- **Limited memory (size, throughput), computing is free (almost)**
- **Sometimes brute-force search might be faster (especially on GPUs)**
  - Avoiding divergence, simpler control-flow
  - Same amount of time to process 10, 100 even 10000 elements in parallel

Parallelization

dist(i,j) = dist (j,i), dist (i,i) = 0



0,1									
0,2	1,2								
0,3	1,3	2,3							
0,4	1,4	2,4	3,4						
0,5	1,5	2,5	3,5	4,5					
0,6	1,6	2,6	3,6	4,6	5,6				
0,7	1,7	2,7	3,7	4,7	5,7	6,7			
0,8	1,8	2,8	3,8	4,8	5,8	6,8	7,8		
0,9	1,9	2,9	3,9	4,9	5,9	6,9	7,9	8,9	

GPU implementation

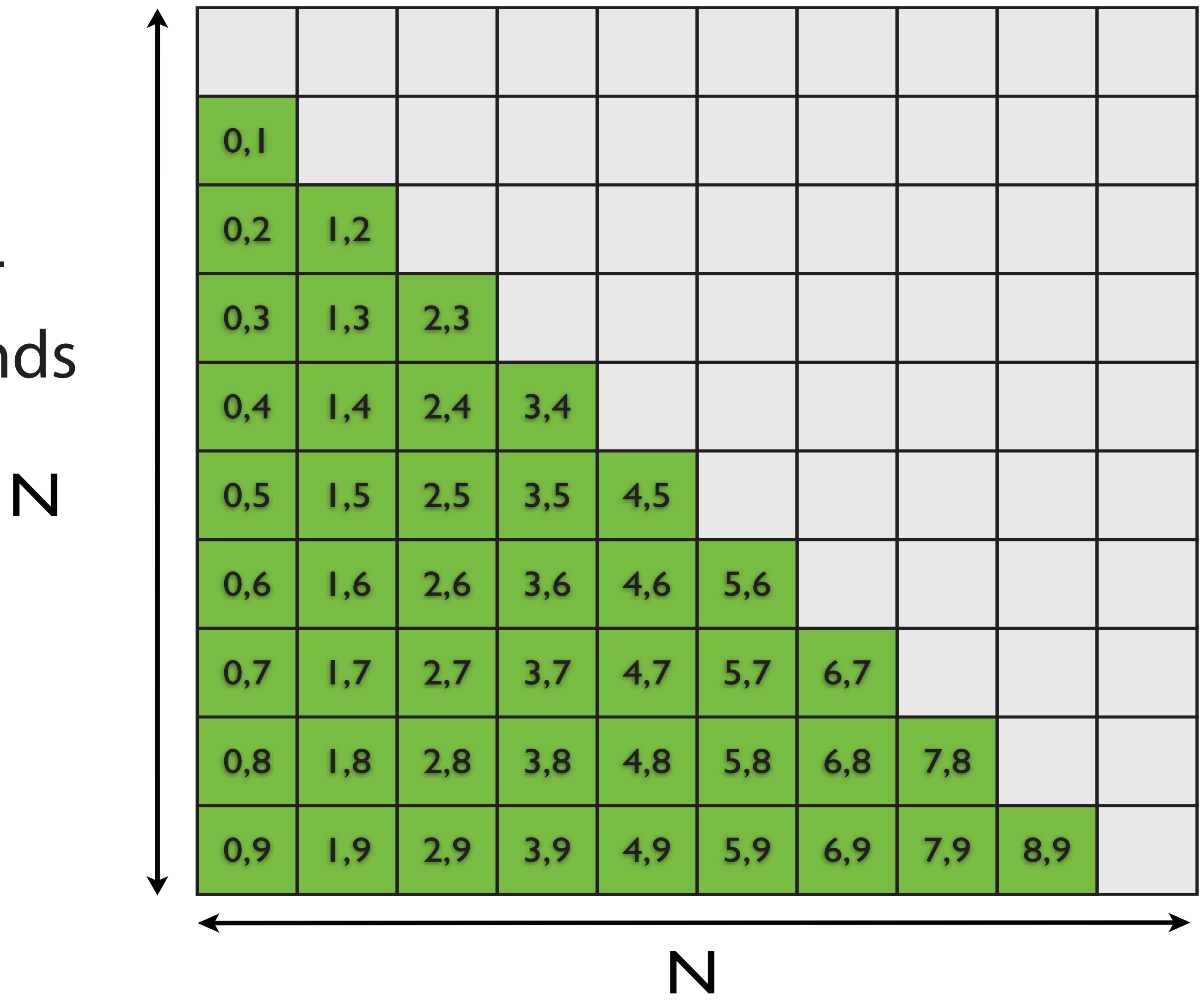
A simple function for EUC_2D problems

```
int calculateDistance2D
(unsigned int i, unsigned int j, float2* coords) {

        register float dx, dy;

        dx = coords[i].x - coords[j].x;
        dy = coords[i].y - coords[j].y;

        return (int)(sqrtf(dx * dx + dy * dy) + 0.5f);
}
//6 FLOPs + sqrt
```

GPU implementation - First approach

Matrix of city pairs to be checked for a possible swap, each pair corresponds to one GPU job

dist(i,j) = dist (j,i), dist (i,i) = 0



0,1									
0,2	1,2								
0,3	1,3	2,3							
0,4	1,4	2,4	3,4						
0,5	1,5	2,5	3,5	4,5					
0,6	1,6	2,6	3,6	4,6	5,6				
0,7	1,7	2,7	3,7	4,7	5,7	6,7			
0,8	1,8	2,8	3,8	4,8	5,8	6,8	7,8		
0,9	1,9	2,9	3,9	4,9	5,9	6,9	7,9	8,9	

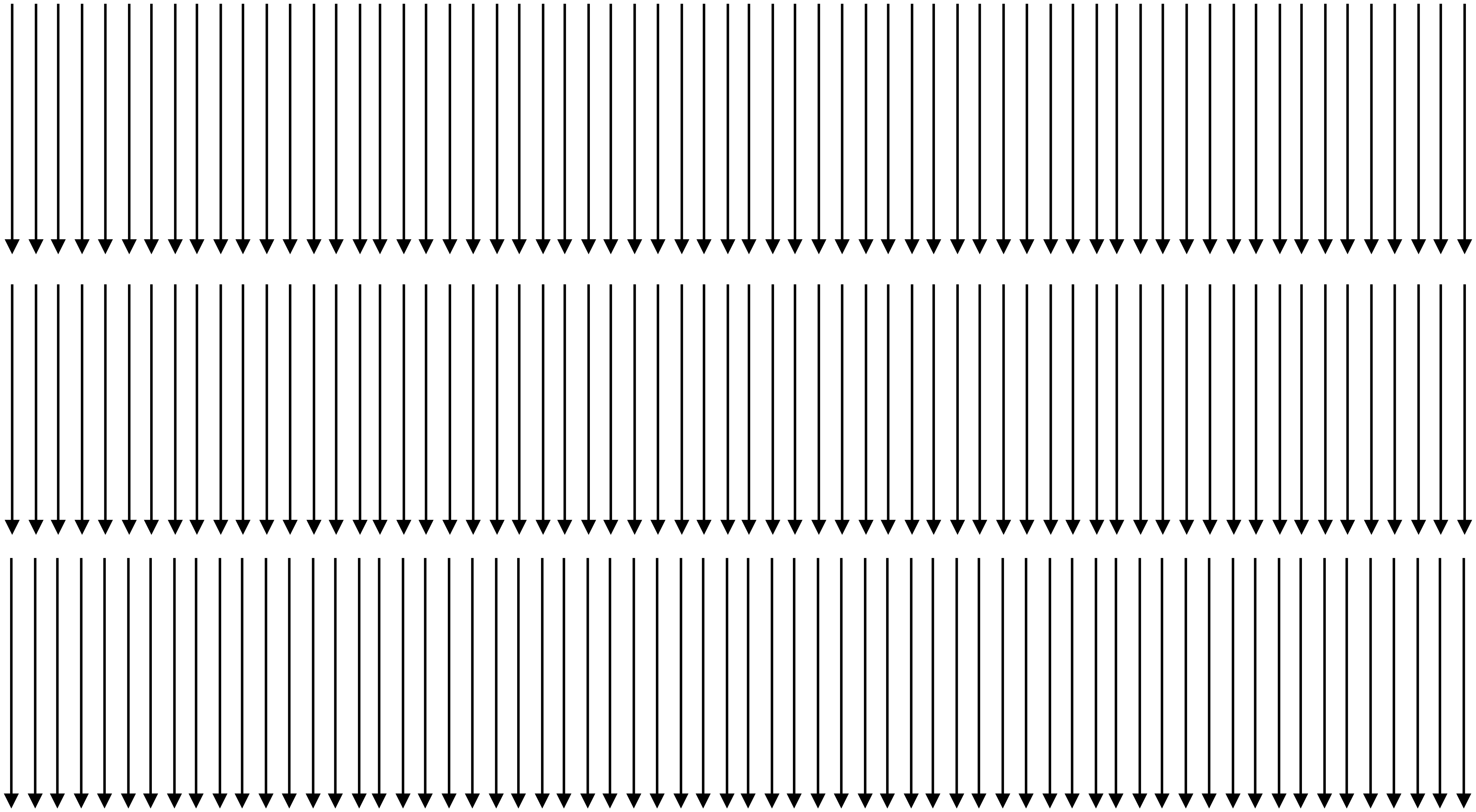
Each thread has to check:

```
if (distance(i, i-1) + distance(j+1, j) >
    distance(i, j+1) + distance(i-1, j))
        update best i and j;
```

Parallelization

- GPU-centric approach

```
dx = coords[i].x - coords[j].x;
dy = coords[i].y - coords[j].y;

sqrtf(dx * dx + dy * dy) + 0.5f;
```



Parallelization

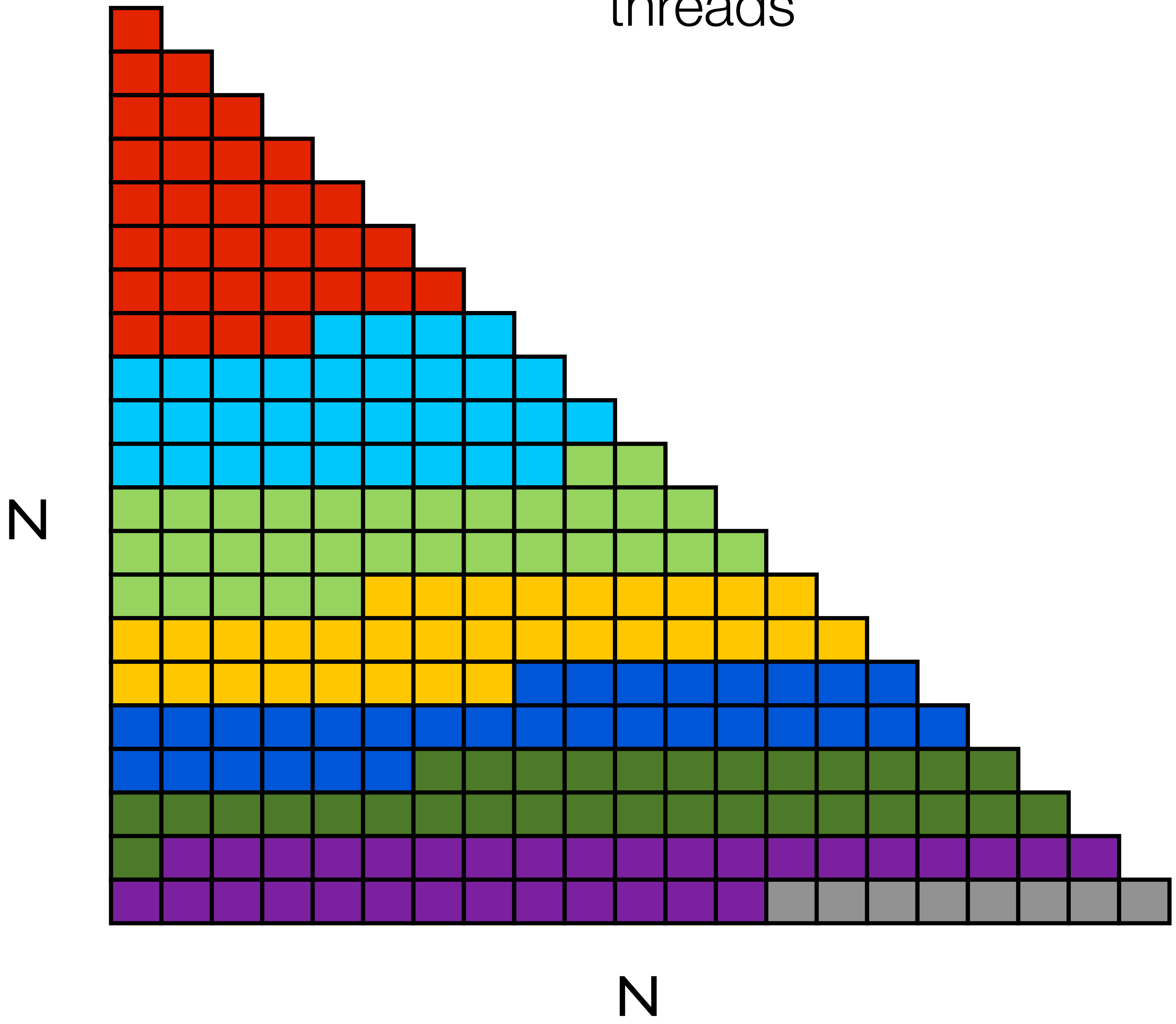
- Porting the application to OpenCL
- Implemented the code in CUDA first
  - Then ported it to OpenCL
- Later I wrote parallel CPU code
  - Multi-threaded + SSE/AVX/MIC vectorization
    - Ported that to OpenCL too

CPU implementation

Each thread has to check a range of distances

```
for (int i = start_i; i < end_i; i++) {
        for (int j = start_j; j < end_j; j += 1) {

        change = calculateDistance2D (i, j + 1) +
        calculateDistance2D (i - 1, j) -
        calculateDistance2D (i, i - 1) -
        calculateDistance2D (j + 1, j) - 0.5f);
        ....
        }
}
```

Example for **32** threads

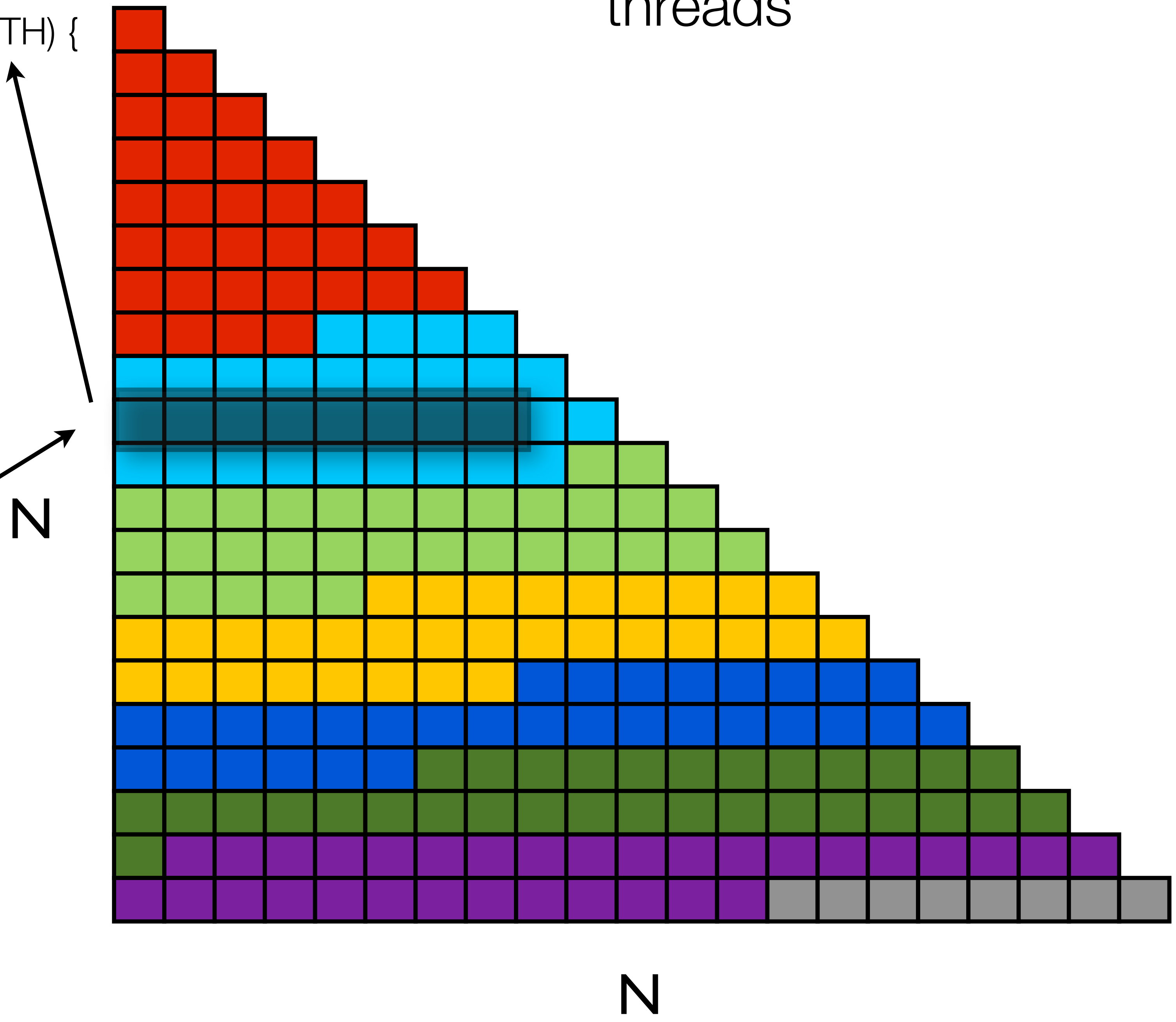


CPU implementation

Each thread has to check a range of distances

```
for (int i = start_i; i < end_i; i++) {
        for (int j = start_j; j < end_j; j += VECTOR_LENGTH) {

                x = vloadn(offset, address);
                y = vloadn(offset, address);
                /*operate on float4, float8 or float16
                calculate the distances in groups*/
        }
}
```

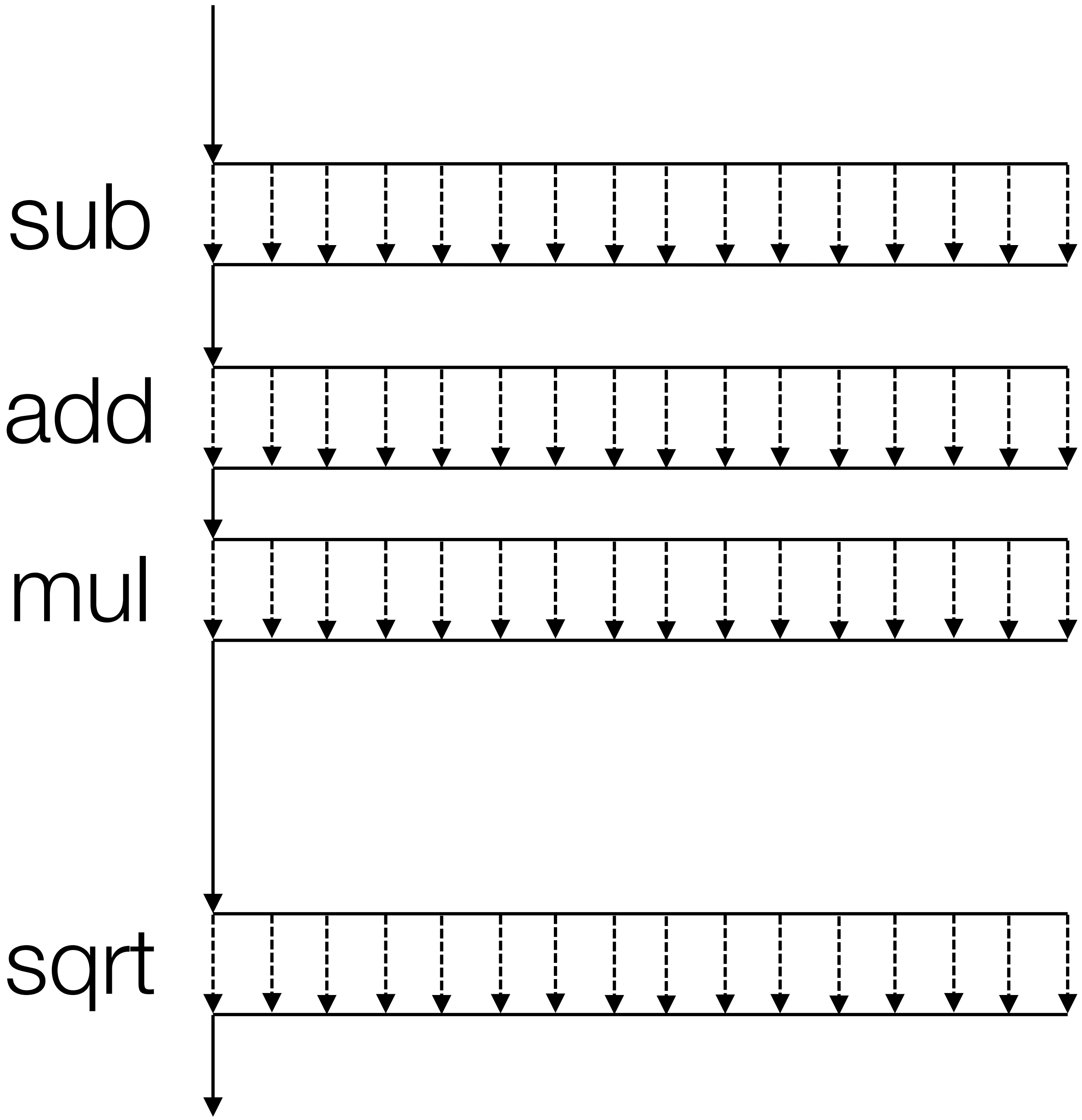
Example for **32** threads



SIMD processing where possible

Parallelization

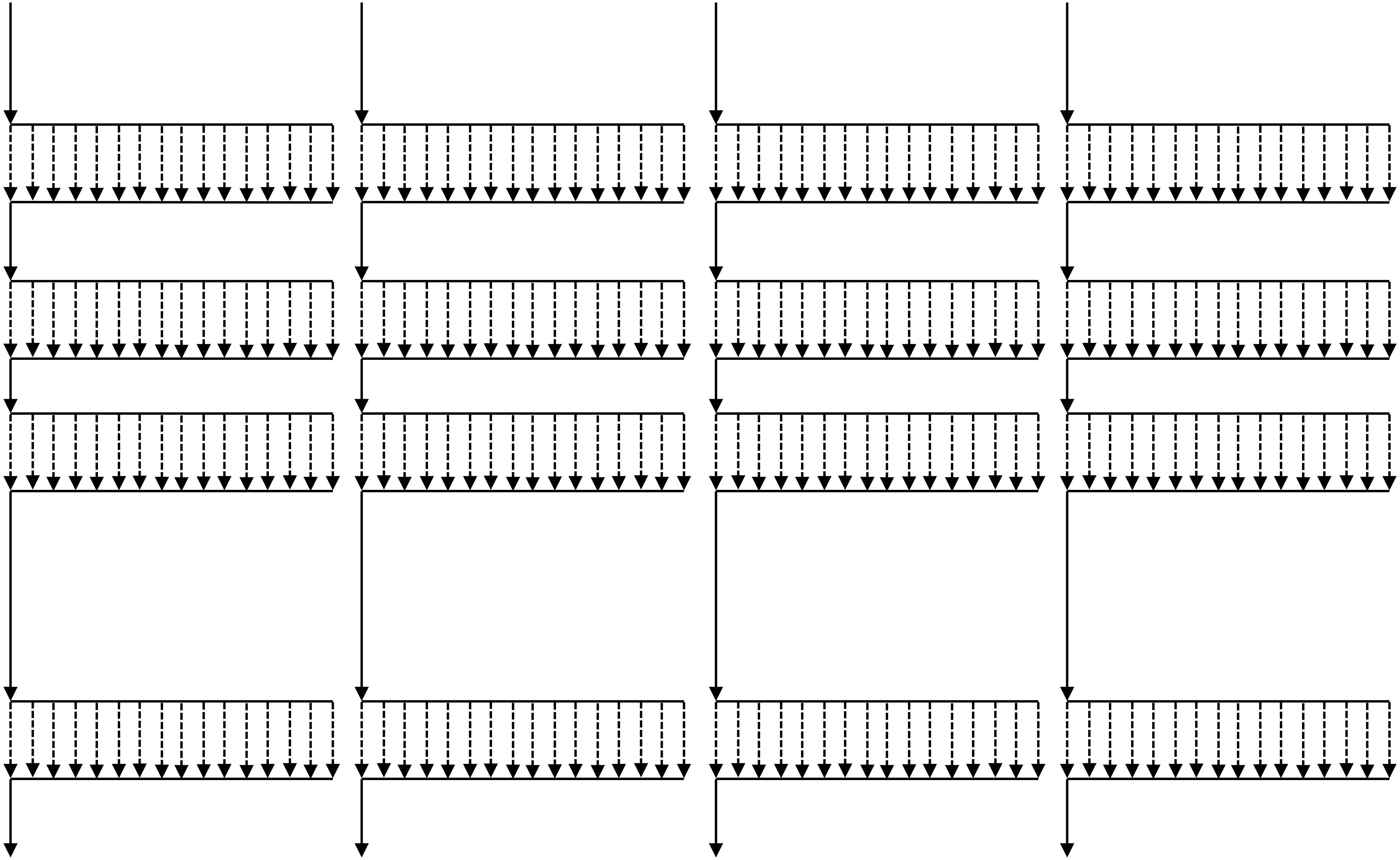
- CPU-centric approach
- I.e process 16 pairs of coordinates at once



```
#ifdef USE_512BIT_VECTORS
#define VEC16F __m512
#define SUB16F(x,y) _mm512_sub_ps(x,y)
#define ADD16F(x,y) _mm512_add_ps(x,y)
#define MUL16F(x,y) _mm512_mul_ps(x,y)
#define MIN16F(x) _mm512_reduce_gmin_ps(x)
#define SQRT16F(x) _mm512_sqrt_ps(x)
#define FAST_SQRT16F(x) _mm512_mul_ps(x,_mm512_rsqrt23_ps(x))
```

Parallelization

- CPU-centric approach - multiple threads



Code translation

- CUDA -> very simple, mostly find and replace, 30 minutes?

CUDA

```
barrier (CLK_LOCAL_MEM_FENCE);
    id = get_global_id (0) + no * get_global_size (0);

    if (id < max) {
        // a nasty formula to calculate the right cell of a
        // triangular matrix index based on the thread id
        j = (unsigned int) (3 + native_sqrt (8.0f * (float) id + 1.0f) ) /2;
        i = id - (j - 2) * (j - 1) / 2 + 1;
        // calculate the effect of (i,j) swap
        change =  calculateDistance2DSimple(i, j + 1, localCoords) +
                calculateDistance2DSimple(i - 1, j, localCoords) -
                  calculateDistance2DSimple(i, i - 1, localCoords) -
                  calculateDistance2DSimple(j + 1, j, localCoords);
       // save if best than already known swap
          .....
}
```

OpenCL

```
__syncthreads();
for (register int no = 0; no < iter; no++) {
    id = local_id + no * packSize;

        // a nasty formula to calculate the right cell of a triangular matrix
        // index based on the thread id
        j = (unsigned int) (3 + __fsqrt_rn (8.0f * (float) id + 1.0f) ) / 2;
        i = id - (j - 2) * (j - 1) / 2 + 1;
        // calculate the effect of (i,j) swap
        change = calculateDistance2DSimple (i, j + 1, coords) +
                 calculateDistance2DSimple (i - 1, j, coords) -
                 calculateDistance2DSimple (i, i - 1, coords) -
                 calculateDistance2DSimple (j + 1, j, coords);

        // save if best than already known swap
        .....
}
```

Code translation

- SSE/AVX code -> simple, 1 hour?, the code can be the same if MACROs are used

256-bit

```
#define VECF __m256
#define SUBF(x,y) _mm256_sub_ps(x,y)
#define ADDF(x,y) _mm256_add_ps(x,y)
#define MULF(x,y) _mm256_mul_ps(x,y)
#define SQRTF(x) _mm256_sqrt_ps(x)
```

→

OpenCL

```
#define VECF float8
#define SUBF(x, y) ((x) - (y))
#define ADDF(x, y) ((x) + (y))
#define MULF(x, y) ((x) * (y))
#define SQRTF(x) native_sqrt(x)
```

OR

512-bit

```
#define VECF __m512
#define SUBF(x,y) _mm512_sub_ps(x,y)
#define ADDF(x,y) _mm512_add_ps(x,y)
#define MULF(x,y) _mm512_mul_ps(x,y)
#define MINF(x) _mm512_reduce_gmin_ps(x)
#define SQRTF(x) _mm512_sqrt_ps(x)
```

→

```
#define VECF float16
#define SUBF(x, y) ((x) - (y))
#define ADDF(x, y) ((x) + (y))
#define MULF(x, y) ((x) * (y))
#define SQRTF(x) native_sqrt(x)
```

Later

```
dx1 = SUBF (coordsA1X, coordsB1X);
dy1 = SUBF (coordsA1Y, coordsB1Y);
dx1 = MULF (dx1, dx1);
dy1 = MULF (dy1, dy1);
dx1 = ADDF (dx1, dy1);
dx1 = SQRTF (dx1);
 ....
```

So why is OpenCL implementation important?

- Writing optimized CPU/GPU code takes weeks
    - Redesigning the algorithm, tuning...
- Having a portable and efficient OpenCL code may save some time
- I may target new devices without re-implementing the code

Results

Intel Xeon(R) CPU E5-2690

Native
1 core: 1.8 GFLOP/s
16 cores: 25.61 GFLOP/s
1 core SIMD (AVX): 11.60 GFLOP/s
16 cores SIMD (AVX): 184.89 GFLOP/s 49.6%

OpenCL
CPU-centric: 158.72 GFLOP/s 42.8%
GPU-centric: 31.46 GFLOP/s

Intel Xeon Phi (5110P):

Native
1 core: 0.7 GFLOP/s
60 cores: 41.2 GFLOP/s
1 core SIMD: 8.1 GFLOP/s
60 cores SIMD: 442.68 GFLOP/s 43.5%

OpenCL
GPU-centric: 7.27 GFLOP/s
CPU-centric: **`clBuildProgram`** segfault

GPU-centric - CUDA port
CPU-centric - Xeon Phi port

AMD Radeon 7970 1GHz

OpenCL
GPU-centric: 848.15 GFLOP/s 21.2%
CPU-centric: **`clBuildProgram`** segfault

NVIDIA GTX Titan

CUDA: 1193.30 GFLOP/s 25.3%

OpenCL
CPU-centric: 1137.49 GFLOP/s 24.2%
GPU-centric: 755.82 GFLOP/s 16.1%

1000 GFLOP/s = approximately
40 billion edge-pairs comparisons per second

Results

100000-city problem:
5 Billion pairs of edges ~ 155 GFLOPs in total

Intel Xeon(R) CPU E5-2690

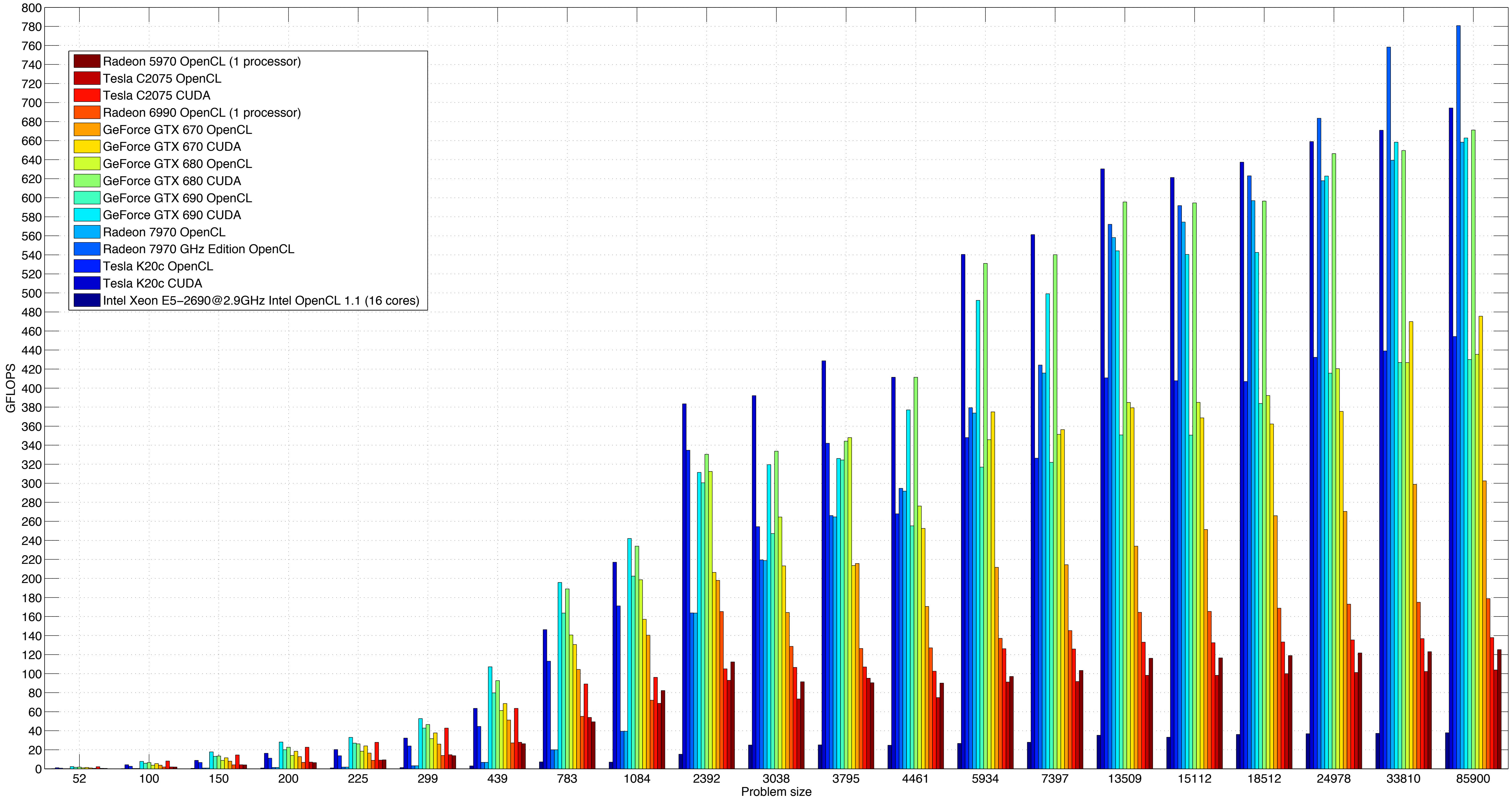
1 core, No SIMD: 86s
16 cores, AVX: 0.84s
OpenCL: 0.98s

AMD Radeon 7970 OpenCL: 0.18s

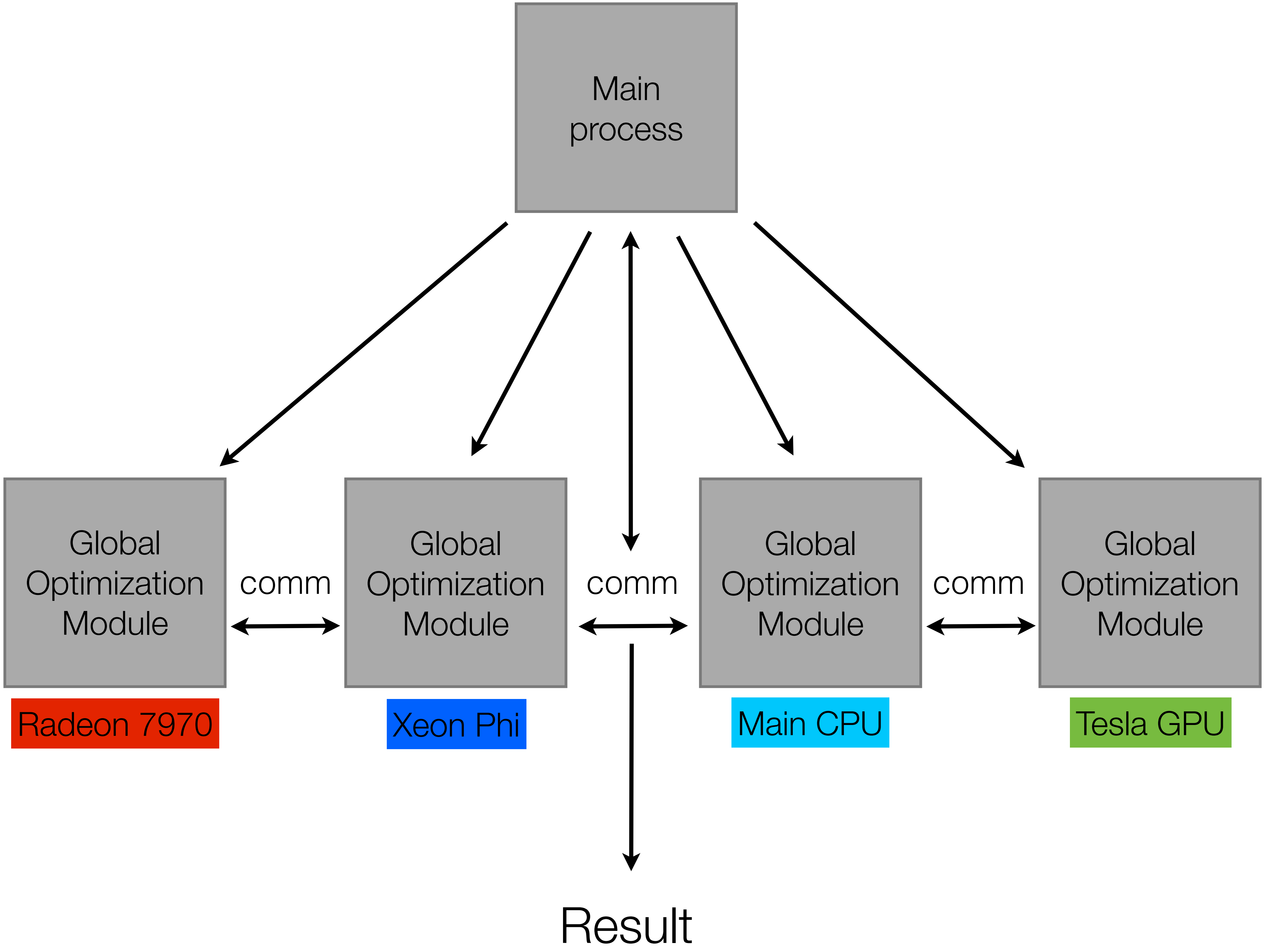
NVIDIA GTX Titan OpenCL: 0.14s

Results - I tested many GPUs and CPUs

I need to include the latest OpenCL implementation



Inter-device cooperation



Main process

Global Optimization Module

comm

Global Optimization Module

comm

Global Optimization Module

comm

Global Optimization Module

Radeon 7970

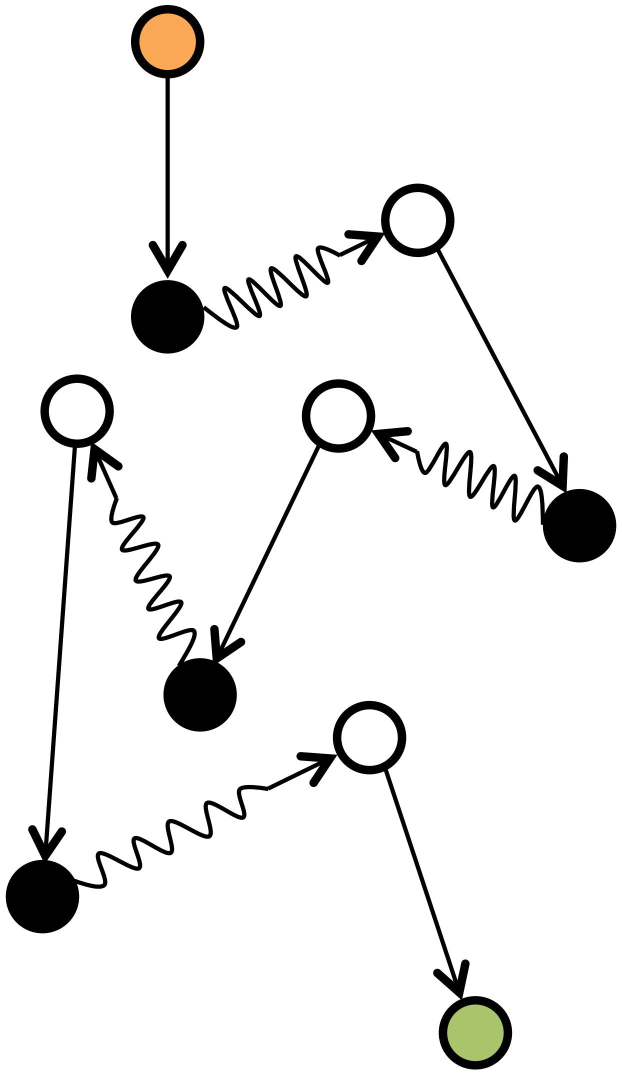
Xeon Phi

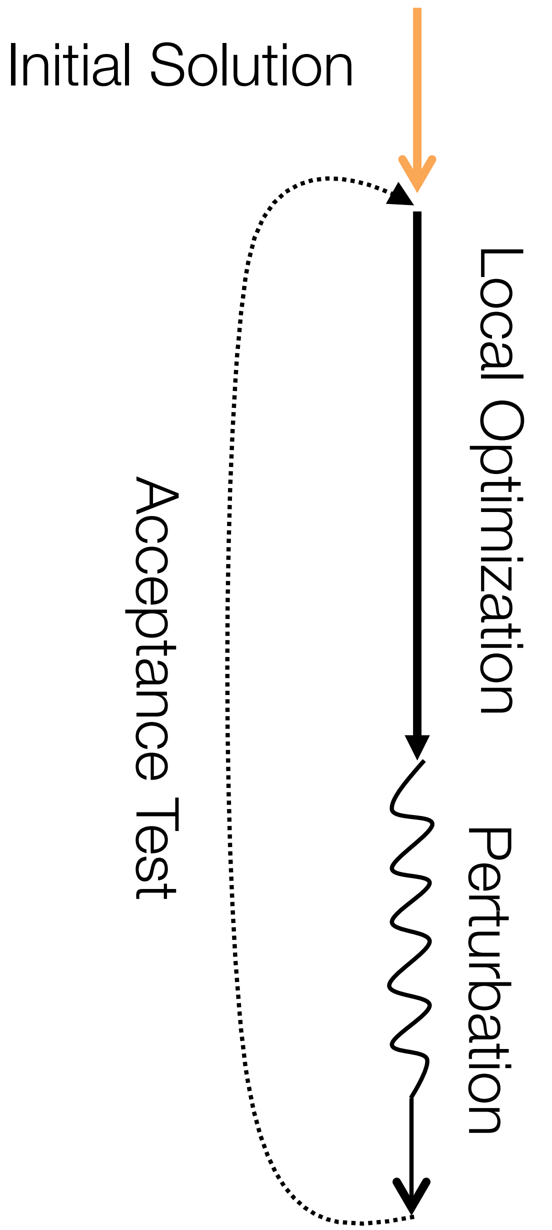
Main CPU

Tesla GPU

Result

Inter-device cooperation





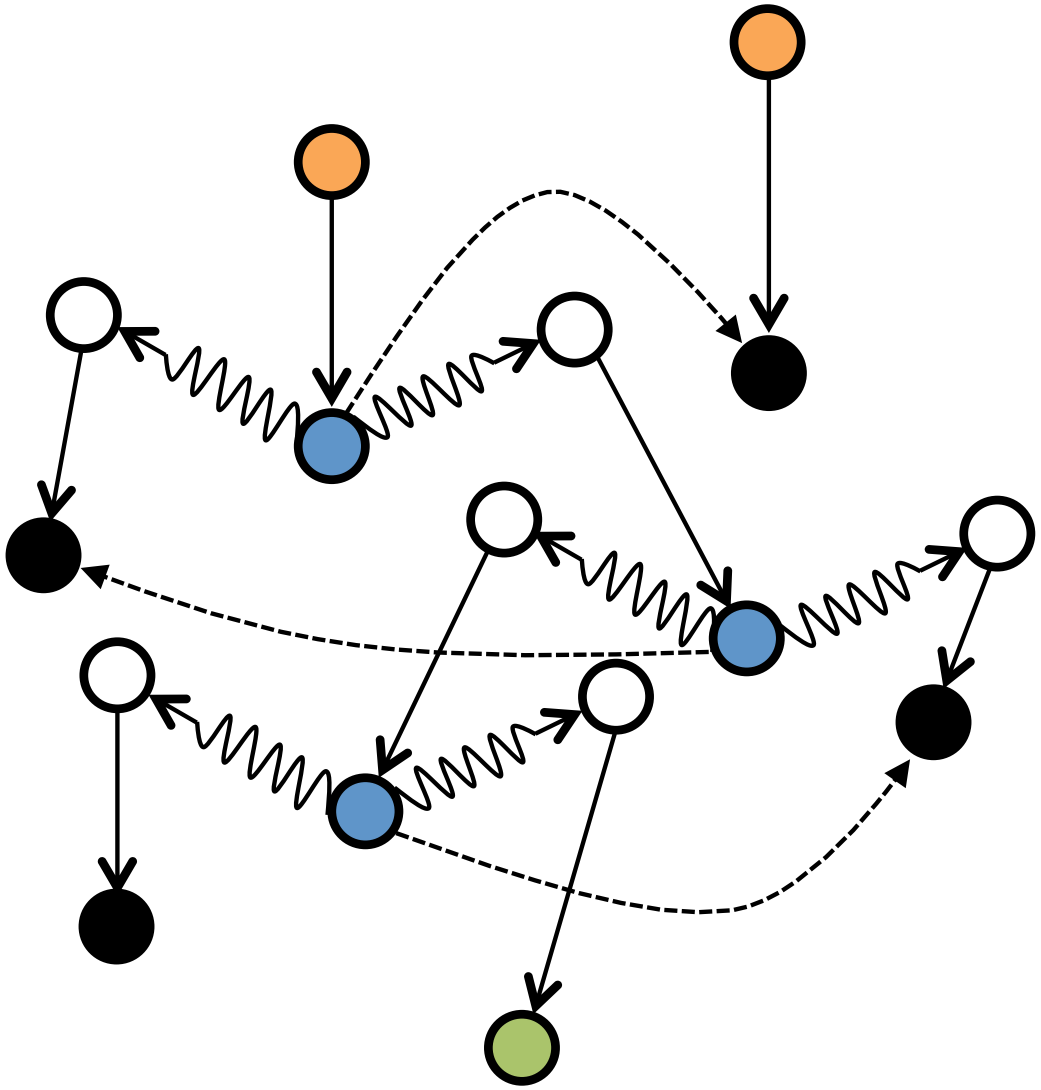
Initial Solution

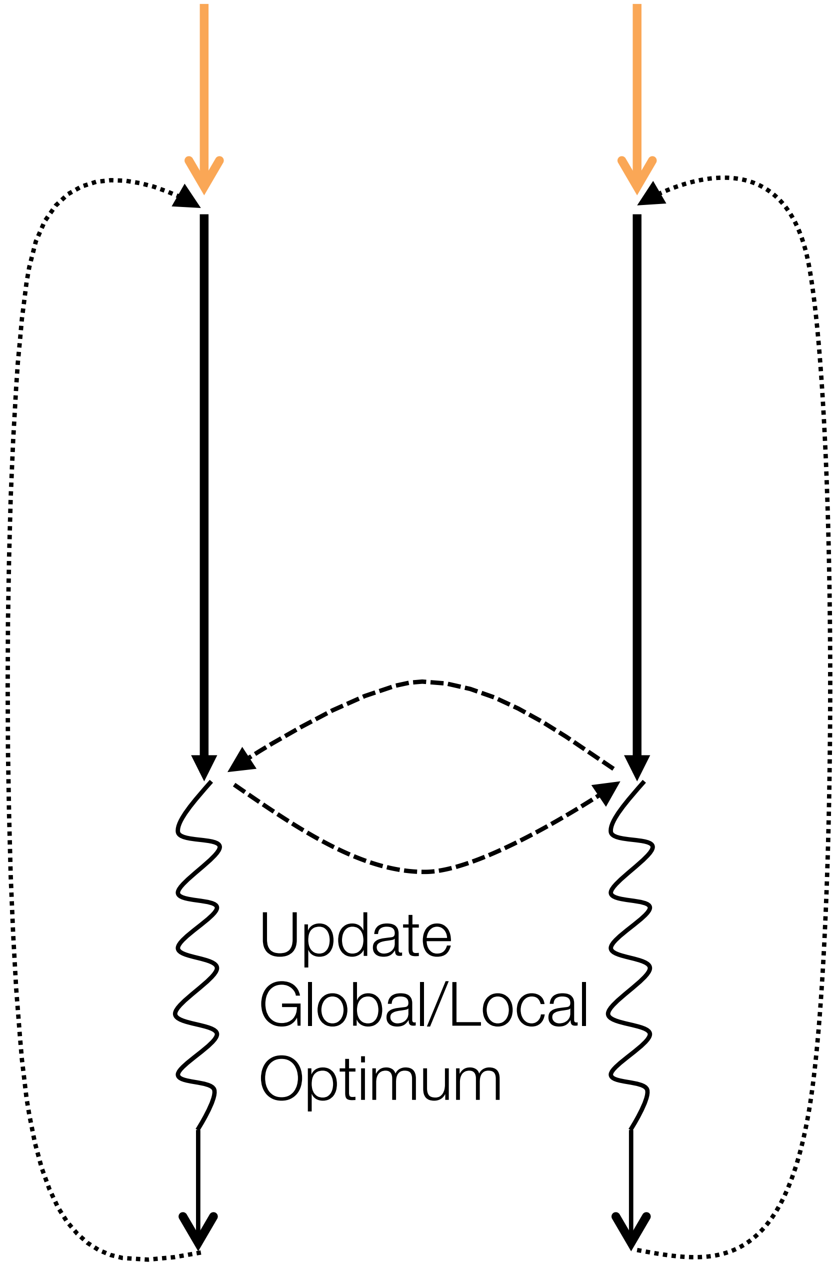
Local Optimization

Perturbation

Acceptance Test

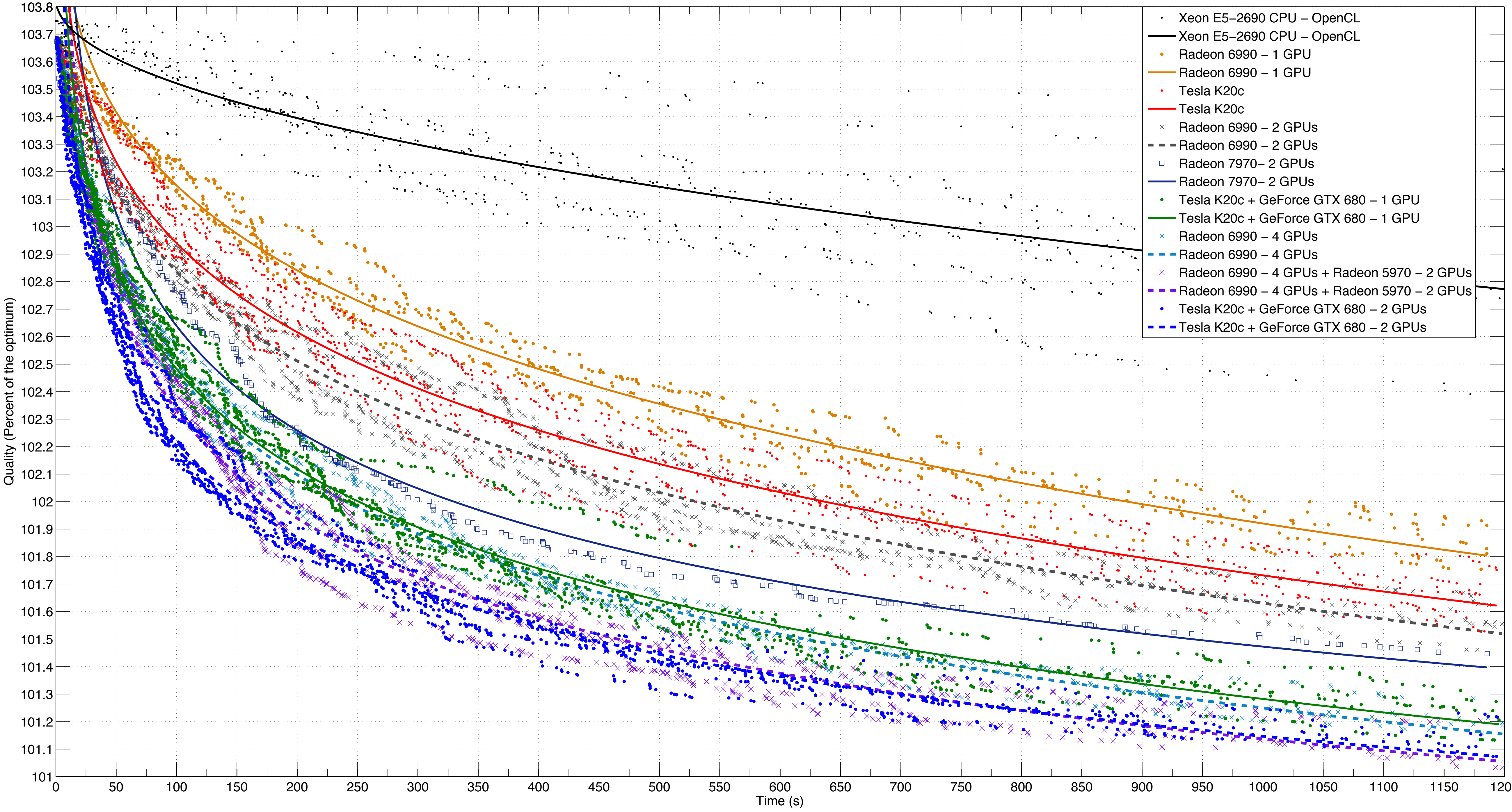
Inter-device cooperation





Shared memory (if within a node)
MPI (between the nodes)

Results


Quality (Percent of the optimum)
Time (s)
103.8
103.7
103.6
103.5
103.4
103.3
103.2
103.1
103
102.9
102.8
102.7
102.6
102.5
102.4
102.3
102.2
102.1
102
101.9
101.8
101.7
101.6
101.5
101.4
101.3
101.2
101.1
101
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
800
850
900
950
1000
1050
1100
1150
1200
Xeon E5−2690 CPU − OpenCL
Xeon E5−2690 CPU − OpenCL
Radeon 6990 − 1 GPU
Radeon 6990 − 1 GPU
Tesla K20c
Tesla K20c
Radeon 6990 − 2 GPUs
Radeon 6990 − 2 GPUs
Radeon 7970− 2 GPUs
Radeon 7970− 2 GPUs
Tesla K20c + GeForce GTX 680 − 1 GPU
Tesla K20c + GeForce GTX 680 − 1 GPU
Radeon 6990 − 4 GPUs
Radeon 6990 − 4 GPUs
Radeon 6990 − 4 GPUs + Radeon 5970 − 2 GPUs
Radeon 6990 − 4 GPUs + Radeon 5970 − 2 GPUs
Tesla K20c + GeForce GTX 680 − 2 GPUs
Tesla K20c + GeForce GTX 680 − 2 GPUs

LOGO - **LO**cal **G**PU **O**ptimization

http://olab.is.s.u-tokyo.ac.jp/~kamil.rocki/projects.html

Linux/Mac: Source code, Windows: Binary with GUI

GPL License

+ More resources about LOGO (slides, papers)

Summary

- Big TSP problems can be solved very quickly in parallel! (The fastest **OpenCL** TSP Solver?)
- The same OpenCL code can run efficiently on (almost) all tested devices
- **The implementation matters!**
- Sometimes ‘naive’ algorithms might run faster overall in parallel
- It seems that the CPU-based approach is more universal
    - **Prepare for future architectures**
- Avoid memory accesses (true for both CPUs and GPUs now)
    - Memory is a scarce resource, FLOPs are abundant

Backup

Xeon Phi, Max FLOP/s test

```
__kernel void sum_double16(__global double* const dA,  __global double* dResult)
{

    INIT

    double16 a = (double16)(dA[tx],....., dA[tx]);
    double16 b = (double16)(1.01,,....,1.16);

       MAIN_LOOP

    dResult[pIndex] = a.s0 +.... + b.sf;
}
```

```
#define FMAD128(a, b) \
        a = b * a + b; \
        b = a * b + a; \
        a = b * a + b; \
        b = a * b + a; \
        a = b * a + b; \
             .....
```

http://olab.is.s.u-tokyo.ac.jp/~kamil.rocki/FlopsCL.exe

http://olab.is.s.u-tokyo.ac.jp/~kamil.rocki/FlopsCL_src_linux.zip

Xeon Phi, Max FLOP/s test

```
CL_DEVICE_NAME = Intel(R) Many Integrated Core Acceleration Card
CL_DEVICE_VENDOR = Intel(R) Corporation
CL_DEVICE_VERSION = OpenCL 1.2
CL_DRIVER_VERSION = 1.2
CL_DEVICE_MAX_COMPUTE_UNITS = 236
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = 3
CL_DEVICE_MAX_WORK_ITEM_SIZES = 1024 / 1024 / 1024
CL_DEVICE_MAX_WORK_GROUP_SIZE = 1024
CL_DEVICE_MAX_CLOCK_FREQUENCY = 1052 MHz
CL_DEVICE_GLOBAL_MEM_SIZE = 5773 MB
CL_DEVICE_ERROR_CORRECTION_SUPPORT = YES
CL_DEVICE_LOCAL_MEM_SIZE = 32 kB
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE = 128 kB
        Compiling...
        Starting tests...
        [double  ] Time: 0.077155s, 890.67 GFLOP/s
        [double2 ] Time: 0.151210s, 908.93 GFLOP/s
        [double4 ] Time: 0.303471s, 905.78 GFLOP/s
        [double8 ] Time: 0.932834s, 589.34 GFLOP/s
        [double16] Time: 46.744131s, 23.52 GFLOP/s
```

i7-3960X, Max FLOP/s test

AMD

```
Device 1:
        CL_DEVICE_NAME = Intel(R) Core(TM) i7-3960X CPU @ 3.30GHz
        CL_DEVICE_VENDOR = GenuineIntel
        CL_DEVICE_VERSION = OpenCL 1.2 AMD-APP (1124.2)
        CL_DRIVER_VERSION = 1124.2 (sse2,avx)
        CL_DEVICE_MAX_COMPUTE_UNITS = 12
        CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = 3
        CL_DEVICE_MAX_WORK_ITEM_SIZES = 1024 / 1024 / 1024
        CL_DEVICE_MAX_WORK_GROUP_SIZE = 1024
        CL_DEVICE_MAX_CLOCK_FREQUENCY = 3301 MHz
        CL_DEVICE_GLOBAL_MEM_SIZE = 16058 MB
        CL_DEVICE_ERROR_CORRECTION_SUPPORT = NO
        CL_DEVICE_LOCAL_MEM_SIZE = 32 kB
        CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE = 64 kB
                Compiling...
                Starting tests...

                [float   ] Time: 0.395943s, 10.85 GFLOP/s
                [float2  ] Time: 0.396494s, 21.66 GFLOP/s
                [float4  ] Time: 0.404283s, 42.49 GFLOP/s
                [float8  ] Time: 0.406457s, 84.53 GFLOP/s
                [float16 ] Time: 0.416296s, 165.07 GFLOP/s
```

Intel

```
Device 0:
        CL_DEVICE_NAME =        Intel(R) Core(TM) i7-3960X CPU @ 3.30GHz
        CL_DEVICE_VENDOR = Intel(R) Corporation
        CL_DEVICE_VERSION = OpenCL 1.2 (Build 43113)
        CL_DRIVER_VERSION = 1.2
        CL_DEVICE_MAX_COMPUTE_UNITS = 12
        CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = 3
        CL_DEVICE_MAX_WORK_ITEM_SIZES = 1024 / 1024 / 1024
        CL_DEVICE_MAX_WORK_GROUP_SIZE = 1024
        CL_DEVICE_MAX_CLOCK_FREQUENCY = 3300 MHz
        CL_DEVICE_GLOBAL_MEM_SIZE = 16058 MB
        CL_DEVICE_ERROR_CORRECTION_SUPPORT = NO
        CL_DEVICE_LOCAL_MEM_SIZE = 32 kB
        CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE = 128 kB
                Compiling...
                Starting tests...

                [float   ] Time: 0.049610s, 86.57 GFLOP/s
                [float2  ] Time: 0.050083s, 171.52 GFLOP/s
                [float4  ] Time: 0.057464s, 298.97 GFLOP/s
                [float8  ] Time: 0.254403s, 135.06 GFLOP/s
                [float16 ] Time: 0.645904s, 106.39 GFLOP/s
```

?

Thank you!