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

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

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 26pt local minima for AT&T 22. Tour cost (vertical axis) is plotted against (a) mean distance to the other local minima and (b) distance to the global minimum.

Iterated Global Search - Global Optimization (Approximate algorithm)

1: procedure ITERATED LOCAL SEARCH
2: \( s_0 := \text{GenerateInitialSolution()} \)
3: \( s^* := \text{2optLocalSearch}(s_0) \)  Parallel
4: while (termination condition not met)
5: \( s' := \text{Perturbation}(s^*) \)
6: \( s^{*'} := \text{2optLocalSearch}(s') \)  Parallel
7: \( s^* := \text{AcceptanceCriterion}(s^*, s^{*'}) \)
8: end while
9: end procedure
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:

\[
\binom{n-2}{2} = (n-2) \times (n-3)/2 \text{ checks}
\]

• 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)

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

\[ n^2 \quad n \]
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

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

\[ n^2 \quad n \]
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

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

\[
\begin{align*}
\text{Memory needed} & = \frac{n^2}{24} + \frac{n}{26} \\
\end{align*}
\]

\[
\begin{align*}
\text{Memory needed} & = \frac{n^2}{24} + \frac{n}{26} \\
\end{align*}
\]
LUT (Look Up Table) vs recalculation

Time

'Not-so-clever' algorithm

Clever algorithm

Number of cores
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

\[ \text{dist}(i,j) = \text{dist}(j,i), \text{dist}(i,i) = 0 \]

<table>
<thead>
<tr>
<th></th>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
<th>0,4</th>
<th>0,5</th>
<th>0,6</th>
<th>0,7</th>
<th>0,8</th>
<th>0,9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matrix of city pairs to be checked for a possible swap, each pair corresponds to a unique distance value. The diagonal values are set to 0, indicating no change in distance for a city pair with itself.
GPU implementation

A simple function for EUC_2D problems

```c
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**

Each thread has to check:

\[
\text{if } (\text{distance}(i, i-1) + \text{distance}(j+1, j) > \text{distance}(i, j+1) + \text{distance}(i-1, j)) \text{ update best } i \text{ and } j;
\]
Parallelization

- GPU-centric approach

\[ dx = \text{coords}[i].x - \text{coords}[j].x; \]
\[ dy = \text{coords}[i].y - \text{coords}[j].y; \]
\[ \text{sqrtf}(dx \times dx + dy \times dy) + 0.5f; \]
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

```java
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);
        ....
    }
}
```
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
Parallelization

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

```c
#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

```c
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 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

         .......
    }

    __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 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

         .......
    }
```

OpenCL

13年5月14日火曜日
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)

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)

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

#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

<table>
<thead>
<tr>
<th>System</th>
<th>Native</th>
<th>CPU-centric - CUDA port</th>
<th>CPU-centric - Xeon Phi port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intel Xeon(R) CPU E5-2690</strong></td>
<td>1 core: 1.8 GFLOP/s</td>
<td>GPU-centric: 158.72 GFLOP/s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16 cores: 25.61 GFLOP/s</td>
<td></td>
<td>GPU-centric: 31.46 GFLOP/s</td>
</tr>
<tr>
<td></td>
<td>1 core SIMD (AVX): 11.60 GFLOP/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16 cores SIMD (AVX): 184.89 GFLOP/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OpenCL</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU-centric: 158.72 GFLOP/s</td>
<td>GPU-centric: 31.46 GFLOP/s</td>
<td></td>
</tr>
<tr>
<td><strong>Intel Xeon Phi (5110P):</strong></td>
<td>1 core: 0.7 GFLOP/s</td>
<td>CPU-centric: 7.27 GFLOP/s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>60 cores: 41.2 GFLOP/s</td>
<td>CPU-centric: clBuildProgram segfault</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 core SIMD: 8.1 GFLOP/s</td>
<td>GPU-centric: 442.68 GFLOP/s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>60 cores SIMD: 442.68 GFLOP/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AMD Radeon 7970 1GHz</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GPU-centric: 848.15 GFLOP/s</td>
<td>GPU-centric: 848.15 GFLOP/s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU-centric: clBuildProgram segfault</td>
<td>CPU-centric: clBuildProgram segfault</td>
<td></td>
</tr>
<tr>
<td><strong>NVIDIA GTX Titan</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CUDA: 1193.30 GFLOP/s</td>
<td>GPU-centric: 755.82 GFLOP/s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU-centric: 1137.49 GFLOP/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
- Radeon 7970

Global Optimization Module
- Xeon Phi

Global Optimization Module
- Main CPU

Global Optimization Module
- Tesla GPU

Result
Inter-device cooperation

Initial Solution

Local Optimization

Acceptance Test

Perturbation
Inter-device cooperation

Shared memory (if within a node)
MPI (between the nodes)
Results
LOGO - LOCAL GPU OPTIMIZATION

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
typedef struct double16 {
    double s0;
    double sf;
} double16;

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

__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;
}
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!