A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software

Fifth International Workshop on FPGA for Software Programmers
FSP 2018
August 31, 2018, Dublin, Ireland

Leonardo Solis-Vasquez    Andreas Koch

Technische Universität Darmstadt
Introduction

Current status of OpenCL for FPGAs

- Performance
  - In some cases: perf. with OpenCL is drastically lower than with HDLs
- Application complexity
  - Some: code fragments
  - Others: fairly regular computation & communication patterns

A more realistic case study:

- Molecular Docking (MD): AutoDock
- Several applications, e.g.: structure-based drug design
- Performs time-consuming calculations & search methods
Key aspects of molecular docking

MD aims to predict the best ways two molecules will interact

- **Representation**
  - Encoding of docking problem
  - E.g.: translation, rotation, torsion

- **Scoring function**
  - Energy of a particular pose
  - Lower binding energy is better

- **Search methods**
  - Finding an optimal pose
  - Which methods should be used?
AutoDock 4.2 (AD4)

- AutoDock\(^1\) is one of the most cited\(^2\) molecular docking tools

**Main features**

- **Binding encodings:** entities of a population

- **Offspring entities are generated through LGA:** Lamarckian Genetic Algorithm

- **LGA = Global Search (GS) + Local Search (LS)**
  - **GS:** entire population updated through a genetic algorithm: *crossover, mutation, selection*
  - **LS:** new entities from population subset (default 6%) are generated using small random arithmetic variations

- **Scoring of binding positions:** binding energy

\(^1\) [http://autodock.scripps.edu](http://autodock.scripps.edu)

\(^2\) Sousa et al. [2006]
AD4 scoring function (SF)

- Binding energy (Kcal mol\(^{-1}\)) from molecular mechanics
  - Interatomic distance: \(r_{ij}\), between atoms \(i\) and \(j\)
  - Molecular size (# atoms): receptor >1000, ligand <100

\[
SF = \sum_{i,j} \left[ W_{vdw} \left( \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^6} \right) + W_{hb} E(t) \left( \frac{C_{ij}}{r_{ij}^{12}} - \frac{D_{ij}}{r_{ij}^{10}} \right) + W_{el} \left( \frac{q_i q_j}{\epsilon (r_{ij}) r_{ij}} \right) + W_{ds} \left( S_i V_j + S_j V_i \right) e^{\frac{-r_{ij}^2}{2\sigma^2}} \right]
\]

- Energy components
  - Intramolecular: ligand ↔ ligand (computes SF)
  - Intermolecular: receptor ↔ ligand (replaced with grids)
AD4 algorithm

DOCKING JOB
- num-docking-runs: 50

LAMARCKIAN GENETIC ALGORITHM
- max-num-energy-evals: 2,500,000
- max-num.generations: 27,000

LOCAL SEARCH
- subset: 9 (6% population)
- max-num-iterations: 300

GLOBAL SEARCH
- population-size: 150

Step 1 (GS)
Create new generation

Step 2-3-4
Score entities

Step 1 (LS)
Modify degrees of freedom of entity

Step 2-3-4
Score entities

Step 2
Ligand atomic coordinates

Step 3
Intermolecular energy

Step 4
Intramolecular energy

Cluster analysis

Typical runtime distribution
- <1%
- <10%
- >90%

L. Solis-Vasquez, A. Koch
AutoDock on FPGAs using OpenCL
August 31, 2018 6 / 29
OpenCL programming models

OpenCL work-item: an individual processing thread

Data parallel (NDRange)

```c
kernel void
dp_mul(__global const float *a,
      __global const float *b,
      __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // Executed over "n" work-items
// Work-items are grouped
// into work-groups
```

Task parallel (single work-item)

```c
kernel void
tp_mul(__global const float *a,
        __global const float *b,
        __global float *c,
        int n)
{
    int i;

    // Loop is pipelined
    // Might require a #pragma
    for(i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```
Previous work on OpenCL AD4

- Latest OpenCL version for GPUs\(^3\) → data parallel
  - Docking runs are performed in parallel
  - Each entity is processed by a work-group
  - Fine-grained tasks are performed by work-items

- Speed-up of data-parallel OpenCL wrt. baseline
  - Baseline: single-threaded CPU (original AutoDock)
  - GPU: \(\sim 55.7\times\) max. speed-up 😊
  - FPGA: \(\sim 1000\times\) slower! 😞
  - GPU outperforms FPGA in all cases

- Could FPGAs benefit from an OpenCL task-parallel approach?

---

\(^3\) Solis-Vasquez et al. [2017]
Contributions

1. OpenCL task-parallel implementation of AutoDock

2. Design choices not extensively discussed previously
   - Multiple-producers-to-single consumer datapaths
   - Time-intensive loops with variable runtime

3. The first open-source OpenCL version of AutoDock for FPGAs
   - Performance improvements over single-threaded CPU
   - Maximum speed-up of $\sim 2.7x$ on an Arria-10 FPGA
Starting point

Single docking-run architecture

Existing pipeline architecture for AutoDock

Docking runs are executed sequentially
  ▶ A new docking run is started after the previous has finished

Each task is fine-grained pipelined

---

4 Pechan et al. [2010]
**OpenCL development**

Single docking-run architecture\(^4\)

- Each task coded as a single work-item kernel
- Kernels communicate via OpenCL pipes (= Intel channels)
- General design practices:
  - Pipelining every loop within each kernel
  - Minimizing loops initiation-interval (II)
- Design & optimization steps are organized in phases

---

\(^4\) Pechan et al. [2010]
First phase: development

- Closed loops (channels ↔ kernels) prevent channel-depth optimization
- Passing through global memory? → worked only in emulation
- Temporal populations stored on-chip: local memory
- Population creation & update in LGA → data dependencies
Encountered limitations

- *Emulator does not run interacting work-items in parallel* \(^5\)
  - Some concurrent exec. behaviors $\rightarrow$ inconsistent results
- Such concurrency limitation is critical
  - Heuristic-based search in AD4 can mask errors
- Operations might be rescheduled by the compiler
  - E.g.: rescheduling of channels $\rightarrow$ deadlocks

```c
// Rd & wr channel calls are BLOCKING
kernel void producer(__global const uint *src,...) {
    ...
    write_channel_intel(c0, src[0]);
    mem_fence(CLK_CHANNEL_MEM_FENCE);
    write_channel_intel(c1, src[1]);
    ...
} // Enforce order with fences
```

```c
// Rd & wr channel calls are BLOCKING
kernel void consumer(__global const uint *dst,...) {
    ...
    dst[0] = read_channel_intel(c0);
    mem_fence(CLK_CHANNEL_MEM_FENCE);
    dst[1] = read_channel_intel(c1);
    ...
} // Enforce order with fences
```

\(^5\)Intel FPGA SDK for OpenCL [2017]
First phase: runtime impact

- In all experiments:
  - Default LGA config.\(^6\)
  - 100 docking runs

- Used 5 PDB\(^7\) compounds

<table>
<thead>
<tr>
<th>Ligand-Receptor</th>
<th>PDB ID</th>
<th>3ptb</th>
<th>1stp</th>
<th>4hmg</th>
<th>3ce3</th>
<th>3c1x</th>
</tr>
</thead>
<tbody>
<tr>
<td># Atoms</td>
<td></td>
<td>13</td>
<td>18</td>
<td>27</td>
<td>37</td>
<td>46</td>
</tr>
<tr>
<td># Torsions</td>
<td></td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Lower performance 😞

---

\(^6\) AutoDockTools [2012]

\(^7\) Protein Data Bank archive: [https://www.rcsb.org/](https://www.rcsb.org/)
First phase: what to optimize next?

- **LGA data-dependencies**
  - Population creation & update
  - Random numbers in GS and LS

- **Bottleneck: energy calc.**
  - Coords → InterE → IntraE
Second phase: development

- **LS and RNGs become separate kernels**
- **Decoupling → increase concurrency**

**Coords:** data dependency in array of rotated atoms $\Rightarrow II = 36$

**InterE & IntraE:** $II = 1$
Second phase: runtime impact

i5-6600K CPU vs. Arria-10 GX 1150 FPGA

Although bottleneck was optimized, performance became even lower 😞
Second phase: what is next?

- **LS**: code refactoring for pipelining
- **LS-entities are independent** → exploit LS concurrency!

- LS: code refactoring for pipelining
- LS-entities are independent → exploit LS concurrency!
Third phase: development

- Inner LS loops: most were pipelined
- Outmost LS loop: carried dependency on genotype data → not pipelined
- LS replication → multiple-producers to single-consumer
- Variable # energy evals → ready signals + Arbiter
Allocation of constant data

- OpenCL memory-space qualifiers for constant data:
  - `__constant`: on-chip cache, default: 16 KB
  - `__global const`: off-chip, maximum: 16 GB

- If `__constant` args require larger spaces than cache size:
  - Perf. penalties: `__constant` > `__global const`
  - Reason: `__global const` accesses are implemented with extra circuitry → can tolerate longer latencies

- Appropriate allocation in Coords, InterE, IntraE kernels
  - Small arrays: 12 KB → `__constant`
  - Large look-up tables: >270 KB → `__global const`
    - E.g.: lists of rotations & intramolecular contributors
Arithmetic representation

- Arria-10 FPGAs have hardened floating-point DSP units
  - Our design started all in floating-point

- For bottleneck: floating-point or fixed-point?

- Coords kernel
  - Keeps track of rotated atoms → loop-carried dependency
  - Floating-point → pipelined with II=36
  - Fixed-point (16.16) → pipelined with II=10 😊

- InterE & IntraE kernels
  - Floating-point → pipelined with II=1
  - Fixed-point (32.32) → pipelined with II=1
  -Floating-point results in faster designs 😊
## Third phase: runtime impact

<table>
<thead>
<tr>
<th>Development phase</th>
<th># LS replicas</th>
<th>Arithmetic representation</th>
<th>LS</th>
<th>Coords</th>
<th>InterE</th>
<th>IntraE</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>1</td>
<td>float</td>
<td>float</td>
<td>float</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td>1</td>
<td>float</td>
<td>float</td>
<td>float</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Third</td>
<td>3</td>
<td>fixed</td>
<td>fixed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speed-up of 1.5x with 3ptb! 😊

### i5-6600K CPU vs. Arria-10 GX 1150 FPGA

<table>
<thead>
<tr>
<th></th>
<th>Serial-CPU</th>
<th>First-phase</th>
<th>Second-phase</th>
<th>Third-phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (s)</td>
<td>5.86</td>
<td>2.903</td>
<td>2.550</td>
<td>3.76</td>
</tr>
<tr>
<td>3ptb</td>
<td>1.416</td>
<td>5.784</td>
<td>6.678</td>
<td>1.013</td>
</tr>
<tr>
<td>1stp</td>
<td>1.867</td>
<td>6.636</td>
<td>8.121</td>
<td>1.364</td>
</tr>
<tr>
<td>4hmg</td>
<td>2.841</td>
<td>8.519</td>
<td>9.247</td>
<td>1.790</td>
</tr>
<tr>
<td>3ce3</td>
<td>836</td>
<td>12.573</td>
<td>14,502</td>
<td></td>
</tr>
<tr>
<td>3c1x</td>
<td>1,487</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 3ptb
- 1stp
- 4hmg
- 3ce3
- 3c1x
Fourth phase: development

- LS: further replicated
- # LS replicas: based on # LS-entities
- Arbiter: receives only ready signals
- Coords: mux selects genotype
Fourth phase: runtime impact

<table>
<thead>
<tr>
<th>Develop. phase</th>
<th># LS replicas</th>
<th>Arithmetic representation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LS</td>
</tr>
<tr>
<td>First</td>
<td>1</td>
<td>float</td>
</tr>
<tr>
<td>Second</td>
<td>1</td>
<td>float</td>
</tr>
<tr>
<td>Third</td>
<td>3</td>
<td>fixed</td>
</tr>
<tr>
<td>Fourth</td>
<td>5</td>
<td>fixed</td>
</tr>
</tbody>
</table>

Speed-up of 1.8x with 3ptb! 😊

i5-6600K CPU vs. Arria-10 GX 1150 FPGA

Runtime (s)
Best performance results

<table>
<thead>
<tr>
<th>Develop. phase</th>
<th># LS replicas</th>
<th>Arithmetic representation</th>
<th>LS</th>
<th>Coords</th>
<th>InterE</th>
<th>IntraE</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>1</td>
<td></td>
<td></td>
<td>float</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td>1</td>
<td></td>
<td></td>
<td>float</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Third</td>
<td>3</td>
<td></td>
<td></td>
<td>fixed</td>
<td>fixed</td>
<td></td>
</tr>
<tr>
<td>Fourth-A</td>
<td>5</td>
<td></td>
<td></td>
<td>fixed</td>
<td>fixed</td>
<td>float</td>
</tr>
<tr>
<td>Fourth-B</td>
<td>9</td>
<td></td>
<td></td>
<td>fixed</td>
<td>fixed</td>
<td></td>
</tr>
<tr>
<td>Fourth-C</td>
<td>9</td>
<td></td>
<td></td>
<td>float</td>
<td>fixed</td>
<td></td>
</tr>
<tr>
<td>Fourth-D</td>
<td>9</td>
<td></td>
<td></td>
<td>float</td>
<td>float</td>
<td></td>
</tr>
</tbody>
</table>

Speed-up of fastest design (Fourth-B)

i5-6600K CPU vs. Arria-10 GX 1150 FPGA
Resource utilization & frequency

| Develop. phase | # LS replicas | Arithmetic representation
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LS</td>
<td>Coords</td>
</tr>
<tr>
<td>First</td>
<td>1</td>
<td>float</td>
</tr>
<tr>
<td>Second</td>
<td>1</td>
<td>float</td>
</tr>
<tr>
<td>Third</td>
<td>3</td>
<td>fixed</td>
</tr>
<tr>
<td>Fourth-A</td>
<td>5</td>
<td>fixed</td>
</tr>
<tr>
<td>Fourth-B</td>
<td>9</td>
<td>fixed</td>
</tr>
<tr>
<td>Fourth-C</td>
<td>9</td>
<td>float</td>
</tr>
<tr>
<td>Fourth-D</td>
<td>9</td>
<td>float</td>
</tr>
</tbody>
</table>

Utilization percentage (%) in Arria-10 GX 1150 FPGA

Frequency (MHz)

0 100 200

First 174.4 187.5 172.6 187.5 185.7 185.7
Second 215.2 174.4 187.5 172.6 187.5 185.7 185.7
Third 202.2 174.4 187.5 172.6 187.5 185.7 185.7
Fourth-A 202.2 174.4 187.5 172.6 187.5 185.7 185.7
Fourth-B 202.2 174.4 187.5 172.6 187.5 185.7 185.7
Fourth-C 202.2 174.4 187.5 172.6 187.5 185.7 185.7
Fourth-D 202.2 174.4 187.5 172.6 187.5 185.7 185.7

L. Solis-Vasquez, A. Koch
AutoDock on FPGAs using OpenCL
August 31, 2018 26 / 29
Challenges for higher performance

- Required **awareness** of the underlying hardware
  - For ensuring correctness: explicit synchronization between channel calls using fences (already discussed)
  - For improving II: re-arrangement the local memory layout

```c
__local int __attribute__((numbanks(4), bankwidth(4)))
lmem[8][4];
#pragma unroll
for (int i = 0; i<4; i+=2)
{
    lmem[x][i] = ...;
}
```
Recipes for higher performance

- AD4 has termination criteria known only at runtime
  - Loop unrolling was possible only in few cases
  - Loop pipelining was much more exploited

- Achieving II=1 is not always possible
  - E.g. : Coords: II = 36
  - Split large sections (LGA) into smaller instances (LS, RNG)
  - Replicate slower data-producer kernels (LS)

- Appropriate allocation of constant data

- Arithmetic representation
Concluding remarks

- Performance of previous work: OpenCL data-parallel AutoDock
  - GPUs: \( \sim 55x \) speed-up
  - FPGAs: three orders of magnitude slowdown

- This work: design & optimization methodology
  - Task-based parallelization
  - Detailed development phases
  - FPGA-specific techniques using OpenCL
  - Exploration of different architectural choices

- Overall result of our experiments
  - Maximum of \( \sim 2.7x \) speed-up on FPGA
A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software

https://git.esa.informatik.tu-darmstadt.de/docking/ocladock-fpga

Leonardo Solis-Vasquez

solis@esa.tu-darmstadt.de

https://www.esa.cs.tu-darmstadt.de
Supplementary slides
LGA method - main calls

```plaintext
lamarckian_genetic_algorithm {
    while lga-stop-condition is false {
        genetic_generation (population); // global
        for entity in random-subset (population) // local
            local_search (get_genotype (entity));
        update (population);
    }
}
```
Solis-Wets local-search method

```c
local_search (genotype) {
    while ls-stop-condition is false {
        delta = create_delta (step);
        newgenotype1 = add_on_every_gene (genotype, delta);

        if (energy (newgenotype1) < energy (genotype))
            genotype = newgenotype1;
            success++; fail = 0;
        else
            newgenotype2 = sub_on_every_gene (genotype, delta);

            if (energy (newgenotype2) < energy (genotype))
                genotype = newgenotype2;
                success++; fail = 0;
            else
                success = 0; fail++;

        step = update_step (success, fail);
    }
}
```
Functional validation

Comparison of energy & size of best cluster

<table>
<thead>
<tr>
<th>PDB ID</th>
<th># Atoms</th>
<th># Torsions</th>
<th>Energy of best pose (Kcal mol$^{-1}$)</th>
<th>Size of best cluster (100 docking runs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Serial baseline</td>
<td>OpenCL FPGA</td>
</tr>
<tr>
<td>3ptb</td>
<td>13</td>
<td>2</td>
<td>$-5.55$</td>
<td>$-5.53$</td>
</tr>
<tr>
<td>1stp</td>
<td>18</td>
<td>5</td>
<td>$-8.37$</td>
<td>$-7.76$</td>
</tr>
<tr>
<td>4hmg</td>
<td>27</td>
<td>10</td>
<td>$-3.68$</td>
<td>$-4.11$</td>
</tr>
<tr>
<td>3ce3</td>
<td>37</td>
<td>5</td>
<td>$-11.59$</td>
<td>$-10.88$</td>
</tr>
<tr>
<td>3c1x</td>
<td>46</td>
<td>8</td>
<td>$-13.61$</td>
<td>$-12.61$</td>
</tr>
</tbody>
</table>

- Different selection schemes → discrepancies in the cluster size
  - Serial baseline: *proportional selection*
  - OpenCL FPGA: *binary tournament*

- Binary tournament chosen for better performance leads to...
  - more diverse populations
  - less dense clusters
## Resource utilization

### FPGA resource utilization & frequency

<table>
<thead>
<tr>
<th>Design config.</th>
<th>ALMs</th>
<th>RAMs</th>
<th>DSPs</th>
<th>Freq. (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Total: 427 200</strong></td>
<td><strong>Total: 2 713</strong></td>
<td><strong>Total: 1 518</strong></td>
<td></td>
</tr>
<tr>
<td>DC1</td>
<td>129 301 (30%)</td>
<td>1 075 (40%)</td>
<td>388 (26%)</td>
<td>215.2</td>
</tr>
<tr>
<td>DC2</td>
<td>128 018 (30%)</td>
<td>999 (37%)</td>
<td>262 (17%)</td>
<td>174.4</td>
</tr>
<tr>
<td>DC3</td>
<td>158 586 (37%)</td>
<td>1 799 (66%)</td>
<td>548 (36%)</td>
<td>187.5</td>
</tr>
<tr>
<td>DC4a</td>
<td>177 509 (42%)</td>
<td>1 826 (67%)</td>
<td>586 (39%)</td>
<td>172.6</td>
</tr>
<tr>
<td>DC4b</td>
<td>222 372 (52%)</td>
<td>1 880 (69%)</td>
<td>662 (44%)</td>
<td>187.5</td>
</tr>
<tr>
<td>DC4c</td>
<td>220 427 (52%)</td>
<td>1 898 (70%)</td>
<td>659 (43%)</td>
<td>185.7</td>
</tr>
<tr>
<td>DC4d</td>
<td>219 359 (51%)</td>
<td>1 944 (72%)</td>
<td>383 (25%)</td>
<td>185.7</td>
</tr>
</tbody>
</table>

- Arria-10 GX 1150 FPGA, Gidel Proc10A card, 16 GB RAM
- Latest compiler supported by the BSP: Intel FPGA SDK for OpenCL v16.0
**Performance results**

### Execution runtime & best speed-up for 100 docking runs

#### Execution runtime (seconds)

<table>
<thead>
<tr>
<th>Design config.</th>
<th>Ligand-Receptor PDB ID</th>
<th>3ptb</th>
<th>1stp</th>
<th>4hmg</th>
<th>3ce3</th>
<th>3c1x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial CPU</td>
<td>3ptb</td>
<td>586</td>
<td>836</td>
<td>1416</td>
<td>1867</td>
<td>2841</td>
</tr>
<tr>
<td></td>
<td>1stp</td>
<td>836</td>
<td>1416</td>
<td>2841</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4hmg</td>
<td>1416</td>
<td></td>
<td>2841</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3ce3</td>
<td>1867</td>
<td>2841</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3c1x</td>
<td>2841</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC1</td>
<td>3ptb</td>
<td>2903</td>
<td>5784</td>
<td>6636</td>
<td>8519</td>
<td>12573</td>
</tr>
<tr>
<td></td>
<td>1stp</td>
<td>5784</td>
<td>6636</td>
<td>8519</td>
<td>12573</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4hmg</td>
<td>6636</td>
<td>8519</td>
<td>12573</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3ce3</td>
<td>8519</td>
<td>12573</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3c1x</td>
<td>12573</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC2</td>
<td>3ptb</td>
<td>2550</td>
<td>6678</td>
<td>8121</td>
<td>9247</td>
<td>14502</td>
</tr>
<tr>
<td></td>
<td>1stp</td>
<td>6678</td>
<td>8121</td>
<td>9247</td>
<td>14502</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4hmg</td>
<td>8121</td>
<td>9247</td>
<td>14502</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3ce3</td>
<td>9247</td>
<td>14502</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3c1x</td>
<td>14502</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC3</td>
<td>3ptb</td>
<td>376</td>
<td>739</td>
<td>1013</td>
<td>1364</td>
<td>1790</td>
</tr>
<tr>
<td></td>
<td>1stp</td>
<td>739</td>
<td>1013</td>
<td>1364</td>
<td>1790</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4hmg</td>
<td>1013</td>
<td>1364</td>
<td>1790</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3ce3</td>
<td>1364</td>
<td>1790</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3c1x</td>
<td>1790</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC4a</td>
<td>3ptb</td>
<td>315</td>
<td>563</td>
<td>788</td>
<td>1096</td>
<td>1496</td>
</tr>
<tr>
<td></td>
<td>1stp</td>
<td>563</td>
<td>788</td>
<td>1096</td>
<td>1496</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4hmg</td>
<td>788</td>
<td>1096</td>
<td>1496</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3ce3</td>
<td>1096</td>
<td>1496</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3c1x</td>
<td>1496</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC4b</td>
<td>3ptb</td>
<td>211</td>
<td>385</td>
<td>623</td>
<td>1077</td>
<td>1487</td>
</tr>
<tr>
<td></td>
<td>1stp</td>
<td>385</td>
<td>623</td>
<td>1077</td>
<td>1487</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4hmg</td>
<td>623</td>
<td>1077</td>
<td>1487</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3ce3</td>
<td>1077</td>
<td>1487</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3c1x</td>
<td>1487</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC4c</td>
<td>3ptb</td>
<td>215</td>
<td>388</td>
<td>634</td>
<td>1079</td>
<td>1491</td>
</tr>
<tr>
<td></td>
<td>1stp</td>
<td>388</td>
<td>634</td>
<td>1079</td>
<td>1491</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4hmg</td>
<td>634</td>
<td>1079</td>
<td>1491</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3ce3</td>
<td>1079</td>
<td>1491</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3c1x</td>
<td>1491</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC4d</td>
<td>3ptb</td>
<td>332</td>
<td>706</td>
<td>933</td>
<td>1250</td>
<td>1759</td>
</tr>
<tr>
<td></td>
<td>1stp</td>
<td>706</td>
<td>933</td>
<td>1250</td>
<td>1759</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4hmg</td>
<td>933</td>
<td>1250</td>
<td>1759</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3ce3</td>
<td>1250</td>
<td>1759</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3c1x</td>
<td>1759</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Best speed-up

- DC4b: 2.77x, 2.17x, 2.27x, 1.73x, 1.91x

---

- i5-6600K single CPU core, at 3.5 GHz, 16 GB RAM
- Arria-10 GX 1150 FPGA, Gidel Proc10A card, 16 GB RAM
Energy-efficiency results

Energy consumption & best energy-efficiency gains for 100 docking runs

<table>
<thead>
<tr>
<th>Ligand-Receptor PDB ID</th>
<th>Design config.</th>
<th>3ptb</th>
<th>1stp</th>
<th>4hmg</th>
<th>3ce3</th>
<th>3c1x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial CPU</td>
<td>11.80</td>
<td>16.69</td>
<td>28.07</td>
<td>36.27</td>
<td>54.85</td>
<td></td>
</tr>
<tr>
<td>DC4b</td>
<td>6.33</td>
<td>11.50</td>
<td>18.69</td>
<td>32.31</td>
<td>44.61</td>
<td></td>
</tr>
</tbody>
</table>

Best energy improvement

| Ligand-Receptor PDB ID | DC4b | 1.86x | 1.45x | 1.50x | 1.12x | 1.23x |

- i5-6600K single CPU core
  - Sampling interval of 50 ms, using power perf. counters
  - Power samples were integrated over time to derive energy

- Arria-10 GX 1150 FPGA
  - Power values from fully placed & routed designs: *quartus.pow*
  - Estimation of ~ 30 W was multiplied by the respective runtime
Comparison with a HDL design

- AutoDock accelerator designed in Verilog\(^8\)
  - Similar pipelining architecture
  - No LS-kernel replication
  - Runtimes of a single docking-run were reported

For smaller PDB molecules: runtimes are similar
- OpenCL: 3ptb: \(~2.1\) s, 1stp: \(~3.8\) s
- Verilog: average of 60 PDBs: \(~3.1\) s

For larger PDB molecules:
- OpenCL runtimes become longer than the \(~3.1\) s average
- Verilog does not appear to suffer from this
  - (Most likely) due to finer manually-controlled pipelining

\(^8\)Pechan et al. [2010]
Tool support

- Design specification & emulation-based verification
  - As easy as with GPUs
  - Optimization reports: to assess impact of code modifications
    - Initiation interval (II)
    - Estimated resource-utilization

- Hardware validation & subsequent optimization cycles
  - Much more involved
  - Hardware profiler: to pinpoint bottlenecks like:
    - Unbalanced communication traffic between producer (GS, LS) & consumer (Arbiter) kernels
    - Inefficient memory accesses
Challenges for higher performance

- Required *awareness* of the underlying hardware
  - For improving II: code-refactoring using hardware constructs

```c
// Unoptimized accumulation
// Array a[] has N elements
double temp_sum = 0;

// II = 11
for (int i = 0; i < N; ++i) {
    temp_sum += arr[i];
}

*result = temp_sum;
```

```c
// Optimized accumulation using shift register

// Create shift reg.
#define II_CYCLES 12
double shift_reg[II_CYCLES+1];

// Initialize all reg. elements to 0
...

// II = 1
// Load a[i] into end of shift reg.
for (int i = 0; i < N; ++i) {
    // if N > II_CYCLE,
    // add to shift_reg[0] to preserve values
    shift_reg[II_CYCLES] = shift_reg[0] + arr[i];

    #pragma unroll
    // Shift every element of shift reg.
    for (int j = 0; j < II_CYCLES; ++j) {
        shift_reg[j] = shift_reg[j + 1];
    }
}
...
```
Fences for local-memory within single work-item kernels

- Standard says they guard access to data qualified with `__local`
- Actually they protect the access to data stored in block RAM
- Either OpenCL private or `__local` arrays can be mapped to block RAM
  - As long as their size $\geq 64$ bytes
- Such fences used in Coords kernel
  - For computing correctly atomic positions (loop-carried dependency)
Productivity

- Smallest design: first phase $\rightarrow$ 4 kernels
  - LGA, Coords, InterE, & IntraE

- Correctly emulated design for the first phase: $\sim$ 4 weeks

- Largest designs: fourth phase \{B, C, D\} $\rightarrow$ 27 kernels each
  - LGA, Coords, InterE, IntraE
  - Arbiter, four RNG-GS$^9$, nine LS, & nine RNG-LS

- Completion of whole development: $\sim$ 5 months
  - Delays caused by emulator limitations
  - Large FPGA synthesis and mapping times
    - $\sim$ 8 hours for each of our largest designs

$^9$ Four different genetic ops: crossover, mutation, GS & LS selection