Performance Evaluation and Improvements of the PoCL Open-Source OpenCL Implementation on Intel CPUs

Tobias Baumann, Matthias Noack, Thomas Steinke

Supercomputing Department
Zuse Institute Berlin

9th International Workshop on OpenCL and SYCL, 26-29 April 2021
Motivation

- Top500 status quo: 91.8% of all systems feature Intel CPUs
Motivation

- Top500 status quo: 91.8% of all systems feature Intel CPUs

- Top500 trends: increased diversity
  - accelerators and co-processors (27.2%)
  - ARM CPUs (1%, including #1)
  - x86 CPUs from AMD (4.4%)

⇒ portability more important than ever

Top 500 Nov. 2020
New Systems by Arch

Motivation

- Top500 status quo: 91.8% of all systems feature Intel CPUs

- Top500 trends: increased diversity
  - accelerators and co-processors (27.2%)
  - ARM CPUs (1%, including #1)
  - x86 CPUs from AMD (4.4%)

⇒ portability more important than ever

- OpenCL widely supported, but not every vendor’s 1st choice
  ⇒ vendor-independent open source impl.: PoCL

Portable Computing Language (PoCL)

- portable, open source (MIT) OpenCL implementation
  - maintained by Customized Parallel Computing group at Tampere University, Finland

Portable Computing Language (PoCL)

- portable, open source (MIT) OpenCL implementation
  - maintained by Customized Parallel Computing group at Tampere University, Finland
- aims to support a variety of compute devices in a single platform
- longer term goal: enhance performance portability of OpenCL programs

Portable Computing Language (PoCL)

- portable, open source (MIT) OpenCL implementation
  - maintained by Customized Parallel Computing group at Tampere University, Finland
- aims to support a variety of compute devices in a single platform
- longer term goal: enhance performance portability of OpenCL programs

- currently supported devices:
  - various CPUs
  - NVIDIA GPUs via libcuda
  - HSA-supported GPUs
  - TCE ASIPs (experimental)
  - multiple (private) adaptations in active production use

Portable Computing Language (PoCL)

- portable, open source (MIT) OpenCL implementation
  - maintained by Customized Parallel Computing group at Tampere University, Finland
- aims to support a variety of compute devices in a single platform
- longer term goal: enhance performance portability of OpenCL programs

- currently supported devices:
  - various CPUs
  - NVIDIA GPUs via libcuda
  - HSA-supported GPUs
  - TCE ASIPs (experimental)
  - multiple (private) adaptations in active production use

- compiler: Clang/LLVM
- vectorised math libraries: SLEEF, libclc

Contributions

1. Performance evaluation PoCL vs. Intel OpenCL
Contributions

1. Performance evaluation PoCL vs. Intel OpenCL

2. PoCL bug fixes:
   - WG size algorithm (one-off)
   - scheduler (integer division)
Contributions

1. Performance evaluation PoCL vs. Intel OpenCL

2. PoCL bug fixes:
   - WG size algorithm (one-off)
   - scheduler (integer division)

3. New PoCL CPU driver using Intel Threading Building Blocks (TBB)
Contributions

1. Performance evaluation PoCL vs. Intel OpenCL

2. PoCL bug fixes:
   - WG size algorithm (one-off)
   - scheduler (integer division)

3. New PoCL CPU driver using Intel Threading Building Blocks (TBB)

4. Evaluation of vectorisation strategies and LLVM’s loop and VPlan vectorisers
PoCL Work-Group Size Algorithm (simplified, for CPUs)

- Problem: How to determine WG size if none is specified?

1. **Step 1:** Maximize WG size
   - Maximum WG size: 4096 WI
   - Preferred WG multiple (for vectorization): 8 WI

2. **Step 2:** Reduce WG size
   - Reason: at least one WG/CU
   - Minimum WG size: 32 WI
   - Can lead to distributions like 1.6 WGs/CU

- Parallelization threshold: Number of CUs × 32 WI
- Omit whole algorithm: Explicitly specify WG size

**Improvement #1:** Minor off-by-one issue in step 2
PoCL Work-Group Size Algorithm (simplified, for CPUs)

- Problem: How to determine WG size if none is specified?
- Step 1: **maximise** WG size
  - maximum WG size: 4096 WI
  - preferred WG multiple (for vectorisation): 8 WI
- Step 2: **reduce** WG size
  - reason: at least one WG/CU
  - minimum WG size: 32 WI
  - parallelisation threshold: \( \# \text{CU} \times 32 \text{ WI} \)
- Omit whole algorithm: explicitly specify WG size

**Improvement #1:** minor off-by-one issue in step 2
Problem: How to determine WG size if none is specified?

Step 1: maximise WG size
- maximum WG size: 4096 WI
- preferred WG multiple (for vectorisation): 8 WI

Step 2: reduce WG size
- reason: at least one WG/CU
- minimum WG size: 32 WI
- can lead to distributions like 1.6 WGs/CU
PoCL Work-Group Size Algorithm (simplified, for CPUs)

- **Problem:** How to determine WG size if none is specified?

- **Step 1:** **maximise** WG size
  - maximum WG size: 4096 WI
  - preferred WG multiple (for vectorisation): 8 WI

- **Step 2:** **reduce** WG size
  - reason: at least one WG/CU
  - minimum WG size: 32 WI
  - can lead to distributions like 1.6 WGs/CU

- **parallelisation threshold:** $\#CU \times 32$ WI
PoCL Work-Group Size Algorithm (simplified, for CPUs)

- **Problem**: How to determine WG size if none is specified?
- **Step 1**: *maximise* WG size
  - maximum WG size: 4096 WI
  - preferred WG multiple (for vectorisation): 8 WI
- **Step 2**: *reduce* WG size
  - reason: at least one WG/CU
  - minimum WG size: 32 WI
  - can lead to distributions like 1.6 WGs/CU
- **parallelisation threshold**: $\#CU \times 32$ WI
- **omit whole algorithm**: explicitly specify WG size
PoCL Work-Group Size Algorithm (simplified, for CPUs)

- Problem: How to determine WG size if none is specified?
- Step 1: **maximise** WG size
  - maximum WG size: 4096 WI
  - preferred WG multiple (for vectorisation): 8 WI
- Step 2: **reduce** WG size
  - reason: at least one WG/CU
  - minimum WG size: 32 WI
  - can lead to distributions like 1.6 WGs/CU
- parallelisation threshold: $\#CU \times 32$ WI
- omit whole algorithm: explicitly specify WG size

Improvement #1: minor off-by-one issue in step 2
PoCL Scheduling Algorithm (*pthread* device)

- one (software) thread per CU (hardware thread)
- problem: schedule WGs to threads

- example: schedule 64 WGs to 4 threads
  - balanced workload
  - no scheduling overhead
PoCL Scheduling Algorithm (*pthread* device)

- one (software) thread per CU (hardware thread)
- problem: schedule WGs to threads
- pull scheduler
- threads pass through the scheduler *sequentially*
- algorithm: number of remaining WGs \(\div\) #threads
PoCL Scheduling Algorithm (*pthread* device)

- one (software) thread per CU (hardware thread)
- problem: schedule WGs to threads
- pull scheduler
- threads pass through the scheduler *sequentially*
- algorithm: number of remaining WGs ÷ #threads

- example: schedule 64 WGs to 4 threads
  - balanced workload
  - no scheduling overhead
PoCL Scheduling Algorithm Balanced Workload

WG remaining: 64  WG scheduled: 0  WG to schedule: 16

$t_1$:

$t_2$:

$t_3$:

$t_4$:

kernel runtime
PoCL Scheduling Algorithm Balanced Workload

WGs remaining: 48  WGs scheduled: 16  WGs to schedule: 12

$t_1:$

$t_2:$

$t_3:$

$t_4:$

kernel runtime
PoCL Scheduling Algorithm Balanced Workload

WGs remaining: 36  
WGs scheduled: 28  
WGs to schedule: 9

$t_1$:  
$t_2$:  
$t_3$:  
$t_4$:  

kernel runtime

Improvement #2: integer division bug
PoCL Scheduling Algorithm Balanced Workload

WG remaining: 27  
WG scheduled: 37  
WG to schedule: 7

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{pocl_diagram}
\caption{PoCL scheduling algorithm for balanced workload.}
\end{figure}
PoCL Scheduling Algorithm Balanced Workload

WG remaining: 20  WG scheduled: 44  WG to schedule: 5

\[ t_1 : \]
\[ t_2 : \]
\[ t_3 : \]
\[ t_4 : \]

kernel runtime
PoCL Scheduling Algorithm Balanced Workload

WGs remaining: 15  WGs scheduled: 49  WGs to schedule: 4

$t_1$:  
$t_2$:  
$t_3$:  
$t_4$:  

kernel runtime

Improvement #2: integer division bug
PoCL Scheduling Algorithm Balanced Workload

WG remaining: 0  WG scheduled: 64  WG to schedule: -

\[ t_1: \]
\[ t_2: \]
\[ t_3: \]
\[ t_4: \]

kernel runtime
## PoCL Scheduling Algorithm Balanced Workload

<table>
<thead>
<tr>
<th>WGs remaining:</th>
<th>WGs scheduled:</th>
<th>WGs to schedule:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Improvement #2: integer division bug
Scheduling Algorithms: OpenMP "static" vs. PoCL "default"

**static**

- chunk_size
- work group
- thread

- threads
- kernel runtime

worst case (t=16)

best case (t=10)

default

work group (twice the time)
Scheduling Algorithms: OpenMP "static" vs. PoCL "default"

**static**
- chunk_size
- work group
- thread

**default**
- threads

kernel runtime

worst case (t=16)
best case (t=10)
default
default
work group (twice the time)
Scheduling Algorithms: OpenMP "static" vs. PoCL "default"

- **Static**
  - chunk_size
  - work group
  - thread
  - kernel runtime

- **Default**
  - work group
  - work group (twice the time)
  - kernel runtime

**Best case (t=10) - Worst case (t=16)**

Tobias Baumann

PoCL Performance Evaluation and Improvements 9th IWOCL, 26-29 April 2021
Scheduling Algorithms: OpenMP "static" vs. PoCL "default"

**static**
- chunk_size
- work group
- thread

**default**
- work group
- work group (twice the time)

**kernel runtime**
- threads

Worst case (t=16)

Best case (t=10)
TBB device for PoCL

- Intel Threading Building Blocks (TBB) library
  - higher level alternative to pthread
  - used by Intel OpenCL
  - open source (Apache 2.0)
  - available partitioners for scheduling: auto, affinity, static, simple
  - optional parameter for all partitioners: grain size
  - additional work stealing strategy
TBB device for PoCL

- Intel Threading Building Blocks (TBB) library
  - higher level alternative to \textit{pthread}
  - used by Intel OpenCL
  - open source (Apache 2.0)
  - available \textbf{partitioners} for scheduling: auto, affinity, static, simple
  - optional parameter for all partitioners: grain size
  - additional work stealing strategy

- Integration into PoCL:
  - derived from the \textit{pthread} device
  - replaced relevant code sections with calls to TBB library
  - custom env vars: \texttt{POCL\_TBB\_PARTITIONER}, \texttt{POCL\_TBB\_GRAIN\_SIZE}
Setup

- **Software:**
  - Intel CPU Runtime for OpenCL Applications 2021.1
  - PoCL 1.4 LLVM 9
  - PoCL 1.6 LLVM 11
  - PoCL 1.6 LLVM 11 with proposed TBB device

- **Hardware:** dual socket Intel Xeon Gold 6138 (Skylake)
  - 20 cores per socket $\Rightarrow$ 80 hardware threads
  - AVX-512 units: 2 per core $\Rightarrow$ 1 per hardware thread
Synthetic Micro Benchmark: op

- synthetic, low-level, micro benchmark
- goal: measure the performance of operators and OpenCL built-in functions

```c
void op_kernel (DATA_T* in_a, DATA_T* in_b, DATA_T* out) {
    int id = get_global_id(0);
    DATA_T tmp_a = in_a[id];
    DATA_T tmp_b = in_b[id];
    for (int i = 0; i < ITERATIONS; ++i) {
        tmp_a = OP_TO_BENCHMARK (tmp_a, tmp_b); // data dep.
    }
    out[id] = tmp_a;
}
```

- compile-time specialisation via: DATA_T, ITERATIONS, and OP_TO_BENCHMARK
Automatic vs. Manual Vectorisation

- **Automatically** by the compiler or **manually** by using vector data types
- \( \#WIs = \#data\_\text{elements} \div \text{vector\_length} \)
- e.g. resulting work-items (WIs) spanning 16 data elements:

<table>
<thead>
<tr>
<th>vec. mode</th>
<th>vec. length</th>
<th>#WIs</th>
<th>logical data layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic</td>
<td>1</td>
<td>16</td>
<td><img src="image" alt="Logical data layout for automatic vectorisation" /></td>
</tr>
<tr>
<td>manual4</td>
<td>4</td>
<td>4</td>
<td><img src="image" alt="Logical data layout for manual vectorisation (4)" /></td>
</tr>
<tr>
<td>manual8</td>
<td>8</td>
<td>2</td>
<td><img src="image" alt="Logical data layout for manual vectorisation (8)" /></td>
</tr>
</tbody>
</table>
Synthetic Micro Benchmark: op

average kernel runtime [ms]

- nop
- fma
- sqrt
- sin

PoCL Performance Evaluation and Improvements 9th IWOCL, 26-29 April 2021
Initial PoCL/Intel slowdown:
- fma: 4.59×
- sqrt: 1.95×
- sin: 8.39×
Synthetic Micro Benchmark: op

PoCL 1.4 $\Rightarrow$ PoCL 1.6 speedup:
- fma: $1.3 \times$
- sqrt: $1.4 \times$
- sin: $1.66 \times$
Synthetic Micro Benchmark: op

TBB device speedup:
\(\text{fma: } 1.58 \times, \text{sqrt: } 1.2 \times, \text{sin: } 1.04 \times\)
Total speedup:
fma: 2.06×, sqrt: 1.68×, sin: 1.74×
Proxy Application Benchmark: hexciton

- proxy application benchmark for the **DM-HEOM** code
- same kernel in different variants and stages of optimisation
- vectorisation strategies: **automatic** and **manual**
- OpenMP version using KART library for runtime compilation
- instruction mix: streamlined FMA instructions
- fixed number of $512 \times 1024$ small $7 \times 7$ matrices

Proxy Application Benchmark: hexciton

Intel/PoCL 1.6 gap:
0.51× (implicit WG size)
0.72× (explicit WG size)
Proxy Application Benchmark: hexciton

PoCL 1.6 explicit WG size speedup: 1.40×
Proxy Application Benchmark: hexciton

Even with disabled vectorisation, Intel outperforms PoCL.
Implicit WG sizes: 25.6 WGs/CU (Intel) vs. 1.6 WGs/CU)
Proxy Application Benchmark: hexciton


average kernel runtime [ms]

- automatic vec.
- implicit WG size
- automatic vec.
- explicit WG size
- manual vec.
- implicit WG size
- manual vec.
- explicit WG size

Intel vectoriser enabled
Intel vectoriser disabled
PoCL 1.4 LLVM 9
PoCL 1.6 LLVM 11
PoCL TBB LLVM 11 auto partitioner
PoCL TBB LLVM 11 affinity partitioner

Tobias Baumann
PoCL Performance Evaluation and Improvements
9th IWOCL, 26-29 April 2021
Proxy Application Benchmark: hexciton

![Graph showing average kernel runtime [ms] for different configurations and compiler settings.]

**Speedup using TBB device: 1.06×**

- Intel vectoriser enabled
- Intel vectoriser disabled
- PoCL 1.4 LLVM 9
- PoCL 1.6 LLVM 11
- PoCL TBB LLVM 11 auto partitioner
- PoCL TBB LLVM 11 affinity partitioner

Tobias Baumann

PoCL Performance Evaluation and Improvements 9th IWOCL, 26-29 April 2021 14/20
Proxy Application Benchmark: hexciton

Specifying affinity partitioner: 1.26×

average kernel runtime [ms]

- Intel vectoriser enabled
- Intel vectoriser disabled
- PoCL 1.4 LLVM 9
- PoCL 1.6 LLVM 11
- PoCL TBB LLVM 11 auto partitioner
- PoCL TBB LLVM 11 affinity partitioner

Tobias Baumann

PoCL Performance Evaluation and Improvements

9th IWOCL, 26-29 April 2021
Proxy Application Benchmark: hexciton

Overall speedup compared to Intel: 1.31×
Real World Application Benchmark: raytracing

- derived from a raytracing code published on GitHub
  - ⇒ reflect average real world application code
- more complex data structures
- work-items can take different code paths
- contains trigonometric and various other built-in functions

Real World Application Benchmark: raytracing

![Bar chart showing average kernel runtime in milliseconds for different configurations: Intel, PoCL 1.4 LLVM 9, PoCL 1.6 LLVM 9, PoCL 1.6 LLVM 11, PoCL TBB LLVM 11. The x-axis represents the resolution 2160p, and the y-axis represents the average kernel runtime in milliseconds.]
Real World Application Benchmark: raytracing

Initial PoCL/Intel slowdown: $1.62 \times$

Tobias Baumann

PoCL Performance Evaluation and Improvements

9th IWOCL, 26-29 April 2021
Real World Application Benchmark: raytracing

PoCL 1.4 ⇒ PoCL 1.6 slowdown: 1.02×
Real World Application Benchmark: raytracing

LLVM 9 $\Rightarrow$ LLVM 11 speedup: 1.25\times
Real World Application Benchmark: raytracing

average kernel runtime [ms]

Intel
PoCL 1.4 LLVM 9
PoCL 1.6 LLVM 9
PoCL 1.6 LLVM 11
PoCL TBB LLVM 11

TBB device speedup: 1.09×
Real World Application Benchmark: raytracing

Slowdown compared to Intel: $1.32 \times \Rightarrow 1.21 \times$
Vectorisation in PoCL

manual vectorisation  automatic vectorisation

```c
void op_kernel ( double8 * in_a ,
                double8 * in_b ,
                double8 * out )
{
    // ...
    for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b ;
    }
    // ...
}

vfmadd231
pd
%zmm1,%zmm2,%zmm3
⇒ trivial
⇒ 8 doubles per OpenCL work-item

void op_kernel ( double * in_a ,
                double * in_b ,
                double * out )
{
    // ...
    for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b ;
    }
    // ...
}

vfmadd231
sd
%xmm0,%xmm1,%xmm0
⇒ no vectorisation
⇒ inner loop not vectorisable (reduction)
```
Vectorisation in PoCL

**manual vectorisation**

```c
void op_kernel (double8* in_a ,
                double8* in_b ,
                double8* out )
{
    // ...
    for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b ;
    }
    // ...
}
```

**automatic vectorisation**

```c
void op_kernel (double* in_a ,
                double* in_b ,
                double* out )
{
    // ...
    for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b ;
    }
    // ...
}
```
Vectorisation in PoCL

**manual vectorisation**

```c
void op_kernel ( double8* in_a,
                 double8* in_b,
                 double8* out )
{
    // ... for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b;
    }
    // ...}
}
```

\[ \text{vfmadd231pd} \ %zmm1,\%zmm2,\%zmm3 \]

⇒ **trivial** for the compiler
⇒ 8 doubles per OpenCL work-item

**automatic vectorisation**
Vectorisation in PoCL

**manual vectorisation**

```c
void op_kernel ( double8* in_a ,
    double8* in_b ,
    double8* out )
{
    // ...
    for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b ;
    }
    // ...
}
```

**automatic vectorisation**

```c
void op_kernel ( double* in_a ,
    double* in_b ,
    double* out )
{
    // ...
    for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b ;
    }
    // ...
}
```

\[ \text{vfmad231pd} \ %\text{zmm1,}\%\text{zmm2,}\%\text{zmm3} \]

⇒ **trivial** for the compiler
⇒ 8 doubles per OpenCL work-item
Vectorisation in PoCL

**manual vectorisation**

```c
void op_kernel (double8* in_a,
               double8* in_b,
               double8* out)
{
    // ...
    for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b;
    }
    // ...
}
```

`vfmadd231pd %zmm1,%zmm2,%zmm3`

⇒ **trivial** for the compiler

⇒ 8 doubles per OpenCL work-item

**automatic vectorisation**

```c
void op_kernel (double* in_a,
                double* in_b,
                double* out)
{
    // ...
    for (/* ITERATIONS */)
    {
        tmp_a += tmp_a * tmp_b;
    }
    // ...
}
```

`vfmadd231sd %xmm0,%xmm1,%xmm0`

⇒ **no vectorisation** across work-items

⇒ inner loop not vectorisable (reduction)
Goal: Automatic Work-Item Vectorisation in PoCL

• SIMD vectorisation across work-items on CPUs ≈ SIMT execution on GPUs ⇒ requires outer-loop vectorisation outside kernel function ⇒ implemented in Intel OpenCL, but not in PoCL

• Feasibility-evaluation through simplified OpenMP-proxy of OpenCL runtime:

/* automatic vectorisation scheme */
#pragma omp parallel for // threading across WGs
for ( int group_id = 0; group_id < ( num / VEC_LENGTH ); ++ group_id )
{
  /* vectorised work - item loop */
#pragma omp simd // vectorisation across WIs inside WG
  for ( int local_id = 0; local_id < VEC_LENGTH ; ++ local_id )
  {
    // scalar typed kernel code , e.g. hexciton_benchmark
  }
}
Goal: Automatic Work-Item Vectorisation in PoCL

- **SIMD** vectorisation across work-items on CPUs ≈ **SIMT** execution on GPUs
  - requires **outer-loop vectorisation** outside kernel function
  - implemented in Intel OpenCL, but **not in PoCL**
Goal: Automatic Work-Item Vectorisation in PoCL

- **SIMD** vectorisation across work-items on CPUs $\approx$ **SIMT** execution on GPUs
  - requires *outer-loop vectorisation* outside kernel function
  - implemented in Intel OpenCL, but **not in PoCL**

- **Feasibility-evaluation** through simplified OpenMP-proxy of OpenCL runtime:
**Goal: Automatic Work-Item Vectorisation in PoCL**

- **SIMD** vectorisation across work-items on CPUs $\approx$ **SIMT** execution on GPUs
  - requires **outer-loop vectorisation** outside kernel function
  - implemented in Intel OpenCL, but **not in PoCL**

- **Feasibility-evaluation** through simplified OpenMP-proxy of OpenCL runtime:

```c
/* automatic vectorisation scheme */
#pragma omp parallel for // threading across WGs
for (int group_id = 0; group_id < (num / VEC_LENGTH); ++group_id)
{
    /* vectorised work-item loop */
    #pragma omp simd // vectorisation across WIs inside WG
    for (int local_id = 0; local_id < VEC_LENGTH; ++local_id)
    {
        // scalar typed kernel code, e.g. hexciton_benchmark
    }
}
```
Goal: Automatic Work-Item Vectorisation in PoCL

- **SIMD** vectorisation across work-items on CPUs ≈ **SIMT** execution on GPUs
  - requires **outer-loop vectorisation** outside kernel function
  - implemented in Intel OpenCL, but **not in PoCL**
- **Feasibility-evaluation** through simplified OpenMP-proxy of OpenCL runtime:

```c
/* automatic vectorisation scheme */
#pragma omp parallel for // threading across WGs
for (int group_id = 0; group_id < (num / VEC_LENGTH); ++group_id)
{
    /* vectorised work-item loop */
    #pragma omp simd // vectorisation across WIs inside WG
    for (int local_id = 0; local_id < VEC_LENGTH; ++local_id)
    {
        // scalar typed kernel code, e.g. hexciton_benchmark
    }
}
```

Can LLVM 11 do this?
hexciton\_benchmark: LLVM Vectoriser Comparison

![Bar chart showing performance comparison between two LLVM vectorisers.](image)

- **Automatic vec. temporaries**
- **Automatic vec. global buffer**
- **Automatic vec. inner SIMD loop**
- **Manual vec. temporaries**
- **Manual vec. global buffer**

The chart compares the average kernel runtime [ms] for two LLVM vectorisers: LLVM 11 loop vectoriser and LLVM 11 VPlan vectoriser.
hexciton_benchmark: LLVM Vectoriser Comparison

Details in the code can make a large difference.

The graph shows the average kernel runtime [ms] for different vectorisation techniques using LLVM 11 loop vectoriser and LLVM 11 VPlan vectoriser. The techniques include automatic vec. temporaries, automatic vec. global buffer, automatic vec. inner SIMD loop, manual vec. temporaries, and manual vec. global buffer.
hexciton_benchmark: LLVM Vectoriser Comparison

- **loop vectoriser**: does not actually vectorise
- **VPlan vectoriser**: performs intended vectorisation

---

Tobias Baumann

*PoCL Performance Evaluation and Improvements* 9th IWOCL, 26-29 April 2021
VPlan: successfully outer loop vectorisation, but
⇒ fails to recognise memory access pattern (AoSoA)
⇒ expensive gather/scatter instructions
hexciton_benchmark: LLVM Vectoriser Comparison

VPlan speedup with temporaries: $1.37 \times$ ⇒ still $1.91 \times$ slower than manual

- automatic vec. temporaries
- automatic vec. global buffer
- automatic vec. inner SIMD loop
- manual vec. temporaries
- manual vec. global buffer

average kernel runtime [ms]

LLVM 11 loop vectoriser
LLVM 11 VPlan vectoriser
hexciton_benchmark: LLVM Vectoriser Comparison

loop permutation: SIMD loop was moved inside
⇒ inner loop vectorisation
⇒ both vectorisers succeed
Tobias Baumann

PoCL Performance Evaluation and Improvements

9th IWOCL, 26-29 April 2021

19/20

hexciton_benchmark: LLVM Vectoriser Comparison

manual: slight slowdown with VPlan vectoriser

average kernel runtime [ms]

LLVM 11 loop vectoriser
LLVM 11 VPlan vectoriser

automatic vec. temporaries
automatic vec. global buffer
automatic vec. inner SIMD loop
manual vec. temporaries
manual vec. global buffer
Summary

- Performance Evaluation PoCL vs. Intel OpenCL on Intel CPUs
  - comparable performance only with manual vectorisation

https://doi.org/10.1145/3456669.3456698

Funded by:
German Federal Ministry of Education and Research (BMBF), Research Campus MODAL, project no. 05M20ZBM
Summary

- Performance Evaluation PoCL vs. Intel OpenCL on Intel CPUs
  - comparable performance only with manual vectorisation
- TBB-based device for PoCL
  - speedups over Intel possible (achieved: 1.3×)
- Two bugs fixed in PoCL


https://doi.org/10.1145/3456669.3456698

Funded by:
German Federal Ministry of Education and Research (BMBF), Research Campus MODAL, project no. 05M20ZBM
Summary

- Performance Evaluation PoCL vs. Intel OpenCL on Intel CPUs
  - comparable performance only with manual vectorisation
- TBB-based device for PoCL
  - speedups over Intel possible (achieved: 1.3×)
- Two bugs fixed in PoCL
- LLVM vectoriser evaluation
  - outer loop vectorisation across work-items in PoCL depends on LLVM’s VPlan progress


Funded by:
German Federal Ministry of Education and Research (BMBF), Research Campus MODAL, project no. 05M20ZBM
Summary

- Performance Evaluation PoCL vs. Intel OpenCL on Intel CPUs
  - comparable performance only with manual vectorisation
- TBB-based device for PoCL
  - speedups over Intel possible (achieved: $1.3 \times$)
- Two bugs fixed in PoCL
- LLVM vectoriser evaluation
  - outer loop vectorisation across work-items in PoCL depends on LLVM’s VPlan progress

Thank you for your attention!

tobias.baumann@zib.de


https://doi.org/10.1145/3456669.3456698

Funded by:
German Federal Ministry of Education and Research (BMBF), Research Campus MODAL, project no. 05M20ZBM