Performance Evolution of Different SYCL Implementations based on the Parallel Least Squares Support Vector Machine Library

Marcel Breyer, University of Stuttgart
Alexander Van Craen, Dirk Pflüger
Motivation

Runtime in seconds on an NVIDIA 100 GPU for different SYCL implementations:
- nd_range AOT
- nd_range
- hierarchical AOT
- hierarchical

AOT = Ahead-Of-Time

16,384 x 4096
What to know about PLSSVM
Support Vector Machines (SVMs) and their problems

- SVMs as supervised machine learning technique
- originally meant for binary classification
Support Vector Machines (SVMs) and their problems

- SVMs as supervised machine learning technique
- originally meant for binary classification
- SVMs have to solve a convex quadratic problem
  - state-of-the-art: Sequential Minimal Optimization (SMO) (proposed by Platt in 1998)
  - inherently sequential algorithm
Support Vector Machines (SVMs) and their problems

- SVMs as supervised machine learning technique
- originally meant for binary classification
- SVMs have to solve a convex quadratic problem
  → state-of-the-art: Sequential Minimal Optimization (SMO) (proposed by Platt in 1998)
  → inherently sequential algorithm
- many SVM implementations modify SMO to exploit some parallelism
  → still not well suited for modern, highly parallel hardware
Support Vector Machines (SVMs) and their problems

- SVMs as supervised machine learning technique
- originally meant for binary classification
- SVMs have to solve a convex quadratic problem
  → state-of-the-art: Sequential Minimal Optimization (SMO) (proposed by Platt in 1998)
  → inherently sequential algorithm
- many SVM implementations modify SMO to exploit some parallelism
  → still not well suited for modern, highly parallel hardware

→ Least Squares Support Vector Machine (LS-SVM)
  (proposed by Suykens and Vandewalle in 1999)
Support Vector Machines (SVMs) and their problems

- SVMs as supervised machine learning technique
- originally meant for binary classification
- SVMs have to solve a convex quadratic problem
  - state-of-the-art: Sequential Minimal Optimization (SMO) (proposed by Platt in 1998)
  - inherently sequential algorithm
- many SVM implementations modify SMO to exploit some parallelism
  - still not well suited for modern, highly parallel hardware

→ Least Squares Support Vector Machine (LS-SVM)
  (proposed by Suykens and Vandewalle in 1999)

- reformulation of standard SVM to solving a system of linear equations
- massively parallel algorithms known
We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

\[
\begin{bmatrix}
Q & \vec{1}_n \\
\vec{1}_n^T & 0
\end{bmatrix} \cdot \begin{bmatrix}
\alpha \\
b
\end{bmatrix} = \begin{bmatrix}
y \\
0
\end{bmatrix}
\]

where \( Q \) is the kernel matrix according to

\[
Q_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \quad \text{(with } \delta_{ij} = \begin{cases} 
1 & i = j \\
0 & \text{else}
\end{cases} \text{)}
\]
We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

\[
\begin{bmatrix}
Q & \vec{1}_n \\
\vec{1}_n^T & 0
\end{bmatrix}
\begin{bmatrix}
\alpha \\
b
\end{bmatrix} =
\begin{bmatrix}
y \\
0
\end{bmatrix}
\]

where \( Q \) is the kernel matrix according to

\[
Q_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \quad \left( \text{with} \quad \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)
\]

\( Q \) is symmetric positive-definite
We parallelized the most complex operations in the CG algorithm. LS-SVMs solve the system of linear equations:

\[
\begin{bmatrix}
Q & \vec{1}_n^T \\
\vec{1}_n & 0
\end{bmatrix}
\begin{bmatrix}
\alpha \\
b
\end{bmatrix} =
\begin{bmatrix}
y \\
0
\end{bmatrix}
\]

where \( Q \) is the kernel matrix according to

\[
Q_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \quad \text{(with } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases})
\]

\( \Rightarrow \) \( Q \) is symmetric positive-definite

\( \Rightarrow \) Conjugate Gradient algorithm: (variant of Shewchuk et al.)

1: \( i \leftarrow 0 \)
2: \( r \leftarrow b - Ax \)
3: \( d \leftarrow r \)
4: \( \delta_{new} \leftarrow r^T r \)
5: \( \delta_0 \leftarrow \delta_{new} \)
6: \textbf{while} \( i < i_{\text{max}} \) and \( \delta_{new} > \epsilon^2 \delta_0 \) \textbf{do}
7: \( q \leftarrow Ad \)
8: \( \alpha \leftarrow \frac{\delta_{new}}{\delta_{old}} \)
9: \( x \leftarrow x + \alpha d \)
10: \textbf{if} \( i \) is divisible by 50 \textbf{then}
11: \( r \leftarrow b - Ax \)
12: \textbf{else}
13: \( r \leftarrow r - \alpha q \)
14: \textbf{end if}
15: \( \delta_{old} \leftarrow \delta_{new} \)
16: \( \delta_{new} \leftarrow r^T r \)
17: \( \beta \leftarrow \frac{\delta_{new}}{\delta_{old}} \)
18: \( d \leftarrow r + \beta d \)
19: \( i \leftarrow i + 1 \)
20: \textbf{end while}
We parallelized the most complex operations in the CG algorithm.

LS-SVMs solve the system of linear equations:

\[
\begin{bmatrix}
Q & \vec{1}_n \\
\vec{1}_n^T & 0
\end{bmatrix} \cdot \begin{bmatrix}
\alpha \\
b
\end{bmatrix} = \begin{bmatrix}
y \\
0
\end{bmatrix}
\]

where \( Q \) is the kernel matrix according to

\[Q_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \quad \text{(with } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases})\]

\( Q \) is symmetric positive-definite

\( \to \) Conjugate Gradient algorithm: (variant of Shewchuk et al.)

- Setup or constant operations \( \to \) host

1: \( i \leftarrow 0 \)
2: \( r \leftarrow b - Ax \)
3: \( l \leftarrow 0 \)
4: \( \delta_{new} \leftarrow r^T r \)
5: \( \delta_0 \leftarrow \delta_{new} \)
6: \( \text{while } i < i_{max} \text{ and } \delta_{new} > \epsilon^2 \delta_0 \text{ do} \)
7: \( q \leftarrow Ad \)
8: \( \alpha \leftarrow \frac{\delta_{new}}{d^T q} \)
9: \( x \leftarrow x + \alpha d \)
10: \( \text{if } i \text{ is divisible by 50 then} \)
11: \( r \leftarrow b - Ax \)
12: \( \text{else} \)
13: \( r \leftarrow r - \alpha q \)
14: \( \text{end if} \)
15: \( \delta_{old} \leftarrow \delta_{new} \)
16: \( \delta_{new} \leftarrow r^T r \)
17: \( \beta \leftarrow \frac{\delta_{new}}{\delta_{old}} \)
18: \( d \leftarrow r + \beta d \)
19: \( i \leftarrow i + 1 \)
20: \( \text{end while} \)
We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

\[
\begin{bmatrix}
Q & \vec{1}_n \\
\vec{1}_n^T & 0
\end{bmatrix}
\begin{bmatrix}
\alpha \\
b
\end{bmatrix}
=
\begin{bmatrix}
y \\
0
\end{bmatrix}
\]

where \( Q \) is the kernel matrix according to

\[Q_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \quad \text{with} \quad \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases}\]

\( \rightarrow \) \( Q \) is symmetric positive-definite

\( \rightarrow \) Conjugate Gradient algorithm: (variant of Shewchuk et al.)

- Setup or constant operations \( \rightarrow \) host
- BLAS Level 1 \( \rightarrow \) host

1: \( i \leftarrow 0 \)
2: \( r \leftarrow b - Ax \)
3: \( \delta_0 \leftarrow \delta_{new} \)
4: \( \delta_{new} \leftarrow r^T r \)
5: \( \delta_{new} \leftarrow \delta_{new} / \delta_0 \)
6: \( \textbf{while } i < i_{max} \textbf{ and } \delta_{new} > \epsilon^2 \delta_0 \textbf{ do} \)
7: \( q \leftarrow Ad \)
8: \( \alpha \leftarrow \frac{\delta_{new}}{d^T q} \)
9: \( x \leftarrow x + \alpha d \)
10: \( \textbf{if } i \textbf{ is divisible by } 50 \textbf{ then} \)
11: \( r \leftarrow b - Ax \)
12: \( \textbf{else} \)
13: \( r \leftarrow r - \alpha q \)
14: \( \textbf{end if} \)
15: \( \delta_{old} \leftarrow \delta_{new} \)
16: \( \delta_{new} \leftarrow r^T r \)
17: \( \beta \leftarrow \frac{\delta_{new}}{\delta_{old}} \)
18: \( d \leftarrow r + \beta d \)
19: \( i \leftarrow i + 1 \)
20: \( \textbf{end while} \)
We parallelized the most complex operations in the CG algorithm.

LS-SVMs solve the system of linear equations:
\[
\begin{bmatrix}
Q & \vec{1}_n \\
\vec{1}_n^T & 0
\end{bmatrix}
\begin{bmatrix}
\alpha \\
b
\end{bmatrix}
= 
\begin{bmatrix}
y \\
0
\end{bmatrix}
\]

where \( Q \) is the kernel matrix according to
\[
Q_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij}
\]
(with \( \delta_{ij} = \begin{cases} 
1 & i = j \\
0 & \text{else}
\end{cases} \))

\( Q \) is symmetric positive-definite.

\( \rightarrow \) Conjugate Gradient algorithm: (variant of Shewchuk et al.)

- Setup or constant operations \( \rightarrow \) host
- BLAS Level 1 \( \rightarrow \) host
- BLAS Level 2 \( \rightarrow \) device

1: \( i \leftarrow 0 \)
2: \( r \leftarrow b - Ax \)
3: \( t \leftarrow r \)
4: \( \delta_{new} \leftarrow r^T r \)
5: \( \delta_0 \leftarrow \delta_{new} \)
6: while \( i < i_{\text{max}} \) and \( \delta_{new} > \epsilon^2 \delta_0 \) do
   7: \( q \leftarrow Ad \)
   8: \( \alpha \leftarrow \frac{\delta_{new}}{\delta_0^2} \)
   9: \( x \leftarrow x + \alpha d \)
10: if \( i \) is divisible by 50 then
   11: \( r \leftarrow b - Ax \)
12: else
   13: \( r \leftarrow r - \alpha q \)
14: end if
15: \( \delta_{old} \leftarrow \delta_{new} \)
16: \( \delta_{new} \leftarrow r^T r \)
17: \( \beta \leftarrow \frac{\delta_{new}}{\delta_{old}} \)
18: \( d \leftarrow r + \beta d \)
19: \( i \leftarrow i + 1 \)
20: end while
We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

\[
\begin{bmatrix}
Q & \vec{1}_n \\
\vec{1}_n^T & 0
\end{bmatrix} \cdot \begin{bmatrix}
\alpha \\
b
\end{bmatrix} = \begin{bmatrix}
y \\
0
\end{bmatrix}
\]

where \( Q \) is the kernel matrix according to

\[ Q_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \]  

\[ \text{with } \delta_{ij} = \begin{cases} 
1 & i = j \\
0 & \text{else}
\end{cases} \]

\( \rightarrow \) \( Q \) is **symmetric positive-definite**

\( \rightarrow \) **Conjugate Gradient algorithm:** (variant of Shewchuk et al.)

- Setup or constant operations \( \rightarrow \) host
- BLAS Level 1 \( \rightarrow \) host
- BLAS Level 2 \( \rightarrow \) device

\( \rightarrow \) \( Q \in \mathbb{R}^{\text{num\_data\_points} \times \text{num\_data\_points}} \)
We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

\[
\begin{bmatrix}
Q & \vec{r}_n^T \\
\vec{r}_n & 0
\end{bmatrix}
\cdot
\begin{bmatrix}
\alpha \\
b
\end{bmatrix}
=
\begin{bmatrix}
y \\
0
\end{bmatrix}
\]

where \( Q \) is the kernel matrix according to

\[Q_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \quad \text{(with } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases}\text{)}\]

\( Q \) is symmetric positive-definite

\( Q \) is \underline{symmetric} \, \underline{positive-definite}

Conjugate Gradient algorithm: \( \text{ (variant of Shewchuk et al.)} \)

- Setup or constant operations \( \rightarrow \) host
- BLAS Level 1 \( \rightarrow \) host
- BLAS Level 2 \( \rightarrow \) device

\( Q \in \mathbb{R}^{\text{num\_data\_points} \times \text{num\_data\_points}} \)

\( Q \) \underline{implicitly} calculate \( Q \) in each iteration

1: \( i \leftarrow 0 \)
2: \( r \leftarrow b - A x \)
3: \( d \leftarrow r \)
4: \( \delta_{\text{new}} \leftarrow r^T r \)
5: \( \delta_0 \leftarrow \delta_{\text{new}} \)
6: \textbf{while} \( i < i_{\text{max}} \) \text{ and } \delta_{\text{new}} > \epsilon^2 \delta_0 \textbf{ do}
7: \quad y \leftarrow A d
8: \quad \alpha \leftarrow \frac{\delta_{\text{new}}}{d^T \alpha} \)
9: \quad x \leftarrow x + \alpha d
10: \quad \textbf{if} \ i \text{ is divisible by } 50 \textbf{ then}
11: \quad \textbf{else}
12: \quad r \leftarrow r - \alpha q
13: \quad \delta_{\text{old}} \leftarrow \delta_{\text{new}} \)
14: \quad \beta \leftarrow \frac{\delta_{\text{new}}}{\delta_{\text{old}}} \)
15: \quad d \leftarrow r + \beta d
16: \quad i \leftarrow i + 1
17: \textbf{end while}
PLSSVM - Parallel Least Squares Support Vector Machine

- modern C++17
- open source & on GitHub
- single and double precision via template parameter
- parallelizes implicit matrix-vector multiplication in CG

https://github.com/SC-SGS/PLSSVM

Marcel Breyer, University of Stuttgart, IPVS - SC : Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
PLSSVM - Parallel Least Squares Support Vector Machine

- modern C++17
- open source & on GitHub
- single and double precision via template parameter
- parallelizes implicit matrix-vector multiplication in CG
- backends: OpenMP, CUDA, HIP, OpenCL, and SYCL
- backend and target platform selectable at runtime

https://github.com/SC-SGS/PLSSVM

Marcel Breyer, University of Stuttgart, IPVS - SC : Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
PLSSVM - Parallel Least Squares Support Vector Machine

- modern C++17
- open source & on GitHub
- single and double precision via template parameter
- parallelizes implicit matrix-vector multiplication in CG
- backends: OpenMP, CUDA, HIP, OpenCL, and SYCL
- backend and target platform selectable at runtime
- multi-GPU support for linear kernel function
- drop-in replacement for LIBSVM's svm-train, svm-predict, and svm-scale executables
- currently only binary classification and dense calculations

https://github.com/SC-SGS/PLSSVM
New results and findings
**NVIDIA A100**

![Graph showing runtime in s vs. # data points (4096 features) for different SYCL implementations.](image)

<table>
<thead>
<tr>
<th># data points (4096 features)</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>DPC++ (20220202)</th>
<th>hipSYCL (Feb 01)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>nd_range</td>
<td>nd_range</td>
</tr>
<tr>
<td>256</td>
<td>0.003</td>
<td>0.005</td>
<td>0.008</td>
<td>0.006</td>
</tr>
<tr>
<td>16384</td>
<td>0.287</td>
<td>0.576</td>
<td>1.242</td>
<td>0.543</td>
</tr>
<tr>
<td>65536</td>
<td>4.547</td>
<td>11.113</td>
<td>35.71</td>
<td>8.848</td>
</tr>
</tbody>
</table>

Source: www.nvidia.com

Marcel Breyer, University of Stuttgart, IPVS - SC : Performance Evolution of Different SYCL Implementations based on the PLSSVM Library


## NVIDIA A100

<table>
<thead>
<tr>
<th># data points (4096 features)</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>DPC++ (20221102)</th>
<th>hipSYCL (Oct 20)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>nd_range</td>
<td>nd_range</td>
</tr>
<tr>
<td>256</td>
<td>0.003</td>
<td>0.005</td>
<td>0.004</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−50 %</td>
<td>−67 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−4 %</td>
<td>+15 %</td>
</tr>
<tr>
<td>16384</td>
<td>0.287</td>
<td>0.576</td>
<td>0.358</td>
<td>0.565</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−71 %</td>
<td>+4 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+6 %</td>
<td>+0 %</td>
</tr>
<tr>
<td>65536</td>
<td>4.547</td>
<td>11.113</td>
<td>5.961</td>
<td>9.126</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−83 %</td>
<td>+3 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+7 %</td>
<td>+0 %</td>
</tr>
</tbody>
</table>

Source: [www.nvidia.com](http://www.nvidia.com)

Marcel Breyer, University of Stuttgart, IPVS - SC: Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
**NVIDIA A100: explaining the results using profiling**

<table>
<thead>
<tr>
<th>Resolution</th>
<th>DPC++ 20220202</th>
<th>DPC++ 20221102</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime</td>
<td>1.242 s</td>
<td>0.358 s</td>
<td>0.287 s</td>
</tr>
</tbody>
</table>
# NVIDIA A100: explaining the results using profiling

<table>
<thead>
<tr>
<th>Dimension</th>
<th>DPC++ 20220202</th>
<th>DPC++ 20221102</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384 x 4096</td>
<td>1.242 s</td>
<td>0.358 s</td>
<td>0.287 s</td>
</tr>
<tr>
<td>Runtime</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch efficiency</td>
<td>65.06 %</td>
<td>99.97 %</td>
<td>99.97 %</td>
</tr>
<tr>
<td>Avg divergent branches</td>
<td>3,972,456</td>
<td>170</td>
<td>170</td>
</tr>
</tbody>
</table>
NVIDIA A100: explaining the results using profiling

<table>
<thead>
<tr>
<th></th>
<th>DPC++ 20220202</th>
<th>DPC++ 20221102</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime</td>
<td>1.242 s</td>
<td>0.358 s</td>
<td>0.287 s</td>
</tr>
<tr>
<td>branch efficiency</td>
<td>65.06 %</td>
<td>99.97 %</td>
<td>99.97 %</td>
</tr>
<tr>
<td>avg divergent branches</td>
<td>3 972 456</td>
<td>170</td>
<td>170</td>
</tr>
<tr>
<td>atomics (instr. exec.)</td>
<td>1 418 372 005</td>
<td>30 117 888</td>
<td>18 097 152</td>
</tr>
</tbody>
</table>
NVIDIA A100: explaining the results using profiling

<table>
<thead>
<tr>
<th></th>
<th>DPC++ 20220202</th>
<th>DPC++ 20221102</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime</td>
<td>1.242 s</td>
<td>0.358 s</td>
<td>0.287 s</td>
</tr>
<tr>
<td>branch efficiency</td>
<td>65.06 %</td>
<td>99.97 %</td>
<td>99.97 %</td>
</tr>
<tr>
<td>avg divergent branches</td>
<td>3,972,456</td>
<td>170</td>
<td>170</td>
</tr>
<tr>
<td>atomics (instr. exec.)</td>
<td>1,418,372,005</td>
<td>30,117,888</td>
<td>18,097,152</td>
</tr>
<tr>
<td>register count</td>
<td>164</td>
<td>164</td>
<td>162</td>
</tr>
<tr>
<td>memory</td>
<td>more memory transfers involving shared memory and between global memory ←→ L1 cache</td>
<td>better usage of registers; overall 43% more memory throughput</td>
<td></td>
</tr>
</tbody>
</table>
AMD Radeon Pro VII

Source: www.amd.com

# data points (4096 features)

<table>
<thead>
<tr>
<th></th>
<th>HIP</th>
<th>OpenCL</th>
<th>DPC++ (20220202)</th>
<th>hipSYCL (Feb 01)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>nd_range</td>
<td>hierarchical</td>
</tr>
<tr>
<td>256</td>
<td>0.036</td>
<td>0.008</td>
<td>0.035</td>
<td>0.101</td>
</tr>
<tr>
<td>16384</td>
<td>6.93</td>
<td>1.275</td>
<td>6.532</td>
<td>38.93</td>
</tr>
<tr>
<td>65536</td>
<td>112.21</td>
<td>20.0</td>
<td>104.1</td>
<td>649.8</td>
</tr>
</tbody>
</table>

Source: Marcel Breyer, University of Stuttgart, IPVS - SC: Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
AMD Radeon Pro VII

Source: www.amd.com

<table>
<thead>
<tr>
<th># data points (4096 features)</th>
<th>HIP</th>
<th>OpenCL</th>
<th>DPC++ (20221102)</th>
<th>hipSYCL (Oct 20)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HIP</td>
<td>OpenCL</td>
<td>DPC++ nd_range</td>
<td>hipSYCL nd_range</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>hierarchical</td>
<td>nd_range</td>
</tr>
<tr>
<td>256</td>
<td>0.036</td>
<td>0.008</td>
<td>0.036</td>
<td>0.041</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+3 %</td>
<td>+14 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−12 %</td>
<td>−5 %</td>
</tr>
<tr>
<td>16384</td>
<td>6.93</td>
<td>1.275</td>
<td>6.547</td>
<td>7.327</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0 %</td>
<td>+12 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−16 %</td>
<td>+0 %</td>
</tr>
<tr>
<td>65536</td>
<td>112.21</td>
<td>20.0</td>
<td>104.4</td>
<td>115.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0 %</td>
<td>+12 %</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−18 %</td>
<td>+0 %</td>
</tr>
</tbody>
</table>
Basic idea of the used blocking scheme

\[ \tilde{Q} = \]

Note: each matrix entry \( Q_{ij} \) is calculated using the kernel function \( k(\vec{x}_i, \vec{x}_j) \)!
(e.g., dot products in the linear kernel)
Basic idea of the used blocking scheme

Note: each matrix entry $Q_{ij}$ is calculated using the kernel function $k(\vec{x}_i, \vec{x}_j)$!
(e.g., dot products in the linear kernel)
AMD Radeon Pro VII: Blocking Sizes

![Graph showing the runtime in seconds for different internal block sizes. The x-axis represents the internal block size, and the y-axis represents the runtime in seconds. The graph shows a curve with a minimum runtime at an internal block size of 6. The data points are marked as HIP (Heterogeneous Computing Platform).]

Marcel Breyer, University of Stuttgart, IPVS - SC: Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
AMD Radeon Pro VII: Blocking Sizes

![Graph showing runtime in seconds for different block sizes for HIP and OpenCL implementations. The graph plots runtime against the internal block size.]

Marcel Breyer, University of Stuttgart, IPVS - SC : Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
AMD Radeon Pro VII: Blocking Sizes

![Graph showing runtime in s vs INTERNAL_BLOCK_SIZE for different SYCL implementations.]

- HIP
- OpenCL
- DPC++ nd_range
- DPC++ hierarchical

16,384 × 4096
AMD Radeon Pro VII: Blocking Sizes

![Graph showing runtime in s for different blocking sizes and SYCL implementations.](image)
AMD Radeon Pro VII: updated runtimes with blocking size 4

Source: www.amd.com

<table>
<thead>
<tr>
<th># data points (4096 features)</th>
<th>HIP</th>
<th>OpenCL</th>
<th>DPC++ (20221102)</th>
<th>hipSYCL (Oct 20)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nd_range</td>
<td>hierarchical</td>
<td>nd_range</td>
<td>hierarchical</td>
</tr>
<tr>
<td>256</td>
<td>0.005</td>
<td>0.007</td>
<td>0.009</td>
<td>0.059</td>
</tr>
<tr>
<td></td>
<td>−86 %</td>
<td>−13 %</td>
<td>−75 %</td>
<td>−34 %</td>
</tr>
<tr>
<td>16384</td>
<td>0.891</td>
<td>1.335</td>
<td>1.775</td>
<td>44.24</td>
</tr>
<tr>
<td></td>
<td>−87 %</td>
<td>+5 %</td>
<td>−73 %</td>
<td>−26 %</td>
</tr>
<tr>
<td>65536</td>
<td>14.04</td>
<td>21.00</td>
<td>28.96</td>
<td>762.0</td>
</tr>
<tr>
<td></td>
<td>−87 %</td>
<td>+5 %</td>
<td>−72 %</td>
<td>+44 %</td>
</tr>
</tbody>
</table>
**AMD Radeon Pro VII: explaining the results using profiling**

<table>
<thead>
<tr>
<th>16384 x 4096</th>
<th>HIP</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERNAL_BLOCKING_SIZE</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>runtime</td>
<td>0.891 s</td>
<td>6.930 s</td>
</tr>
</tbody>
</table>
AMD Radeon Pro VII: explaining the results using profiling

<table>
<thead>
<tr>
<th></th>
<th>HIP</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERNAL_BLOCKING_SIZE</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>runtime</td>
<td>0.891 s</td>
<td>6.930 s</td>
</tr>
<tr>
<td>local data share</td>
<td>1024</td>
<td>1563</td>
</tr>
<tr>
<td>scratch memory</td>
<td>0</td>
<td>172</td>
</tr>
<tr>
<td>vector general purpose register</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>
## AMD Radeon Pro VII: explaining the results using profiling

<table>
<thead>
<tr>
<th>16384 x 4096</th>
<th>HIP</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERNAL_BLOCKING_SIZE</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>runtime</td>
<td>0.891 s</td>
<td>6.930 s</td>
</tr>
<tr>
<td>local data share</td>
<td>1024</td>
<td>1563</td>
</tr>
<tr>
<td>scratch memory</td>
<td>0</td>
<td>172</td>
</tr>
<tr>
<td>vector general purpose register</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>video memory fetches</td>
<td>84.29 GB</td>
<td>2039.79 GB</td>
</tr>
<tr>
<td>video memory writes</td>
<td>22.26 MB</td>
<td>1952.76 GB</td>
</tr>
<tr>
<td>bank conflicts (lower is better)</td>
<td>13.11 %</td>
<td>0.10 %</td>
</tr>
</tbody>
</table>
Intel Xeon E-2146G

![Graph showing runtime in s vs. # data points for different SYCL implementations](image)

<table>
<thead>
<tr>
<th># data points</th>
<th>OpenMP</th>
<th>OpenCL</th>
<th>DPC++ (20220202)</th>
<th>hipSYCL (Feb 01)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>nd_range</td>
<td>nd_range</td>
</tr>
<tr>
<td>256</td>
<td>0.016</td>
<td>0.085</td>
<td>0.290</td>
<td>0.282</td>
</tr>
<tr>
<td>4096</td>
<td>5.855</td>
<td>5.066</td>
<td>1.869</td>
<td>46.20</td>
</tr>
<tr>
<td>16384</td>
<td>97.16</td>
<td>76.77</td>
<td>29.84</td>
<td>711.6</td>
</tr>
</tbody>
</table>

Source: [www.intel.com](http://www.intel.com)
# data points (4096 features)

<table>
<thead>
<tr>
<th># data points</th>
<th>OpenMP</th>
<th>OpenCL</th>
<th>DPC++ (20221102)</th>
<th>hipSYCL (Oct 20)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>nd_range</td>
<td>hierarchical</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>nd_range</td>
<td>hierarchical</td>
</tr>
<tr>
<td>256</td>
<td>0.016</td>
<td>0.085</td>
<td>0.029</td>
<td>0.163</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-90 %</td>
<td>-7 %</td>
</tr>
<tr>
<td>4096</td>
<td>5.855</td>
<td>5.066</td>
<td>1.866</td>
<td>14.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0 %</td>
<td>-6 %</td>
</tr>
<tr>
<td>16384</td>
<td>97.16</td>
<td>76.77</td>
<td>29.73</td>
<td>234.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0 %</td>
<td>-7 %</td>
</tr>
</tbody>
</table>
Intel Xeon E-2146G

# data points (4096 features)

Source: www.intel.com

<table>
<thead>
<tr>
<th></th>
<th>OpenMP</th>
<th>OpenCL</th>
<th>DPC++ (20221102)</th>
<th>hipSYCL acc (Oct 20)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>nd_range</td>
<td>hierarchical</td>
</tr>
<tr>
<td>256</td>
<td>0.016</td>
<td>0.085</td>
<td>0.029</td>
<td>0.163</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-90 %</td>
<td>-7 %</td>
</tr>
<tr>
<td>4096</td>
<td>5.855</td>
<td>5.066</td>
<td>1.866</td>
<td>14.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0 %</td>
<td>-6 %</td>
</tr>
<tr>
<td>16384</td>
<td>97.16</td>
<td>76.77</td>
<td>29.73</td>
<td>234.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+0 %</td>
<td>-7 %</td>
</tr>
</tbody>
</table>
# Intel Xeon E-2146G: GCC vs. Clang hierarchical profiling results

<table>
<thead>
<tr>
<th></th>
<th>GCC 9.4.0</th>
<th>Clang (DPC++ 20221102)</th>
<th>Clang (DPC++ 20221102) omp.accelerated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.049 s</td>
<td>6.690 s</td>
<td>7.235 s</td>
</tr>
</tbody>
</table>

Marcel Breyer, University of Stuttgart, IPVS - SC : Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
Intel Xeon E-2146G: GCC vs. Clang hierarchical profiling results

<table>
<thead>
<tr>
<th></th>
<th>GCC 9.4.0</th>
<th>Clang (DPC++ 20221102)</th>
<th>Clang (DPC++ 20221102) omp.accelerated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>4.049s</td>
<td>6.690s</td>
<td>7.235s</td>
</tr>
</tbody>
</table>

// GCC: 92.7% of CPU-time
plssvm::sycl_generic::hierarchical_device_kernel_linear<double>::operator() // Clang: 37.7% + 34% + 13% = 84.7% of CPU-time
plssvm::sycl_generic::hierarchical_device_kernel_linear<double>::operator()(hipsycl::sycl::group<(int)2>)
 → const::{lambda(hipsycl::sycl::h_item<(int)2>)#3}::operator()
plssvm::sycl_generic::hierarchical_device_kernel_linear<double>::operator()
plssvm::sycl_generic::hierarchical_device_kernel_linear<double>::operator()(hipsycl::sycl::group<(int)2>)
 → const::{lambda(hipsycl::sycl::h_item<(int)2>)#2}::operator()
## Intel Xeon E-2146G: GCC vs. Clang hierarchical profiling results

<table>
<thead>
<tr>
<th></th>
<th>GCC 9.4.0</th>
<th>Clang (DPC++ 20221102)</th>
<th>Clang (DPC++ 20221102) omp.accelerated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis (4096 × 4096)</td>
<td>4.049 s</td>
<td>6.690 s</td>
<td>7.235 s</td>
</tr>
<tr>
<td>Memory Bound (% of Pipeline Slots)</td>
<td>9.6%</td>
<td>14.8%</td>
<td></td>
</tr>
<tr>
<td>Cache Bound (% of Clockticks)</td>
<td>10.2%</td>
<td>20%</td>
<td></td>
</tr>
<tr>
<td>FP Arith/Mem Rd Instr. Ratio</td>
<td>0.986</td>
<td>0.474</td>
<td></td>
</tr>
<tr>
<td>FP Arith/Mem Wr Instr. Ratio</td>
<td>1.042</td>
<td>0.868</td>
<td></td>
</tr>
<tr>
<td>Thread Oversubscription (% of CPU-time)</td>
<td>97.1%</td>
<td>6.2%</td>
<td></td>
</tr>
<tr>
<td>Spin and Overhead Time (% of CPU-time)</td>
<td>0.0%</td>
<td>12.6%</td>
<td></td>
</tr>
</tbody>
</table>
Key takeaway: the performance portability is good

Performance portability (application efficiency): (proposed by Pennycook, Sewall, and Lee in 2016)

\[
\mathcal{P}(a, p, H) = \begin{cases} 
\frac{|H|}{\sum_{i \in H} e_i(a, p)} & \text{if } i \text{ is supported } \forall i \in H \\
0 & \text{otherwise}
\end{cases}
\]

- **a**: an application (implicit matrix-vector multiplication)
- **p**: a specific problem (16 384 x 4096)
- **H**: a set of platforms (NVIDIA A100, AMD Radeon Pro VII, Intel Xeon)
Key takeaway: the performance portability is good

Performance portability (application efficiency): (proposed by Pennycook, Sewall, and Lee in 2016)

\[ P(a, p, H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a, p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \]

\( a \): an application (implicit matrix-vector multiplication)
\( p \): a specific problem \((16\,384 \times 4096)\)
\( H \): a set of platforms (NVIDIA A100, AMD Radeon Pro VII, Intel Xeon)

<table>
<thead>
<tr>
<th>version</th>
<th>CUDA</th>
<th>HIP</th>
<th>OpenMP</th>
<th>OpenCL</th>
<th>DPC++</th>
<th>hipSYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>20220202/Feb 01</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key takeaway: the performance portability is good

Performance portability (application efficiency): (proposed by Pennycook, Sewall, and Lee in 2016)

\[
\Phi(a, p, H) = \begin{cases} 
\frac{|H|}{\sum_{i \in H} e_i(a, p)} & \text{if } i \text{ is supported } \forall i \in H \\
0 & \text{otherwise}
\end{cases}
\]

\(a\): an application (implicit matrix-vector multiplication)
\(p\): a specific problem \((16\,384 \times 4096)\)
\(H\): a set of platforms \((\text{NVIDIA A100, AMD Radeon Pro VII, Intel Xeon})\)

<table>
<thead>
<tr>
<th>version</th>
<th>CUDA</th>
<th>HIP</th>
<th>OpenMP</th>
<th>OpenCL</th>
<th>DPC++</th>
<th>hipSYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>20220202/Feb 01</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
<td>49.92 %</td>
<td>41.15 %</td>
<td>50.82 %</td>
</tr>
</tbody>
</table>
Key takeaway: the performance portability is good

Performance portability (application efficiency): (proposed by Pennycook, Sewall, and Lee in 2016)

\[ \Phi(a, p, H) = \begin{cases} 
\frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a, p)}} & \text{if } i \text{ is supported } \forall i \in H \\
0 & \text{otherwise}
\end{cases} \]

- \(a\): an application (implicit matrix-vector multiplication)
- \(p\): a specific problem \((16384 \times 4096)\)
- \(H\): a set of platforms \(\text{(NVIDIA A100, AMD Radeon Pro VII, Intel Xeon)}\)

<table>
<thead>
<tr>
<th>version</th>
<th>CUDA</th>
<th>HIP</th>
<th>OpenMP</th>
<th>OpenCL</th>
<th>DPC++</th>
<th>hipSYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>20220202/Feb 01</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
<td>49.92 %</td>
<td>41.15 %</td>
<td>50.82 %</td>
</tr>
<tr>
<td>20221102/Oct 20</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
<td>49.83 %</td>
<td>69.23 %</td>
<td>52.40 %</td>
</tr>
</tbody>
</table>
Conclusion

- fine-tuning hyperparameter (like the blocking size) can have a major impact on the performance
- profiling SYCL code (DPC++ and hipSYCL) is as easy as profiling native code
Conclusion

- fine-tuning hyperparameter (like the blocking size) can have a major impact on the performance
- profiling SYCL code (DPC++ and hipSYCL) is as easy as profiling native code
- installing new DPC++ or hipSYCL versions may drastically increase performance
  - the DPC++ nd_range performance on NVIDIA GPUs drastically improved ($-80\%$)
  - support for omp.accelerated on CPUs in newer hipSYCL versions ($-90\%$)

SYCL provides a better performance portability than OpenCL. In our case, DPC++ has the best performance portability with $P(a, p, H) = 69.23\%$ in addition: SYCL needs drastically fewer lines of code when compared to OpenCL. If performance portability is important, SYCL should be chosen over OpenCL!
Conclusion

- fine-tuning hyperparameter (like the blocking size) can have a major impact on the performance
- profiling SYCL code (DPC++ and hipSYCL) is as easy as profiling native code
- installing new DPC++ or hipSYCL versions may drastically increase performance
  - the DPC++ nd_range performance on NVIDIA GPUs drastically improved (−80 %)
  - support for omp.accelerated on CPUs in newer hipSYCL versions (−90 %)
- SYCL provides a better performance portability than OpenCL
  - in our case, DPC++ has the best performance portability with $\Phi(a, p, H) = 69.23 \%$
- in addition: SYCL needs drastically fewer lines of code when compared to OpenCL
  - in our case, more than the 300 lines of code

Marcel Breyer, University of Stuttgart, IPVS - SC : Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
Conclusion

- fine-tuning hyperparameter (like the blocking size) can have a major impact on the performance
- profiling SYCL code (DPC++ and hipSYCL) is as easy as profiling native code
- installing new DPC++ or hipSYCL versions may drastically increase performance
  - the DPC++ nd_range performance on NVIDIA GPUs drastically improved (−80%)
  - support for omp.accelerated on CPUs in newer hipSYCL versions (−90%)
- SYCL provides a better performance portability than OpenCL
  - in our case, DPC++ has the best performance portability with $\Phi(a, p, H) = 69.23\%$
- in addition: SYCL needs drastically fewer lines of code when compared to OpenCL
  - in our case, more the 300 lines of code

\textbf{If performance portability is important, SYCL should be chosen over OpenCL!}
University of Stuttgart
Germany

Thank you for your attention!

Marcel Breyer
Marcel.Breyer@ipvs.uni-stuttgart.de

Alexander Van Craen
Alexander.Van-Craen@ipvs.uni-stuttgart.de

Prof. Dr. Dirk Pflüger
Dirk.Pflueger@ipvs.uni-stuttgart.de
Further reading about PLSSVM


Additional resources
Basics of Support Vector Machines (SVMs) (proposed by Boser, Guyon, and Vapnik in 1992)

supervised machine learning: binary classification

\[ y = \text{sgn} \left( \langle \vec{w}, \vec{x} \rangle + b \right) \]
PLSSVM supports many different backends

Backend and target platform selectable at runtime

Marcel Breyer, University of Stuttgart, IPVS - SC : Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
Different SYCL kernel invocation types

reverse all elements in an array

```cpp
sycl::nd_range<1> exec{ global, local }; // local memory
local_accessor<int> loc{ local, cgh }; // local memory
cgh.parallel_for(exec, [=](sycl::nd_item<1> item) {
    const int idx = item.get_global_linear_id();
    const int priv = n - idx - 1; // private memory
    loc[idx] = res[idx];
    // explicit barrier
    sycl::group_barrier(item.get_group());
    res[idx] = loc[priv];
});

cgh.parallel_for_work_group(global, local, [=](sycl::group<1> group){
    int loc[LOCAL_SIZE]; // local memory
    sycl::private_memory<int> priv{ group }; // private memory
    group.parallel_for_work_item([&](sycl::h_item<1> item) {
        const int idx = item.get_local_id(0);
        priv(item) = n - idx - 1;
        loc[idx] = res[idx];
    });
    // implicit barrier
    group.parallel_for_work_item([&](sycl::h_item<1> item) {
        const int idx = item.get_local_id(0);
        res[idx] = loc[priv(item)];
    });
});
```

Marcel Breyer, University of Stuttgart, IPVS - SC : Performance Evolution of Different SYCL Implementations based on the PLSSVM Library
Used software and hardware

NVIDIA A100
CUDA 11.4.3
Driver Version 510.85.02

Radeon Pro VII
ROCm 5.3.0
Driver Version 5.18.2.22.40

Intel Xeon E-2146G
Intel DevCloud

DPC++
OpenSource LLVM fork

hipSYCL
OpenSource

Source: www.nvidia.com
Source: www.amd.com
Source: www.intel.com
NVIDIA A100: varying blocking size

![Graph showing varying blocking sizes for different SYCL implementations on NVIDIA A100](image)

- **CUDA**
- **OpenCL**
- **DPC++ nd_range**
- **DPC++ hierarchical**
- **hipSYCL nd_range**
- **hipSYCL hierarchical**

**Runtime in s**

**INTERNAL_BLOCK_SIZE**

16384 x 4096
Key takeaways: new versions improve the performance

<table>
<thead>
<tr>
<th></th>
<th>DPC++</th>
<th>hipSYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nd_range</td>
<td>nd_range</td>
</tr>
<tr>
<td></td>
<td>hierarchical</td>
<td>hierarchical</td>
</tr>
<tr>
<td>NVIDIA A100</td>
<td>↑</td>
<td>→</td>
</tr>
<tr>
<td>AMD Radeon Pro VII</td>
<td>→</td>
<td>↓</td>
</tr>
<tr>
<td>Intel Xeon E-2146G</td>
<td>→</td>
<td>/↑</td>
</tr>
</tbody>
</table>
## Key takeaways: SYCL needs fewer lines of code than OpenCL

<table>
<thead>
<tr>
<th></th>
<th>kernel function</th>
<th>device discovery</th>
<th>other setup and bookkeeping code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>67</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>HIP</td>
<td>67</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td>29</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>OpenCL</td>
<td>65</td>
<td>96</td>
<td>60 (3 custom RAII classes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>27 (custom atomic add)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>166 (kernel compilation &amp; caching)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>83 (custom sha256 for caching)</td>
</tr>
<tr>
<td>SYCL</td>
<td>nd_range</td>
<td>71</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>hierarchical</td>
<td></td>
<td>20 (used function object)</td>
</tr>
<tr>
<td></td>
<td>99</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>