The 11th International workshop on OpenCL and SYCL

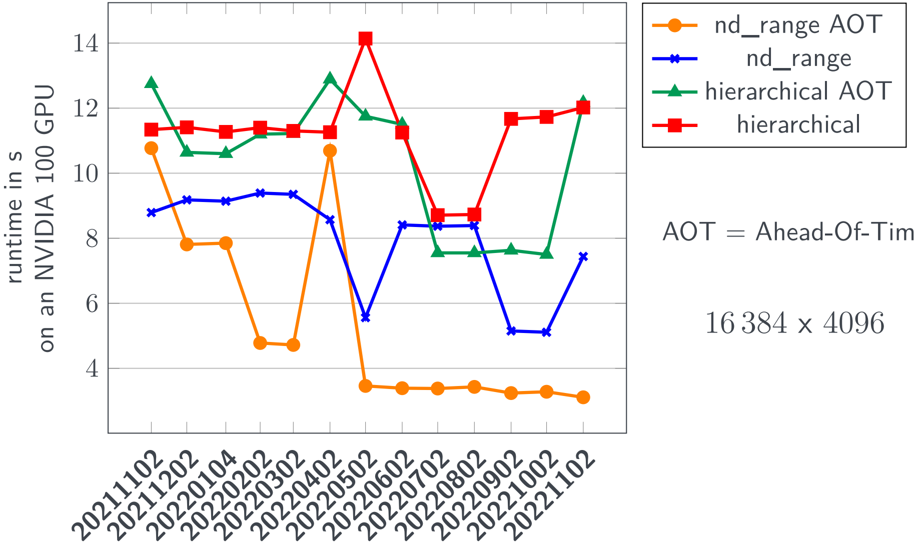
IWOCL & SYCLcon 2023

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



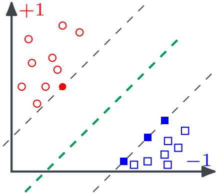
AOT = Ahead-Of-Time

16 384 x 4096

What to know about PLSSVM

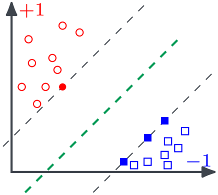
1

Support Vector Machines (SVMs) and their problems



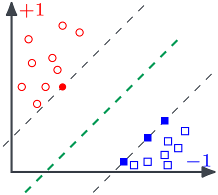
- SVMs as supervised machine learning technique
- originally meant for binary classification

Support Vector Machines (SVMs) and their problems



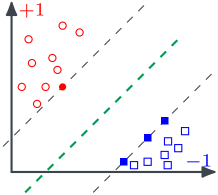
- SVMs as supervised machine learning technique
- originally meant for binary classification
- SVMs have to solve a convex quadratic problem
  ➔ state-of-the-art: Sequential Minimal Optimization (SMO) (proposed by Platt in 1998)
  ➔ inherently sequential algorithm

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

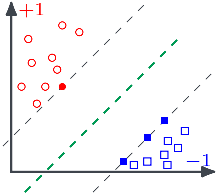
Support Vector Machines (SVMs) and their problems



- 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:

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

where $\boldsymbol{Q}$ is the kernel matrix according to

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

We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

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

where $\boldsymbol{Q}$ is the kernel matrix according to

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

➔ $\boldsymbol{Q}$ is symmetric positive-definite

We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

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

where $\boldsymbol{Q}$ is the kernel matrix according to

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

- ➔ $\boldsymbol{Q}$ is symmetric positive-definite
- ➔ Conjugate Gradient algorithm: (variant of Shewchuk et al.)

$i \leftarrow 0$
$r \leftarrow b - Ax$
$d \leftarrow r$
$\delta_{new} \leftarrow r^T r$
$\delta_0 \leftarrow \delta_{new}$
while $i < i_{max}$ and $\delta_{new} > \epsilon^2 \delta_0$ **do**
    $q \leftarrow Ad$
    $\alpha \leftarrow \frac{\delta_{new}}{d^T q}$
    $x \leftarrow x + \alpha d$
    if i is divisible by 50 **then**
        $r \leftarrow b - Ax$
    else
        $r \leftarrow r - \alpha q$
    end if
    $\delta_{old} \leftarrow \delta_{new}$
    $\delta_{new} \leftarrow r^T r$
    $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \leftarrow r + \beta d$
    $i \leftarrow i + 1$
end while

We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

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

where $\boldsymbol{Q}$ is the kernel matrix according to

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \qquad \left(\text{with } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

- ➔ $\boldsymbol{Q}$ is symmetric positive-definite
- ➔ Conjugate Gradient algorithm: (variant of Shewchuk et al.)
  - Setup or constant operations ➔ host

$i \leftarrow 0$
$r \leftarrow b - Ax$
$d \leftarrow r$
$\delta_{new} \leftarrow r^T r$
$\delta_0 \leftarrow \delta_{new}$
while $i < i_{max}$ and $\delta_{new} > \epsilon^2 \delta_0$ **do**
    $q \leftarrow Ad$
    $\alpha \leftarrow \frac{\delta_{new}}{d^T q}$
    $x \leftarrow x + \alpha d$
    if i is divisible by 50 **then**
        $r \leftarrow b - Ax$
    else
        $r \leftarrow r - \alpha q$
    end if
    $\delta_{old} \leftarrow \delta_{new}$
    $\delta_{new} \leftarrow r^T r$
    $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \leftarrow r + \beta d$
    $i \leftarrow i + 1$
end while

We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

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

where $\boldsymbol{Q}$ is the kernel matrix according to

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

- ➔ $\boldsymbol{Q}$ is symmetric positive-definite
- ➔ Conjugate Gradient algorithm: (variant of Shewchuk et al.)
  - Setup or constant operations ➔ host
  - BLAS Level 1 ➔ host

$i \leftarrow 0$
$r \leftarrow b - Ax$
$d \leftarrow r$
$\delta_{new} \leftarrow r^T r$
$\delta_0 \leftarrow \delta_{new}$
while $i < i_{max}$ and $\delta_{new} > \epsilon^2 \delta_0$ **do**
    $q \leftarrow Ad$
    $\alpha \leftarrow \frac{\delta_{new}}{d^T q}$
    $x \leftarrow x + \alpha d$
    if i is divisible by 50 **then**
        $r \leftarrow b - Ax$
    else
        $r \leftarrow r - \alpha q$
    end if
    $\delta_{old} \leftarrow \delta_{new}$
    $\delta_{new} \leftarrow r^T r$
    $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \leftarrow r + \beta d$
    $i \leftarrow i + 1$
end while

We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

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

where $\boldsymbol{Q}$ is the kernel matrix according to

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \qquad \left(\text{with } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

- ➔ $\boldsymbol{Q}$ is symmetric positive-definite
- ➔ Conjugate Gradient algorithm: (variant of Shewchuk et al.)
  - Setup or constant operations ➔ host
  - BLAS Level 1 ➔ host
  - BLAS Level 2 ➔ device

$i \leftarrow 0$
$r \leftarrow b - Ax$
$d \leftarrow r$
$\delta_{new} \leftarrow r^T r$
$\delta_0 \leftarrow \delta_{new}$
while $i < i_{max}$ and $\delta_{new} > \epsilon^2 \delta_0$ **do**
    $q \leftarrow Ad$
    $\alpha \leftarrow \frac{\delta_{new}}{d^T q}$
    $x \leftarrow x + \alpha d$
    if i is divisible by 50 **then**
        $r \leftarrow b - Ax$
    else
        $r \leftarrow r - \alpha q$
    end if
    $\delta_{old} \leftarrow \delta_{new}$
    $\delta_{new} \leftarrow r^T r$
    $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \leftarrow r + \beta d$
    $i \leftarrow i + 1$
end while

We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

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

where $\boldsymbol{Q}$ is the kernel matrix according to

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \qquad \left(\text{with } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

- → $\boldsymbol{Q}$ is symmetric positive-definite
- → Conjugate Gradient algorithm: (variant of Shewchuk et al.)
  - Setup or constant operations → host
  - BLAS Level 1 → host
  - BLAS Level 2 → device
- → $\boldsymbol{Q} \in \mathbb{R}^{\text{num\_data\_points} \times \text{num\_data\_points}}$

$i \leftarrow 0$
$r \leftarrow b - Ax$
$d \leftarrow r$
$\delta_{new} \leftarrow r^T r$
$\delta_0 \leftarrow \delta_{new}$
while $i < i_{max}$ and $\delta_{new} > \epsilon^2 \delta_0$ **do**
    $q \leftarrow Ad$
    $\alpha \leftarrow \frac{\delta_{new}}{d^T q}$
    $x \leftarrow x + \alpha d$
    if i is divisible by 50 **then**
        $r \leftarrow b - Ax$
    else
        $r \leftarrow r - \alpha q$
    end if
    $\delta_{old} \leftarrow \delta_{new}$
    $\delta_{new} \leftarrow r^T r$
    $\beta \leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \leftarrow r + \beta d$
    $i \leftarrow i + 1$
end while

We parallelized the most complex operations in the CG algorithm

LS-SVMs solve the system of linear equations:

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

where $\boldsymbol{Q}$ is the kernel matrix according to

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij} \qquad \left(\text{with } \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

- → $\boldsymbol{Q}$ is symmetric positive-definite
- → Conjugate Gradient algorithm: (variant of Shewchuk et al.)
  - Setup or constant operations → host
  - BLAS Level 1 → host
  - BLAS Level 2 → device
- → $\boldsymbol{Q} \in \mathbb{R}^{\text{num\_data\_points} \times \text{num\_data\_points}}$
- → implicitly calculate $\boldsymbol{Q}$ in each iteration

```
i ← 0
r ← b − Ax
d ← r
δ_new ← r^T r
δ_0 ← δ_new
while i < i_max and δ_new > ε^2 δ_0 do
    q ← Ad
    α ← δ_new / (d^T q)
    x ← x + αd
    if i is divisible by 50 then
        r ← b − Ax
    else
        r ← r − αq
    end if
    δ_old ← δ_new
    δ_new ← r^T r
    β ← δ_new / δ_old
    d ← r + βd
    i ← i + 1
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

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

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

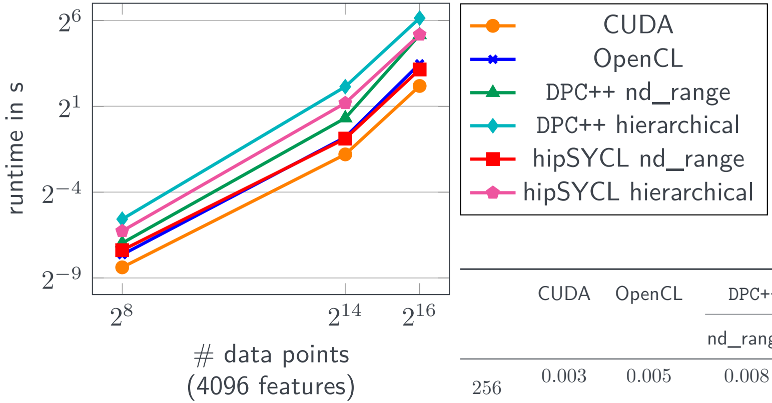
PLSSVM

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

New results and findings

2

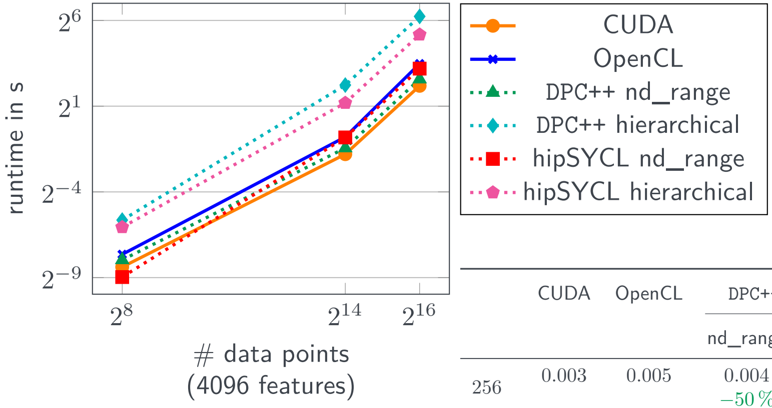
NVIDIA A100



Source: www.nvidia.com

	CUDA	OpenCL	DPC++ (20220202)		hipSYCL (Feb 01)	
			nd_range	hierarchical	nd_range	hierarchical
256	0.003	0.005	0.008	0.021	0.006	0.013
16 384	0.287	0.576	1.242	4.418	0.543	2.281
65 536	4.547	11.113	35.71	70.42	8.848	36.10

NVIDIA A100



Source: www.nvidia.com

	CUDA	OpenCL	DPC++ (20221102)		hipSYCL (Oct 20)	
			nd_range	hierarchical	nd_range	hierarchical
256	0.003	0.005	0.004 −50 %	0.02 −4 %	0.002 −67 %	0.015 +15 %
16 384	0.287	0.576	0.358 −71 %	4.695 +6 %	0.565 +4 %	2.279 +0 %
65 536	4.547	11.113	5.961 −83 %	75.31 +7 %	9.126 +3 %	36.11 +0 %

NVIDIA A100: explaining the results using profiling

$16\,384 \times 4096$	DPC++ 20220202	DPC++ 20221102	CUDA
runtime	$1.242\,\text{s}$	$0.358\,\text{s}$	$0.287\,\text{s}$

NVIDIA A100: explaining the results using profiling

16 384 x 4096	DPC++ 20220202	DPC++ 20221102	CUDA
runtime	1.242 s	0.358 s	0.287 s
branch efficiency	65.06 %	99.97 %	99.97 %
avg divergent branches	3 972 456	170	170

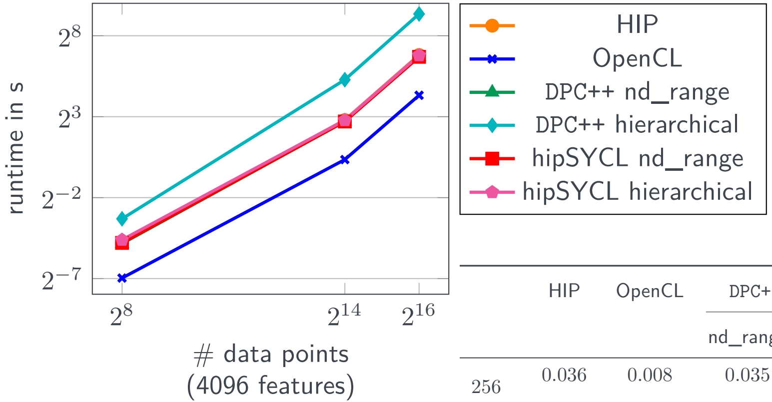
NVIDIA A100: explaining the results using profiling

16 384 x 4096	DPC++ 20220202	DPC++ 20221102	CUDA
runtime	1.242 s	0.358 s	0.287 s
branch efficiency	65.06 %	99.97 %	99.97 %
avg divergent branches	3 972 456	170	170
atomics (instr. exec.)	1 418 372 005	30 117 888	18 097 152

NVIDIA A100: explaining the results using profiling

16 384 x 4096	DPC++ 20220202	DPC++ 20221102	CUDA
runtime	1.242 s	0.358 s	0.287 s
branch efficiency	65.06 %	99.97 %	99.97 %
avg divergent branches	3 972 456	170	170
atomics (instr. exec.)	1 418 372 005	30 117 888	18 097 152
register count	164	164	162
memory	more memory transfers involving shared memory and between global memory ⟷ L1 cache		better usage of registers; overall 43 % more memory throughput

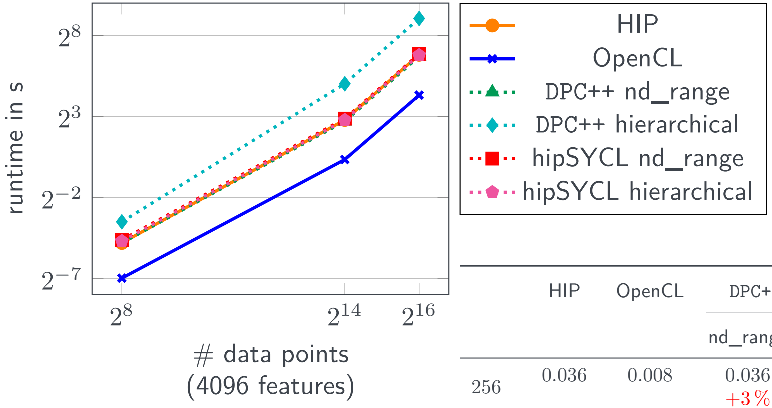
AMD Radeon Pro VII



Source: `www.amd.com`

	HIP	OpenCL	DPC++ (20220202)		hipSYCL (Feb 01)	
			nd_range	hierarchical	nd_range	hierarchical
256	0.036	0.008	0.035	0.101	0.036	0.041
16 384	6.93	1.275	6.532	38.93	6.517	6.875
65 536	112.21	20.0	104.1	649.8	103.6	110.4

AMD Radeon Pro VII

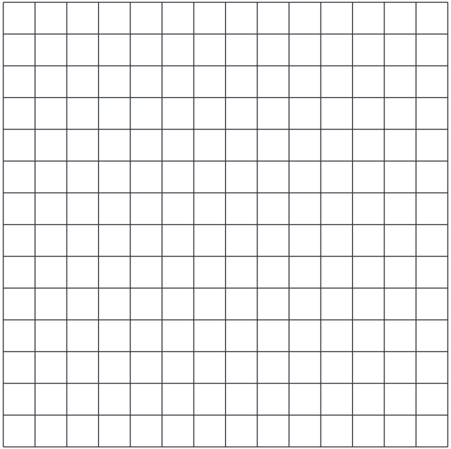


Source: www.amd.com

	HIP	OpenCL	DPC++ (20221102)		hipSYCL (Oct 20)	
			nd_range	hierarchical	nd_range	hierarchical
256	0.036	0.008	0.036 +3 %	0.089 −12 %	0.041 +14 %	0.039 −5 %
16 384	6.93	1.275	6.547 +0 %	32.551 −16 %	7.327 +12 %	6.902 +0 %
65 536	112.21	20.0	104.4 +0 %	530.2 −18 %	115.8 +12 %	110.5 +0 %

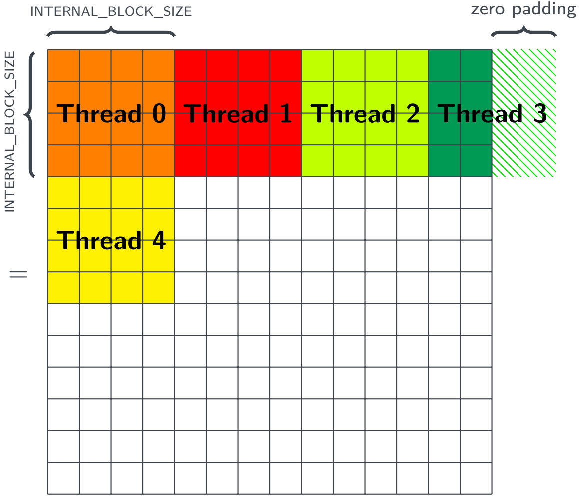
Basic idea of the used blocking scheme

$\bar{Q} =$



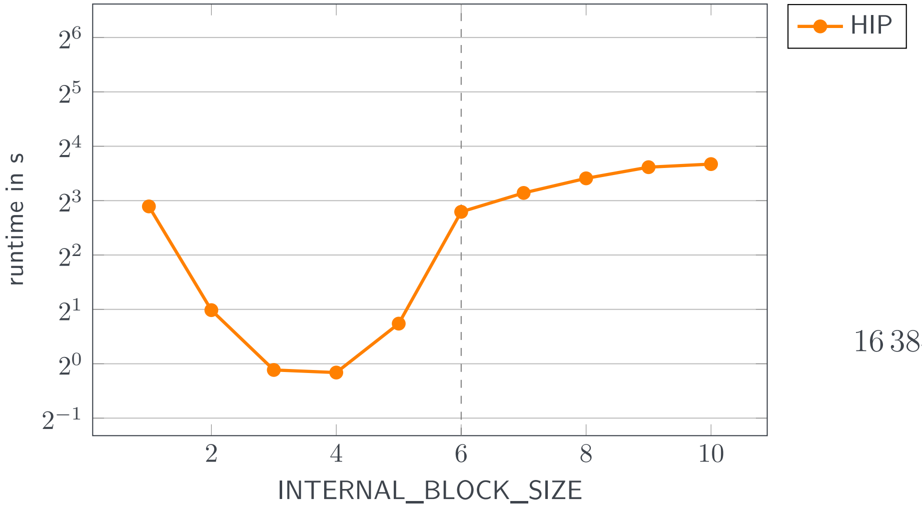
Note: each matrix entry Q_{ij} is calculated using the kernel function $k(\vec{x}_i, \vec{x}_j)$!
(e.g., dot products in the linear kernel)

Basic idea of the used blocking scheme

$\bar{Q} =$ 

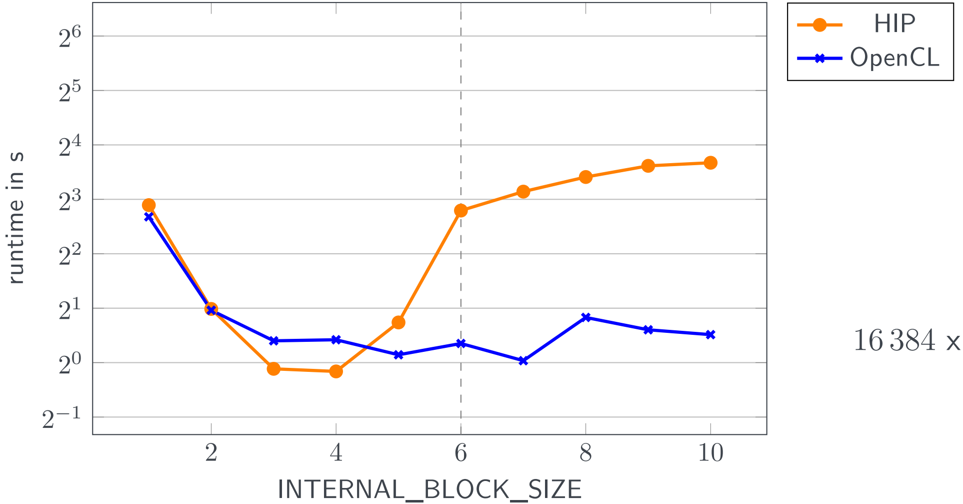
Note: each matrix entry Q_{ij} is calculated using the kernel function $k(\vec{x}_i, \vec{x}_j)$!
(e.g., dot products in the linear kernel)

AMD Radeon Pro VII: Blocking Sizes



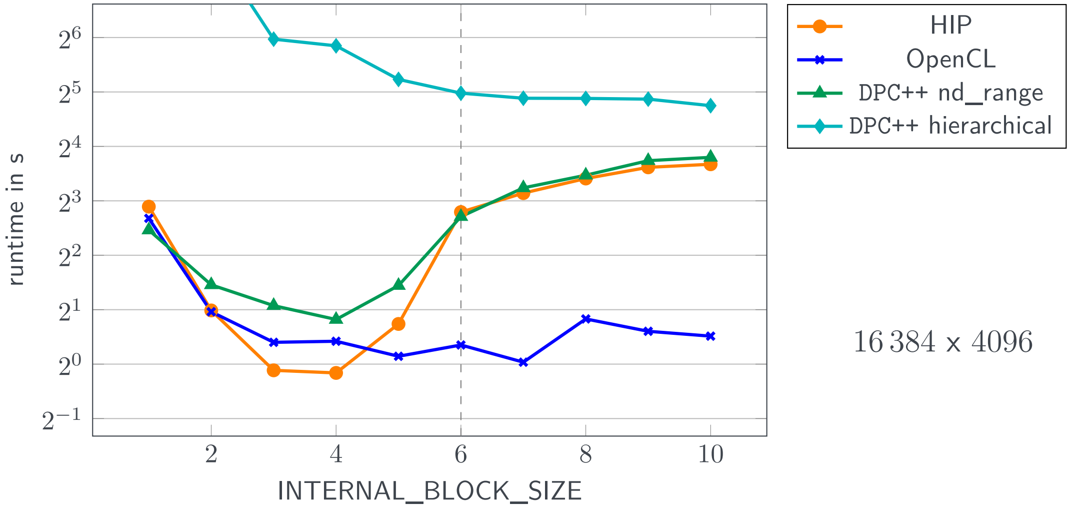
16 384 x 4096

AMD Radeon Pro VII: Blocking Sizes



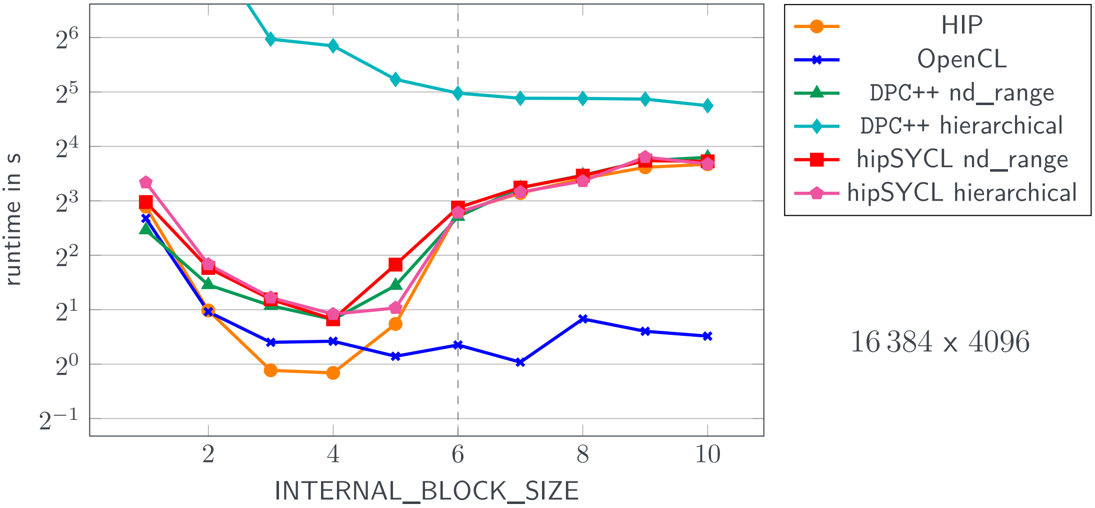
16 384 x 4096

AMD Radeon Pro VII: Blocking Sizes

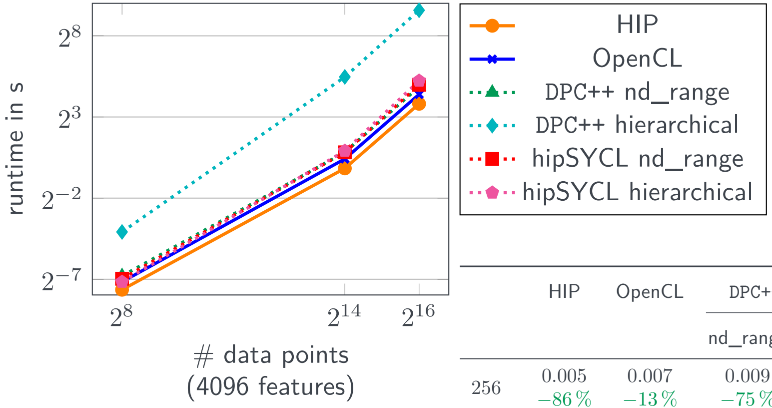


16 384 x 4096

AMD Radeon Pro VII: Blocking Sizes



AMD Radeon Pro VII: updated runtimes with blocking size 4



Source: www.amd.com

	HIP	OpenCL	DPC++ (20221102)		hipSYCL (Oct 20)	
			nd_range	hierarchical	nd_range	hierarchical
256	0.005 −86 %	0.007 −13 %	0.009 −75 %	0.059 −34 %	0.008 −80 %	0.007 −82 %
16 384	0.891 −87 %	1.335 +5 %	1.775 −73 %	44.24 −26 %	1.767 −76 %	1.882 −73 %
65 536	14.04 −87 %	21.00 +5 %	28.96 −72 %	762.0 +44 %	31.58 −73 %	37.57 −66 %

AMD Radeon Pro VII: explaining the results using profiling

$16\,384$ x 4096	HIP		OpenCL	
INTERNAL_BLOCKING_SIZE	4	6	4	6
runtime	0.891 s	6.930 s	1.335 s	1.275 s

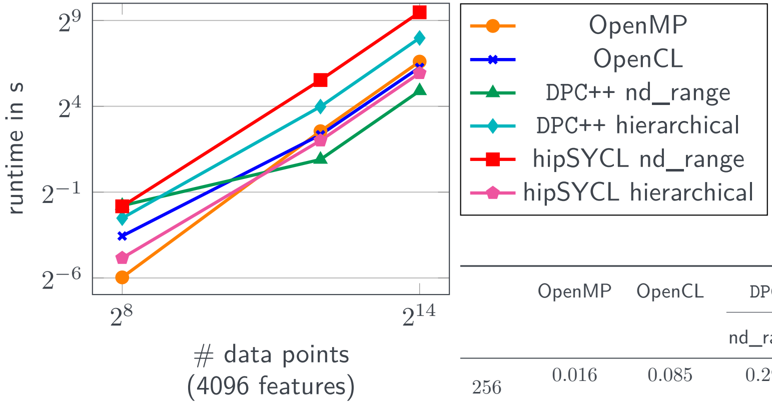
AMD Radeon Pro VII: explaining the results using profiling

$16\,384$ x 4096	HIP		OpenCL	
`INTERNAL_BLOCKING_SIZE`	4	6	4	6
runtime	0.891 s	6.930 s	1.335 s	1.275 s
local data share	1024	1563	1024	1563
scratch memory	0	172	0	0
vector general purpose register	64	64	56	108

AMD Radeon Pro VII: explaining the results using profiling

16 384 x 4096	HIP		OpenCL	
INTERNAL_BLOCKING_SIZE	4	6	4	6
runtime	0.891 s	6.930 s	1.335 s	1.275 s
local data share	1024	1563	1024	1563
scratch memory	0	172	0	0
vector general purpose register	64	64	56	108
video memory fetches	84.29 GB	2039.79 GB	80.69 GB	53.48 GB
video memory writes	22.26 MB	1952.76 GB	19.45 MB	12.73 MB
bank conflicts (lower is better)	13.11 %	0.10 %	20.34 %	4.74 %

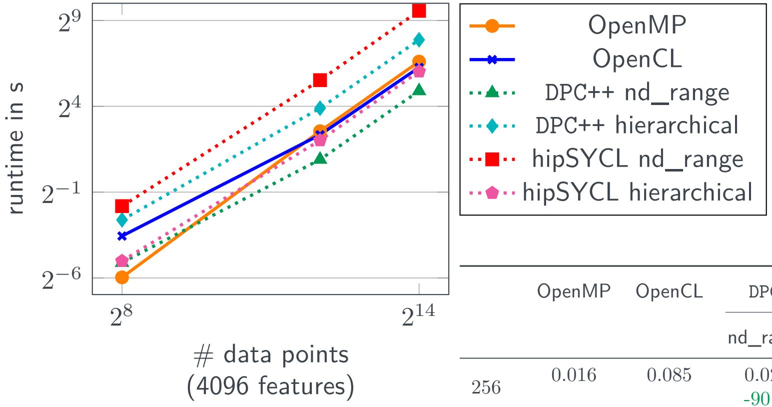
Intel Xeon E-2146G



Source: www.intel.com

	OpenMP	OpenCL	DPC++ (20220202)		hipSYCL (Feb 01)	
			nd_range	hierarchical	nd_range	hierarchical
256	0.016	0.085	0.290	0.175	0.282	0.035
4096	5.855	5.066	1.869	15.82	46.20	4.020
16 384	97.16	76.77	29.84	252.2	711.6	61.04

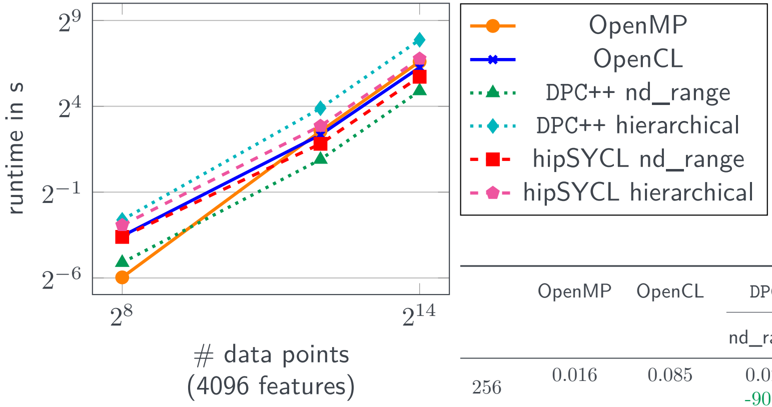
Intel Xeon E-2146G



Source: www.intel.com

	OpenMP	OpenCL	DPC++ (20221102)		hipSYCL (Oct 20)	
			nd_range	hierarchical	nd_range	hierarchical
256	0.016	0.085	0.029 -90 %	0.163 −7 %	0.284 +1 %	0.031 −11 %
4096	5.855	5.066	1.866 +0 %	14.81 −6 %	45.95 +0 %	4.049 +0 %
16 384	97.16	76.77	29.73 +0 %	234.3 −7 %	755.1 +6 %	65.15 +7 %

Intel Xeon E-2146G



Source: `www.intel.com`

	OpenMP	OpenCL	DPC++ (20221102)		hipSYCL acc (Oct 20)	
			nd_range	hierarchical	nd_range	hierarchical
256	0.016	0.085	0.029 −90 %	0.163 −7 %	0.082 −71 %	0.132 +277 %
4096	5.855	5.066	1.866 +0 %	14.81 −6 %	3.521 −92 %	7.235 +80 %
16 384	97.16	76.77	29.73 +0 %	234.3 −7 %	52.72 −93 %	109.2 +79 %

Intel Xeon E-2146G: GCC vs. Clang hierarchical profiling results

GCC 9.4.0	Clang (DPC++ 20221102)	Clang (DPC++ 20221102) omp.accelerated
4.049 s	6.690 s	7.235 s

Intel Xeon E-2146G: GCC vs. Clang hierarchical profiling results

GCC 9.4.0	Clang (DPC++ 20221102)	Clang (DPC++ 20221102) omp.accelerated
4.049 s	6.690 s	7.235 s

```
// 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

GCC 9.4.0	Clang (DPC++ 20221102)	Clang (DPC++ 20221102) omp.accelerated
4.049 s	6.690 s	7.235 s

analysis (4096×4096)	GCC	Clang
Memory Bound (% of Pipeline Slots)	9.6 %	14.8 %
Cache Bound (% of Clockticks)	10.2 %	20 %
FP Arith/Mem Rd Instr. Ratio	0.986	0.474
FP Arith/Mem Wr Instr. Ratio	1.042	0.868
Thread Oversubscription (% of CPU-time)	97.1 %	6.2 %
Spin and Overhead Time (% of CPU-time)	0.0 %	12.6 %

Key takeaway: the performance portability is good

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

$$\Psi(a, p, H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a, p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

a :	an application	(implicit matrix-vector multiplication)
p :	a specific problem	$(16\,384$ 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)

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

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)

version	CUDA	HIP	OpenMP	OpenCL	DPC++	hipSYCL
20220202/Feb 01	0 %	0 %	0 %			

Key takeaway: the performance portability is good

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

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

a : an application (implicit matrix-vector multiplication)

p : a specific problem ($16\,384$ x 4096)

H : a set of platforms (NVIDIA A100, AMD Radeon Pro VII, Intel Xeon)

version	CUDA	HIP	OpenMP	OpenCL	DPC++	hipSYCL
20220202/Feb 01	$0\,\%$	$0\,\%$	$0\,\%$	$49.92\,\%$	$41.15\,\%$	$50.82\,\%$

Key takeaway: the performance portability is good

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

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

a : an application (implicit matrix-vector multiplication)

p : a specific problem ($16\,384$ x 4096)

H : a set of platforms (NVIDIA A100, AMD Radeon Pro VII, Intel Xeon)

version	CUDA	HIP	OpenMP	OpenCL	DPC++	hipSYCL
20220202/Feb 01	$0\,\%$	$0\,\%$	$0\,\%$	$49.92\,\%$	$41.15\,\%$	$50.82\,\%$
20221102/Oct 20	$0\,\%$	$0\,\%$	$0\,\%$	$49.83\,\%$	$69.23\,\%$	$52.40\,\%$

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 ($-90\,\%$)

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

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

If performance portability is important, SYCL should be chosen over OpenCL!

University of Stuttgart
Germany

Thank your 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

[1] Alexander Van Craen, Marcel Breyer, and Dirk Pflüger. "PLSSVM: A (multi-)GPGPU-accelerated Least Squares Support Vector Machine". In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 818–827. DOI: 10.1109/IPDPSW55747.2022.00138.

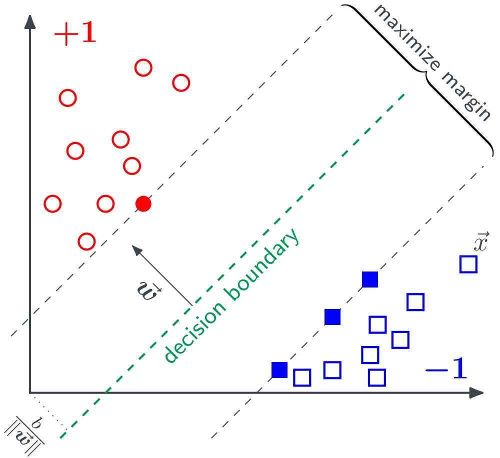
[2] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. "A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware". In: *International Workshop on OpenCL*. IWOCL'22. Bristol, United Kingdom, United Kingdom: Association for Computing Machinery, 2022. ISBN: 9781450396585. DOI: 10.1145/3529538.3529980. URL: https://doi.org/10.1145/3529538.3529980.

[3] Alexander Van Craen, Marcel Breyer, and Dirk Pflüger. "PLSSVM—Parallel Least Squares Support Vector Machine". In: *Software Impacts* 14 (2022), p. 100343. ISSN: 2665-9638. DOI: https://doi.org/10.1016/j.simpa.2022.100343. URL: https://www.sciencedirect.com/science/article/pii/S2665963822000641.

Addi-
tional
resources

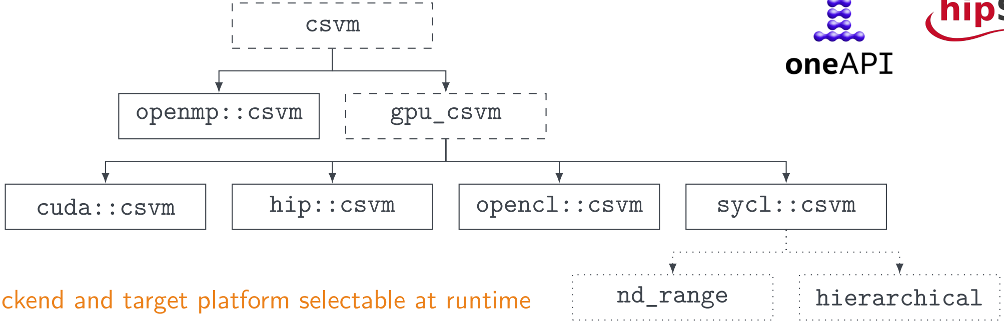
Basics of Support Vector Machines (SVMs) (proposed by Boser, Guyon, and Vapnik in 1992)

supervised machine learning: binary classification



$$y = \operatorname{sgn}\left(\langle \vec{\boldsymbol{w}}, \vec{x} \rangle + b\right)$$

PLSSVM supports many different backends



backend and target platform selectable at runtime

Different SYCL kernel invocation types

reverse all elements in an array

```
sycl::nd_range<1> exec{ global, local };
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];
      sycl::group_barrier(item.get_group());  // explicit barrier
      res[idx] = loc[priv];
   });
```

nd_range (bottom-up)

(CUDA HIP OpenCL)

```
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)];
         });
   });
```

hierarchical (top-down)

Used software and hardware

Source: www.nvidia.com

NVIDIA A100
CUDA 11.4.3
Driver Version 510.85.02

Source: www.amd.com

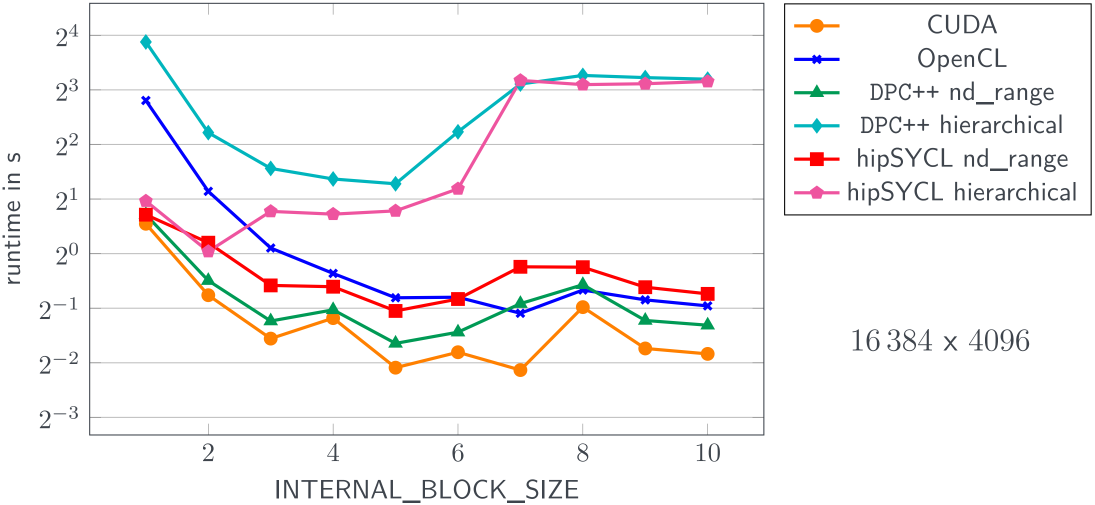
Radeon Pro VII
ROCm 5.3.0
Driver Version 5.18.2.22.40

Source: www.intel.com

Intel Xeon E-2146G
Intel DevCloud

`DPC++`	sycl-nightly/20220202 (February 02, 2022)
OpenSource LLVM fork	sycl-nightly/20221102 (November 02, 2022)
hipSYCL	develop 6962942 (February 01, 2022)
OpenSource	develop 012e16d (October 20, 2022)

NVIDIA A100: varying blocking size



16 384 x 4096

Key takeaways: new versions improve the performance

	DPC++		hipSYCL	
	nd_range	hierarchical	nd_range	hierarchical
NVIDIA A100	↑	→	→	→
AMD Radeon Pro VII	→	↑	↓	→
Intel Xeon E-2146G	→	↗	→/↑	→/↓

Key takeaways: SYCL needs fewer lines of code than OpenCL

		kernel function	device discovery	other setup and bookkeeping code
CUDA		67	-	-
HIP		67	-	-
OpenMP		29	-	-
OpenCL		65	96	166 (kernel compilation & caching) 83 (custom sha256 for caching) 60 (3 custom RAII classes) 27 (custom atomic add) → 336
SYCL	nd_range	71	77	20 (used function object)
	hierarchical	99		