IWOCL 2024

The 12th International Workshop on OpenCL and SYCL

Evaluation of SYCL's Different Data Parallel Kernels

Marcel Breyer, University of Stuttgart

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

APRIL 8-11, 2024 | CHICAGO, USA | IWOCL.ORG

Motivation - CUDA's kernel invocation type

default

(see https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/)

Motivation - CUDA's kernel invocation type

```
int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];

  for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;
    d[i] = 0;
  }

  int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));

  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  staticReverse<<<1,n>>>(d_d, n);
  cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

default

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

(see https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/)

Motivation - SYCL's kernel invocation types

single task

basic data parallel kernels

work-group parallel kernels

hierarchical parallelism

Motivation - SYCL's kernel invocation types

- single task
- basic data parallel kernels
- scoped parallelism
- work-group parallel kernels
- hierarchical parallelism

AdaptiveCpp

Motivation - SYCL's kernel invocation types

~~single task~~

hierarchical parallelism

basic data parallel kernels

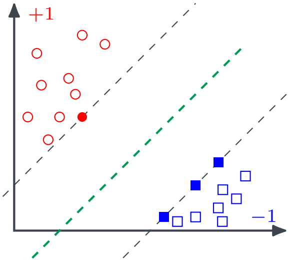
work-group parallel kernels

scoped parallelism

What to know about PLSSVM

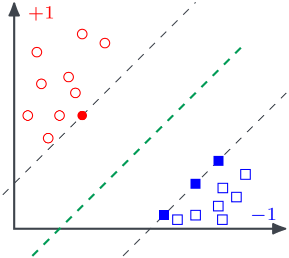
1

Support Vector Machines (SVMs) and PLSSVM



- supervised machine learning technique

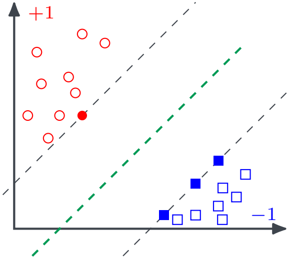
Support Vector Machines (SVMs) and PLSSVM



- supervised machine learning technique
- LS-SVMs as a reformulation of standard SVMs
- solving a system of linear equations

→ massively parallel algorithms known

Support Vector Machines (SVMs) and PLSSVM



- supervised machine learning technique
- LS-SVMs as a reformulation of standard SVMs
- solving a system of linear equations

→ massively parallel algorithms known

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

- explicit and implicit solver
- backends: OpenMP, CUDA, HIP, OpenCL, and SYCL
- multi-class classification via OAA and OAO
- 6 different kernel functions
- multi-GPU support for all kernel functions
- `sklearn.SVC` like Python bindings

The LS-SVM kernel matrix assembly as example application

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

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij}$$

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

The LS-SVM kernel matrix assembly as example application

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

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij}$$

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

$\rightarrow$ we use the Conjugate Gradients algorithm to solve the system of linear equations

The LS-SVM kernel matrix assembly as example application

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

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij}$$

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

→ we use the Conjugate Gradients algorithm to solve the system of linear equations

Kernel function used in our tests: radial basis functions (rbf)

$$k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma \cdot \|\vec{x}_i - \vec{x}_j\|_2^2)$$

The LS-SVM kernel matrix assembly as example application

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

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij}$$

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

$\rightarrow$ we use the Conjugate Gradients algorithm to solve the system of linear equations

Kernel function used in our tests: radial basis functions (rbf)

$$k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma \cdot \|\vec{x}_i - \vec{x}_j\|_2^2)$$

```
for i in num_data_points do
    for j in num_data_points do
        t ← SQUAREDEUCLIDDIST(x_i, x_j)
        Q_ij ← exp(−γ · t)
        if i == j then
            Q_ij ← Q_ij + 1/C
        end if
    end for
end for
```

The LS-SVM kernel matrix assembly as example application

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

$$\boldsymbol{Q}_{ij} = k(\vec{x}_i, \vec{x}_j) + \frac{1}{C} \cdot \delta_{ij}$$

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

$\rightarrow$ we use the Conjugate Gradients algorithm to solve the system of linear equations

Kernel function used in our tests:
radial basis functions (rbf)

$$k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma \cdot \|\vec{x}_i - \vec{x}_j\|_2^2)$$

for i in num_data_points **do**
  for j in num_data_points **do**
    $t \leftarrow \text{SQUAREDEUCLIDDIST}(\vec{x}_i, \vec{x}_j)$
    $\boldsymbol{Q}_{ij} \leftarrow \exp(-\gamma \cdot t)$
    if $i == j$ **then**
      $\boldsymbol{Q}_{ij} \leftarrow \boldsymbol{Q}_{ij} + \frac{1}{C}$
    end if
  end for
end for

`THREAD_BLOCK_SIZE`, `INTERNAL_BLOCK_SIZE`

SYCL's different kernel invocation types

2

Small code examples: reversing the elements in a vector

```
constexpr std::size_t N = 64;
std::vector<int> vec(N);
std::iota(vec.begin(), vec.end(), 0);

// STL
std::reverse(vec.begin(), vec.end());

// manual loop
#pragma omp parallel for
for (std::size_t i = 0; i < N / 2; ++i)
{
    std::swap(vec[i], vec[N - i - 1]);
}
```

Small code examples: reversing the elements in a vector

```cpp
constexpr std::size_t N = 64;
std::vector<int> vec(N);
std::iota(vec.begin(), vec.end(), 0);

// STL
std::reverse(vec.begin(), vec.end());

// manual loop
#pragma omp parallel for
for (std::size_t i = 0; i < N / 2; ++i)
{
    std::swap(vec[i], vec[N - i - 1]);
}
```

```cuda
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

(see https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/)

SYCL's work-group parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    sycl::local_accessor<int> loc{ N, cgh };    // local memory
    cgh.parallel_for(sycl::nd_range<1> exec{ N, N },
                [=](const 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];
    });
});
```

SYCL's work-group parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    sycl::local_accessor<int> loc{ N, cgh };    // local memory
    cgh.parallel_for(sycl::nd_range<1> exec{ N, N },
                [=](const 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];
    });
});
```

SYCL's work-group parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    sycl::local_accessor<int> loc{ N, cgh };     // local memory
    cgh.parallel_for(sycl::nd_range<1> exec{ N, N },
                [=](const 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];
    });
});
```

- **explicitly** have to declare local memory

SYCL's work-group parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    sycl::local_accessor<int> loc{ N, cgh };    // local memory
    cgh.parallel_for(sycl::nd_range<1> exec{ N, N },
                [=](const 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];
    });
});
```

- **explicitly** have to declare local memory
- private memory **implicitly** used in kernels

SYCL's work-group parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    sycl::local_accessor<int> loc{ N, cgh };    // local memory
    cgh.parallel_for(sycl::nd_range<1> exec{ N, N },
                [=](const 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];
    });
});
```

- **explicitly** have to declare local memory
- private memory **implicitly** used in kernels
- **explicit** barriers

Resulting runtimes and profiling

AdaptiveCpp	work-group
A100	16x16, 3x3 3.66 s
MI210	13x13, 3x3 5.08 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s

DPC++	work-group
A100	16x16, 5x5 3.59 s
MI210	16x16, 5x5 3.94 s
2x AMD EPYC 9274F	—

Resulting runtimes and profiling

AdaptiveCpp	work-group
A100	16x16, 3x3 3.66 s
MI210	13x13, 3x3 5.08 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s

DPC++	work-group
A100	16x16, 5x5 3.59 s
MI210	16x16, 5x5 3.94 s
2x AMD EPYC 9274F	—

- NVIDIA A100:
  - 71 % FP64 peak performance

Resulting runtimes and profiling

AdaptiveCpp	work-group
A100	16x16, 3x3 3.66 s
MI210	13x13, 3x3 5.08 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s

DPC++	work-group
A100	16x16, 5x5 3.59 s
MI210	16x16, 5x5 3.94 s
2x AMD EPYC 9274F	—

- NVIDIA A100:
  - 71 % FP64 peak performance
- AMD MI210:
  - AdaptiveCpp vs. DPC++: 19 % better compute unit utilization (up to 100 %)

SYCL's basic data parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        global[idx] = res[idx];
    });
}).wait();
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        const int priv = N - idx - 1;   // private memory
        res[idx] = global[priv];
    });
});
```

SYCL's basic data parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        global[idx] = res[idx];
    });
}).wait();
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        const int priv = N - idx - 1;   // private memory
        res[idx] = global[priv];
    });
});
```

SYCL's basic data parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        global[idx] = res[idx];
    });
}).wait();
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        const int priv = N - idx - 1;   // private memory
        res[idx] = global[priv];
    });
});
```

- **no** local memory available → kernels have to use global memory

SYCL's basic data parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        global[idx] = res[idx];
    });
}).wait();
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        const int priv = N - idx - 1;    // private memory
        res[idx] = global[priv];
    });
});
```

- **no** local memory available → kernels have to use global memory
- private memory **implicitly** used in kernels

SYCL's basic data parallel kernels

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        global[idx] = res[idx];
    });
}).wait();
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(sycl::range{ N }, [=](const sycl::item<1> idx) {
        const int priv = N - idx - 1;   // private memory
        res[idx] = global[priv];
    });
});
```

- **no** local memory available → kernels have to use global memory
- private memory **implicitly** used in kernels
- **no** barriers available → two kernels necessary

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic
NVIDIA A100	16x16, 3x3 3.66 s	16x16, - 14.18 s
AMD MI210	13x13, 3x3 5.08 s	256, - 34.46 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s

DPC++	work-group	basic
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s
MI210	16x16, 5x5 3.94 s	8, - 397.72 s
2x AMD EPYC 9274F	—	

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic
NVIDIA A100	16x16, 3x3 3.66 s	16x16, - 14.18 s
AMD MI210	13x13, 3x3 5.08 s	256, - 34.46 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s

DPC++	work-group	basic
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s
MI210	16x16, 5x5 3.94 s	8, - 397.72 s
2x AMD EPYC 9274F	—	

- NVIDIA A100:
  - only 20 % FP64 peak
  - significant overhead from *global load operations* (LDG)

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic
NVIDIA A100	16x16, 3x3 3.66 s	16x16, - 14.18 s
AMD MI210	13x13, 3x3 5.08 s	256, - 34.46 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s

DPC++	work-group	basic
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s
MI210	16x16, 5x5 3.94 s	8, - 397.72 s
2x AMD EPYC 9274F	—	

- NVIDIA A100:
  - only 20 % FP64 peak
  - significant overhead from *global load operations* (LDG)
- AMD MI210:
  - suboptimal automatic work-group size
  - 3x HBM read requests
  - 11x int32 operations

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic
NVIDIA A100	16x16, 3x3 3.66 s	16x16, - 14.18 s
AMD MI210	13x13, 3x3 5.08 s	256, - 34.46 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s

DPC++	work-group	basic
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s
MI210	16x16, 5x5 3.94 s	8, - 397.72 s
2x AMD EPYC 9274F	—	

- NVIDIA A100:
  - only 20 % FP64 peak
  - significant overhead from *global load operations* (LDG)
- AMD MI210:
  - suboptimal automatic work-group size
  - 3x HBM read requests
  - 11x int32 operations
- 2x AMD EPYC 9274F:
  - no caching → more memory loads

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic
NVIDIA A100	16x16, 3x3 3.66 s	16x16, - 14.18 s
AMD MI210	13x13, 3x3 5.08 s	256, - 34.46 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s

DPC++	work-group	basic
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s
MI210	16x16, 5x5 3.94 s	8, - 397.72 s
2x AMD EPYC 9274F	—	

- NVIDIA A100:
  - only 20 % FP64 peak
  - significant overhead from *global load operations* (LDG)
- AMD MI210:
  - suboptimal automatic work-group size
  - 3x HBM read requests
  - 11x int32 operations
- 2x AMD EPYC 9274F:
  - no caching → more memory loads

Update:
DPC++ AMD GPU: 397.72 s → 45.24 s
8 threads → 64 threads

SYCL's hierarchical parallelism

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for_work_group(sycl::range<1>{ 1 }, sycl::range<1>{ N },
                [=](const sycl::group<1> group) {
        int loc[N];                                           // local memory
        sycl::private_memory<int> priv{ group };    // private memory
        group.parallel_for_work_item([&](sycl::h_item<1> item) {
            const int idx = group[0] * group.get_local_range(0) + 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 = group[0] * group.get_local_range(0) + item.get_local_id(0);
            res[idx] = loc[priv(item)];
        });
    });
});
```

SYCL's hierarchical parallelism

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for_work_group(sycl::range<1>{ 1 }, sycl::range<1>{ N },
                [=](const sycl::group<1> group) {
        int loc[N];                                  // local memory
        sycl::private_memory<int> priv{ group };     // private memory
        group.parallel_for_work_item([&](sycl::h_item<1> item) {
            const int idx = group[0] * group.get_local_range(0) + 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 = group[0] * group.get_local_range(0) + item.get_local_id(0);
            res[idx] = loc[priv(item)];
        });
    });
});
```

SYCL's hierarchical parallelism

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for_work_group(sycl::range<1>{ 1 }, sycl::range<1>{ N },
                [=](const sycl::group<1> group) {
        int loc[N];                                        // local memory
        sycl::private_memory<int> priv{ group };    // private memory
        group.parallel_for_work_item([&](sycl::h_item<1> item) {
            const int idx = group[0] * group.get_local_range(0) + 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 = group[0] * group.get_local_range(0) + item.get_local_id(0);
            res[idx] = loc[priv(item)];
        });
    });
});
```

- local memory **implicitly** used in kernels

SYCL's hierarchical parallelism

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for_work_group(sycl::range<1>{ 1 }, sycl::range<1>{ N },
                [=](const sycl::group<1> group) {
        int loc[N];                                   // local memory
        sycl::private_memory<int> priv{ group };     // private memory
        group.parallel_for_work_item([&](sycl::h_item<1> item) {
            const int idx = group[0] * group.get_local_range(0) + 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 = group[0] * group.get_local_range(0) + item.get_local_id(0);
            res[idx] = loc[priv(item)];
        });
    });
});
```

- local memory **implicitly** used in kernels
- **explicitly** have to declare private memory

SYCL's hierarchical parallelism

```
q.submit([&](sycl::handler &cgh) {
    cgh.parallel_for_work_group(sycl::range<1>{ 1 }, sycl::range<1>{ N },
                        [=](const sycl::group<1> group) {
        int loc[N];                                       // local memory
        sycl::private_memory<int> priv{ group };    // private memory
        group.parallel_for_work_item([&](sycl::h_item<1> item) {
            const int idx = group[0] * group.get_local_range(0) + 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 = group[0] * group.get_local_range(0) + item.get_local_id(0);
            res[idx] = loc[priv(item)];
        });
    });
});
```

- local memory **implicitly** used in kernels
- **explicitly** have to declare private memory
- **implicit** barriers

Hierarchical kernels can result in easier-to-understand code - 1

```
{
    __shared__ double cache[X][Y];
    ...
}
{
    __shared__ double cache[Y][X];
    ...
}
```

Hierarchical kernels can result in easier-to-understand code - 1

```
{
    __shared__ double cache[X][Y];
    ...
}
{
    __shared__ double cache[Y][X];
    ...
}
```

```
__local double cache[X][Y];
...
{
    double (*cache_2)[X] = (double (*)[X]) cache;
    ...
}
```

Hierarchical kernels can result in easier to understand code - 2

work-group

```
sycl::local_accessor<double, 1>
    cache{ sycl::range<1>{ X * Y }, cgh };

void operator()(::sycl::nd_item<2> nd_idx) const {
    ...
    // manually calculate 2D-indices
    cache[x * Y + y] = ...;
    ...
}
```

Hierarchical kernels can result in easier to understand code - 2

work-group

```
sycl::local_accessor<double, 1>
    cache{ sycl::range<1>{ X * Y }, cgh };

void operator()(::sycl::nd_item<2> nd_idx) const {
    ...
    // manually calculate 2D-indices
    cache[x * Y + y] = ...;
    ...
}
```

hierarchical

```
void operator()(::sycl::group<2> group) const {
    {
        double cache[X][Y];
        group.parallel_for_work_item(...);
    }
    {
        double cache[Y][X];
        group.parallel_for_work_item(...);
    }
}
```

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic	hierarchical
A100	16x16, 3x3 3.66 s	16x16, - 14.18 s	16x16, 4x4 3.63 s
MI210	13x13, 3x3 5.08 s	256, - 34.46 s	16x16, 6x6 4.66 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s	12x12, 32x32 27.72 s

DPC++	work-group	basic	hierarchical
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s	16x16, 3x3 4.16 s
MI210	16x16, 5x5 3.94 s	8, - 397.72 s	16x16, 7x7 12.79 s
2x AMD EPYC 9274F		—	

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic	hierarchical
A100	16x16, 3x3 3.66 s	16x16, - 14.18 s	16x16, 4x4 3.63 s
MI210	13x13, 3x3 5.08 s	256, - 34.46 s	16x16, 6x6 4.66 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s	12x12, 32x32 27.72 s

DPC++	work-group	basic	hierarchical
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s	16x16, 3x3 4.16 s
MI210	16x16, 5x5 3.94 s	8, - 397.72 s	16x16, 7x7 12.79 s
2x AMD EPYC 9274F		—	

- NVIDIA A100:
  - 71 % FP64 peak
  - DPC++: 204 % more load and store operations

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic	hierarchical
A100	16x16, 3x3 3.66 s	16x16, - 14.18 s	16x16, 4x4 3.63 s
MI210	13x13, 3x3 5.08 s	256, - 34.46 s	16x16, 6x6 4.66 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s	12x12, 32x32 27.72 s

DPC++	work-group	basic	hierarchical
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s	16x16, 3x3 4.16 s
MI210	16x16, 5x5 3.94 s	8, - 397.72 s	16x16, 7x7 12.79 s
2x AMD EPYC 9274F		—	

- NVIDIA A100:
  - 71 % FP64 peak
  - DPC++: 204 % more load and store operations
- AMD MI210:
  - AdaptiveCpp: only $\frac{1}{3}$ of the LDS instructions compared to work-group parallel

AdaptiveCpp's scoped parallelism

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel(sycl::range<1>{ 1 }, sycl::range<1>{ N }, [=](auto g) {
    sycl::memory_environment(g,
      sycl::require_local_mem<int[N]>(), sycl::require_private_mem<int>(),
      sycl::require_private_mem<int>(),
      [&](auto &loc, auto &idx, auto &priv) {
        sycl::distribute_items_and_wait(g, [&](::sycl::s_item<1> item) {
          idx(item) = g[0] * g.get_logical_local_range(0) + item.get_local_id(g, 0);
          priv(item) = N - idx(item) - 1;
          loc[idx(item)] = res[idx(item)];
        });
        sycl::distribute_items_and_wait(g, [&](::sycl::s_item<1> item) {
          res[idx(item)] = loc[priv(item)];
        });
    });
  });
});
```

AdaptiveCpp's scoped parallelism

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel(sycl::range<1>{ 1 }, sycl::range<1>{ N }, [=](auto g) {
    sycl::memory_environment(g,
      sycl::require_local_mem<int[N]>(), sycl::require_private_mem<int>(),
      sycl::require_private_mem<int>(),
      [&](auto &loc, auto &idx, auto &priv) {
        sycl::distribute_items_and_wait(g, [&](::sycl::s_item<1> item) {
          idx(item) = g[0] * g.get_logical_local_range(0) + item.get_local_id(g, 0);
          priv(item) = N - idx(item) - 1;
          loc[idx(item)] = res[idx(item)];
        });
        sycl::distribute_items_and_wait(g, [&](::sycl::s_item<1> item) {
          res[idx(item)] = loc[priv(item)];
        });
    });
  });
});
```

AdaptiveCpp's scoped parallelism

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel(sycl::range<1>{ 1 }, sycl::range<1>{ N }, [=](auto g) {
    sycl::memory_environment(g,
      sycl::require_local_mem<int[N]>(), sycl::require_private_mem<int>(),
      sycl::require_private_mem<int>(),
      [&](auto &loc, auto &idx, auto &priv) {
        sycl::distribute_items_and_wait(g, [&](::sycl::s_item<1> item) {
          idx(item) = g[0] * g.get_logical_local_range(0) + item.get_local_id(g, 0);
          priv(item) = N - idx(item) - 1;
          loc[idx(item)] = res[idx(item)];
        });
        sycl::distribute_items_and_wait(g, [&](::sycl::s_item<1> item) {
          res[idx(item)] = loc[priv(item)];
        });
    });
  });
});
```

- **explicitly** have to declare local **and** private memory

AdaptiveCpp's scoped parallelism

```
q.submit([&](sycl::handler &cgh) {
  cgh.parallel(sycl::range<1>{ 1 }, sycl::range<1>{ N }, [=](auto g) {
    sycl::memory_environment(g,
      sycl::require_local_mem<int[N]>(), sycl::require_private_mem<int>(),
      sycl::require_private_mem<int>(),
      [&](auto &loc, auto &idx, auto &priv) {
        sycl::distribute_items_and_wait(g, [&](::sycl::s_item<1> item) {
          idx(item) = g[0] * g.get_logical_local_range(0) + item.get_local_id(g, 0);
          priv(item) = N - idx(item) - 1;
          loc[idx(item)] = res[idx(item)];
        });
        sycl::distribute_items_and_wait(g, [&](::sycl::s_item<1> item) {
          res[idx(item)] = loc[priv(item)];
        });
    });
  });
});
```

- **explicitly** have to declare local **and** private memory
- **explicit** barriers

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic	hierarchical	scoped
A100	16x16, 3x3 3.66 s	16x16, - 14.18 s	16x16, 4x4 3.63 s	16x16, 4x4 3.64 s
MI210	13x13, 3x3 5.08 s	256, - 34.46 s	16x16, 6x6 4.66 s	16x16, 6x6 4.63 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s	12x12, 32x32 27.72 s	12x12, 32x32 239.90 s

DPC++	work-group	basic	hierarchical	scoped
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s	16x16, 3x3 4.16 s	—
MI210	16x16, 5x5 3.94 s	8, - 397.72 s	16x16, 7x7 12.79 s	—
2x AMD EPYC 9274F	—			

Resulting runtimes and profiling

AdaptiveCpp	work-group	basic	hierarchical	scoped
A100	16x16, 3x3 3.66 s	16x16, - 14.18 s	16x16, 4x4 3.63 s	16x16, 4x4 3.64 s
MI210	13x13, 3x3 5.08 s	256, - 34.46 s	16x16, 6x6 4.66 s	16x16, 6x6 4.63 s
2x AMD EPYC 9274F	12x12, 32x32 23.81 s	??, - 1782.45 s	12x12, 32x32 27.72 s	12x12, 32x32 239.90 s

DPC++	work-group	basic	hierarchical	scoped
A100	16x16, 5x5 3.59 s	1x768, - 14.32 s	16x16, 3x3 4.16 s	—
MI210	16x16, 5x5 3.94 s	8, - 397.72 s	16x16, 7x7 12.79 s	—
2x AMD EPYC 9274F	—			

- 2x AMD EPYC 9274F:
  - 11.8x wait times compared to work-group parallel
  - missed vectorization opportunities

Summary

AdaptiveCpp	work-group	basic	hierarchical	scoped
A100	3.66 s	14.18 s	3.63 s	3.64 s
MI210	5.08 s	34.46 s	4.66 s	4.63 s
2x AMD EPYC 9274F	23.81 s	1782.45 s	27.72 s	239.90 s

DPC++	work-group	basic	hierarchical
A100	3.59 s	14.32 s	4.16 s
MI210	3.94 s	397.72 s	12.79 s
2x AMD EPYC 9274F		—	

Summary

AdaptiveCpp	work-group	basic	hierarchical	scoped
A100	3.66 s	14.18 s	3.63 s	3.64 s
MI210	5.08 s	34.46 s	4.66 s	4.63 s
2x AMD EPYC 9274F	23.81 s	1782.45 s	27.72 s	239.90 s

DPC++	work-group	basic	hierarchical
A100	3.59 s	14.32 s	4.16 s
MI210	3.94 s	397.72 s	12.79 s
2x AMD EPYC 9274F		—	

Summary

AdaptiveCpp	work-group	basic	hierarchical	scoped
A100	3.66 s	14.18 s	3.63 s	3.64 s
MI210	5.08 s	34.46 s	4.66 s	4.63 s
2x AMD EPYC 9274F	23.81 s	1782.45 s	27.72 s	239.90 s

DPC++	work-group	basic	hierarchical
A100	3.59 s	14.32 s	4.16 s
MI210	3.94 s	397.72 s	12.79 s
2x AMD EPYC 9274F		—	

Summary

AdaptiveCpp	work-group	basic	hierarchical	scoped
A100	3.66 s	14.18 s	3.63 s	3.64 s
MI210	5.08 s	34.46 s	4.66 s	4.63 s
2x AMD EPYC 9274F	23.81 s	1782.45 s	27.72 s	239.90 s

DPC++	work-group	basic	hierarchical
A100	3.59 s	14.32 s	4.16 s
MI210	3.94 s	397.72 s	12.79 s
2x AMD EPYC 9274F		—	

Summary

AdaptiveCpp	work-group	basic	hierarchical	scoped
A100	3.66 s	14.18 s	3.63 s	3.64 s
MI210	5.08 s	34.46 s	4.66 s	4.63 s
2x AMD EPYC 9274F	23.81 s	1782.45 s	27.72 s	239.90 s

DPC++	work-group	basic	hierarchical
A100	3.59 s	14.32 s	4.16 s
MI210	3.94 s	397.72 s	12.79 s
2x AMD EPYC 9274F		—	

Summary

AdaptiveCpp	work-group	basic	hierarchical	scoped
A100	3.66 s	14.18 s	**3.63 s**	3.64 s
MI210	5.08 s	34.46 s	4.66 s	**4.63 s**
2x AMD EPYC 9274F	**23.81 s**	1782.45 s	27.72 s	239.90 s

DPC++	work-group	basic	hierarchical
A100	**3.59 s**	14.32 s	4.16 s
MI210	**3.94 s**	397.72 s	12.79 s
2x AMD EPYC 9274F		—	

Conclusion

- **basic data parallel kernel**
  - useful for rapid prototyping
  - performance-wise too bad for performance-critical code

Conclusion

- **basic data parallel kernel**
  - useful for rapid prototyping
  - performance-wise too bad for performance-critical code
- **work-group parallel kernel**
  - easy to use due to its similarity to CUDA, HIP, OpenCL, etc.
  - best overall performance

Conclusion

- **basic data parallel kernel**
  - useful for rapid prototyping
  - performance-wise too bad for performance-critical code
- **work-group parallel kernel**
  - easy to use due to its similarity to CUDA, HIP, OpenCL, etc.
  - best overall performance
- **hierarchical parallelism** (*"deprecated"*)
  - performance close to or a bit slower than work-group parallel kernels
  - kernel formulation more suitable for some problems than work-group parallel kernels

Conclusion

- **basic data parallel kernel**
  - useful for rapid prototyping
  - performance-wise too bad for performance-critical code
- **work-group parallel kernel**
  - easy to use due to its similarity to CUDA, HIP, OpenCL, etc.
  - best overall performance
- **hierarchical parallelism** (*"deprecated"*)
  - performance close to or a bit slower than work-group parallel kernels
  - kernel formulation more suitable for some problems than work-group parallel kernels
- **scoped parallelism**
  - performance close to hierarchical parallelism on GPUs
  - most explicit kernels

Conclusion

- **basic data parallel kernel**
  - useful for rapid prototyping
  - performance-wise too bad for performance-critical code
- **work-group parallel kernel**
  - easy to use due to its similarity to CUDA, HIP, OpenCL, etc.
  - best overall performance
- **hierarchical parallelism** (*"deprecated"*)
  - performance close to or a bit slower than work-group parallel kernels
  - kernel formulation more suitable for some problems than work-group parallel kernels
- **scoped parallelism**
  - performance close to hierarchical parallelism on GPUs
  - most explicit kernels

It's nice to be able to choose between different kernel formulations!

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.

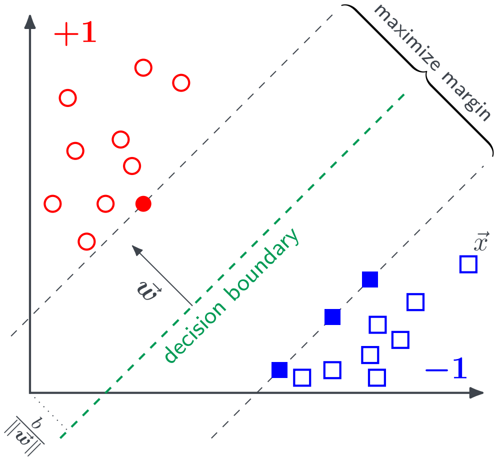
[3] Alexander Van Craen, Marcel Breyer, and Dirk Pflüger. “PLSSVM—Parallel Least Squares Support Vector Machine”. In: *Software Impacts* 14 (2022), p. 100343. ISSN: 2665-9638. DOI: https://doi.org/10.1016/j.simpa.2022.100343. URL: https://www.sciencedirect.com/science/article/pii/S2665963822000641.

[4] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. “Performance Evolution of Different SYCL Implementations Based on the Parallel Least Squares Support Vector Machine Library”. In: *Proceedings of the 2023 International Workshop on OpenCL*. IWOCL '23. Cambridge, United Kingdom: Association for Computing Machinery, 2023. DOI: 10.1145/3585341.3585369. URL: https://doi.org/10.1145/3585341.3585369.

Addi-
tional
resources

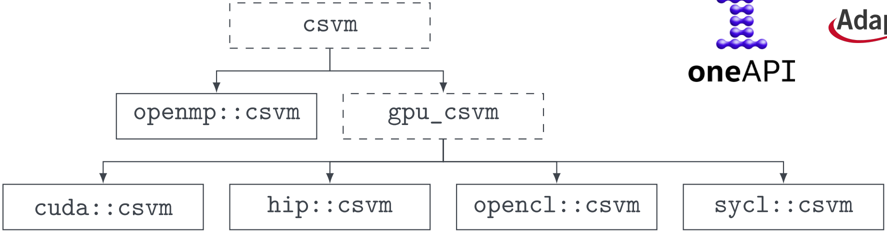
Basics of Support Vector Machines (SVMs) (proposed by Boser, Guyon, and Vapnik in 1992)

supervised machine learning: example for binary classification



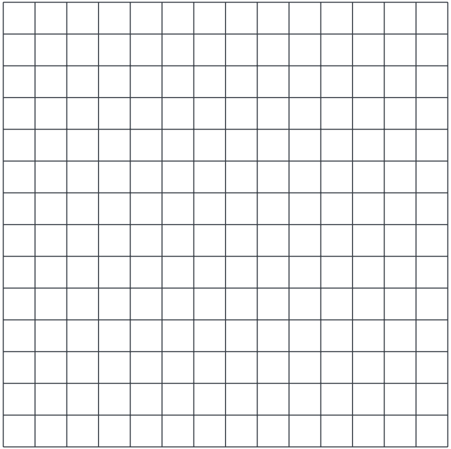
$$y = \operatorname{sgn}\left(\langle \vec{\boldsymbol{w}}, \vec{x} \rangle + b\right)$$

PLSSVM supports many different backends



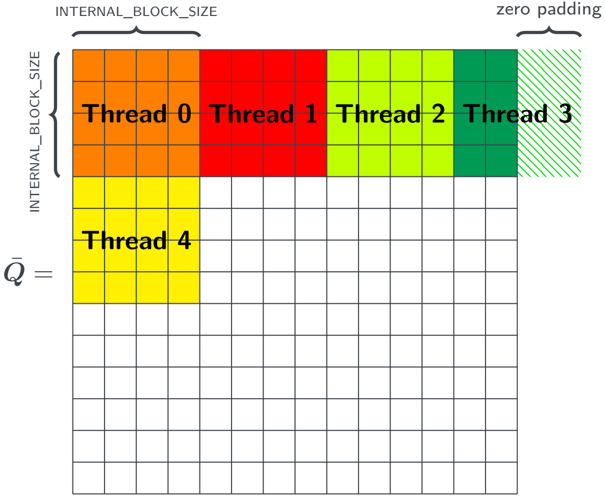
Basic idea of the used blocking scheme

$\bar{Q} =$



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

Used software, hardware, and data set

Source: www.nvidia.com

NVIDIA A100
CUDA 12.2.2
Driver Version 535.129.03

Source: www.amd.com

AMD Instinct MI210
HIP/ROCm 5.7.0
Driver Version 3590.0 (HSA1.1,LC)

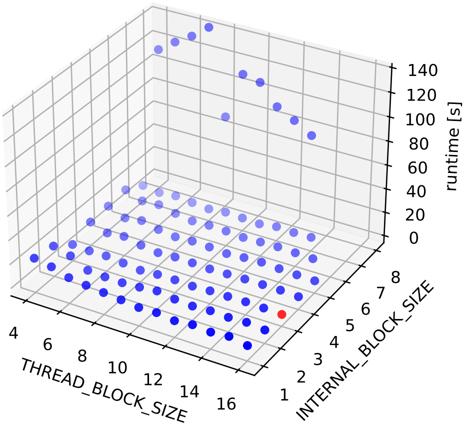
Source: www.intel.com

2x AMD EPYC 9274F

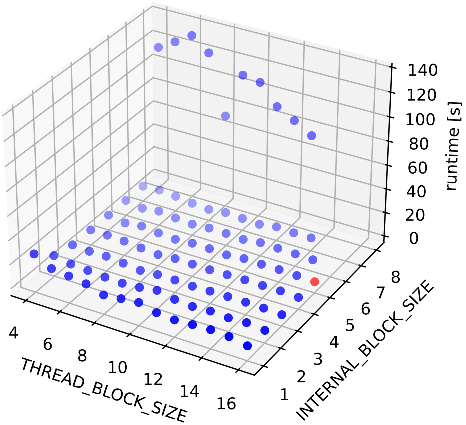
- DPC++ (*OpenSource LLVM fork*): nightly-2023-12-01
- AdaptiveCpp (*OpenSource*): v23.10.0

Street View House Numbers (SVHN) data set: $73\,257 \times 3072$ (RGB images of size 32×32)

work-group parallel kernels hyper-Parameter Tuning on the A100



AdaptiveCpp



DPC++