Using SYCL Joint Matrix Extension for Fast and Portable Matrix Operations

James Brodman, Intel Corp.
Dounia Khaldi*, Bing Yu*, Dmitry Sidorov*, Mateusz Belicki*, Yury Plyakhin*
*Intel Corp.
Introduction

• Programming abstractions for matrix computing: tradeoff between increasing level of abstraction and programmer control

• Deliver unified SYCL matrix interface across matrix hardware: Intel AMX, Intel XMX, Nvidia Tensor Cores, etc.
  • Programmer productivity: Allow the customer to express their applications for matrix hardware with minimal changes
  • Performance: Maps directly to low-level intrinsics/assembly for maximum performance

• Status
  • Implementation: Unified interface is part of oneAPI releases (starting 2023.1)
  • Performance kernels with special tuning (tiling factors, matrix size) for different devices
Outline

• Examples of Matrix Hardware: Intel AMX and Intel XMX
• SYCL Joint Matrix Extension
• Matrix Query Interface
• Tuning for Performance
• Conclusion and Next Steps
Matrix Hardware
Intel XMX in Intel® Data Center GPU Max Series

- Code-named Ponte Vecchio (PVC)
- Xe-HPC 2-Stack Ponte Vecchio GPU
- Each Xe-Stack has 4 slices
- Xe-slice contains 16 Xe-core
- An Xe-core contains 8 vector and 8 matrix engines
Intel AMX High-Level Architecture

- Intel® Xeon® processor codenamed Sapphire Rapids
- Intel AMX, an Intel x86 extension for multiplication of matrices of bf16/int8 elements
SYCL Joint Matrix Extension
<table>
<thead>
<tr>
<th>SYCL joint_matrix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace sycl::ext::oneapi::experimental::matrix</td>
<td>Namespace</td>
</tr>
<tr>
<td>template &lt;typename Group, typename T, use Use, size_t Rows, size_t Cols, layout Layout = layout::dynamic&gt;</td>
<td>- New matrix data type with group scope</td>
</tr>
<tr>
<td>struct joint_matrix;</td>
<td>- Defined with a specified type, use (a, b, accumulator), shape, and layout</td>
</tr>
<tr>
<td>enum class use { a, b, accumulator};</td>
<td></td>
</tr>
<tr>
<td>enum class layout {row_major, col_major, dynamic};</td>
<td></td>
</tr>
<tr>
<td>joint_matrix_fill( Group g, joint_matrix&lt;&amp;&gt; dst, T v);</td>
<td>- Separate memory operations from the compute</td>
</tr>
<tr>
<td>void joint_matrix_load( Group g, joint_matrix&lt;&amp;&gt; dst, multi_ptr&lt;&gt; src, size_t stride, Layout layout); // use::accumulator matrix void joint_matrix_load( Group g, joint_matrix&lt;&amp;&gt; dst, multi_ptr&lt;&gt; src, size_t stride); // use::a and use::b matrices void joint_matrix_store( Group g, joint_matrix&lt;&amp;&gt; src, multi_ptr&lt;&gt; dst, unsigned stride, Layout layout); //use::accumulator matrix</td>
<td>- Group execution scope ➔ joint, Group as argument</td>
</tr>
<tr>
<td>• joint_matrix_mad( Group g, joint_matrix&lt;&amp;&gt;&amp;D, joint_matrix&lt;&amp;&gt;&amp;A, joint_matrix&lt;&amp;&gt;&amp;B, joint_matrix&lt;&amp;&gt;&amp;C);</td>
<td>- Multiply and add</td>
</tr>
<tr>
<td>• void joint_matrix_apply( Group g, joint_matrix&lt;&amp;&gt;&amp;A, F&amp;&amp; func);</td>
<td>- Element-wise ops</td>
</tr>
<tr>
<td>• void joint_matrix_copy( Group g, joint_matrix&lt;Group, T1, Use1, Rows, Cols, Layout1&gt; &amp;dest, joint_matrix&lt;Group, T2, Use2, Rows, Cols, Layout2&gt; &amp;src);</td>
<td>- Copy</td>
</tr>
</tbody>
</table>
using namespace sycl::ext::oneapi::experimental::matrix;

queue q;
range<2> G = {M/tM, N/tN * SG_SIZE};
range<2> L = {1, SG_SIZE};
bfloat16 *memA = malloc_shared<bfloat16>(M*K, q);
bfloat16 *memB = malloc_shared<bfloat16>(K*N, q);
float *memC = malloc_shared<float>(M*N, q);

q.submit([&](sycl::handler& cgh) {
  auto pA = address_space_cast<sycl::access::address_space::global_space,
                              sycl::access::decorated::no>(memA);
  auto pB = address_space_cast<sycl::access::address_space::global_space,
                              sycl::access::decorated::no>(memB);
  auto pC = address_space_cast<sycl::access::address_space::global_space,
                              sycl::access::decorated::no>(memC);

cgh.parallel_for(nd_range<2>(G, L), [=](nd_item<2> item) {
  const auto sg_startx = item.get_global_id(0) - item.get_local_id(0);
  const auto sg_starty = item.get_global_id(1) - item.get_local_id(1);
  sub_group sg = item.get_sub_group();
  joint_matrix<sub_group, bfloat16, use::a, tM, tK, layout::row_major> subA;
  joint_matrix<sub_group, bfloat16, use::b, tK, tN, layout::row_major> subB;
  joint_matrix<sub_group, float, use::accumulator, tM, tN> subC;
  joint_matrix_fill(sg, subC, 0);
  for (int k = 0; k < K; k += tk) {
    joint_matrix_load(sg, subA, pA + sg_startx * tM * K + k, K);
    joint_matrix_load(sg, subB, pB + k * N + sg_starty, N);
    joint_matrix_mad(sg, subC, subA, subB, subC);
  }
  joint_matrix_apply(sg, subC, [=](bfloat16 &x) { x = Relu(x); });
  joint_matrix_store(sg, subC, pC + sg_startx * tM * N + sg_starty, N, row_major);
});
});
q.wait;
Since oneAPI 2023.1: One Joint Matrix Code to Run on Intel AMX, Intel XMX and Nvidia* Tensor Cores

```cpp
joint_matrix<sub_group, Ta, use::a, tM, tK, layout::row_major> subA;
joint_matrix<sub_group, Tb, use::b, tK, tN, layout::row_major> subB;
joint_matrix<sub_group, Tc, use::accumulator, tM, tN> subC;
sub_group sg = item.get_sub_group();
joint_matrix_fill(sg, subC, 0);
for (int k = 0; k < K; k += tK) {
    joint_matrix_load(sg, subA, accA.get_pointer() + sg_startx * tM * K + k, K);
    joint_matrix_load(sg, subB, accB.get_pointer() + k * N + sg_starty/SG_SIZE*tN, N);
    joint_matrix_mad(sg, subC, subA, subB, subC);
}
joint_matrix_apply(sg, subC, [](T x) { x *= alpha; });
joint_matrix_store(sg, subC, accC.get_pointer() + sg_startx * tM * N + sg_starty/SG_SIZE*tN, N, layout::row_major);
```

Parameters are not portable → use query
SYCL joint_matrix

// inputA is MxK, inputB is KxN, inputC is MxN
#define tM=16  tN=16  tK=16

void gemm(size_t global_idx, size_t global_idy, size_t local_idx, size_t local_idy, sub_group sg) {
    joint_matrix<sub_group, half, use::a, tM, tK, row_major> matA;
    joint_matrix<sub_group, half, use::b, tK, tN, row_major> matB;
    joint_matrix<float, use::accumulator, tM, tN> matC;

    const auto sg_startx = global_idx - local_idx;
    const auto sg_starty = global_idy - local_idy;

    joint_matrix_fill(matC, 0.0f);
    for (int step = 0; step < tK; step += tK) {
        uint AStart = sg_startx * tM * K + step;
        uint BStart = step * N + sg_starty;
        joint_matrix_load(sg, matA, inputA + AStart, K);
        joint_matrix_load(sg, matB, inputB + BStart, N);
        joint_matrix_mad(sg, matC, matA, matB, matC);
    }

    joint_matrix_apply(sg, matC, [=](T& x) { x *= alpha; });
    joint_matrix_store(sg, matC, output + sg_startx * tM * N + sg_starty, N, row_major);
}

CUDA Fragments

// inputA is MxK, inputB is KxN, inputC is MxN
#define tM=16  tN=16  tK=16

__global__ void wmma_ker(blockidx) {

    joint_matrix<sub_group, half, use::a, tM, tK, row_major> matA;
    joint_matrix<sub_group, half, use::b, tK, tN, row_major> matB;
    joint_matrix<float, use::accumulator, tM, tN> matC;

    uint row = (blockIdx%x - 1)*tM + 1;
    uint col = (blockIdx%y - 1)*tN + 1;
    fill_fragment(matC, 0.0f);
    for (int step = 0; step < tK; step += tK) {
        uint AStart = row * rowStrideA + step;
        uint BStart = col * colStrideB + step;
        load_matrix_sync(matA, inputA + AStart, K);
        load_matrix_sync(matB, inputB + BStart, N);
        mma_sync(matC, matA, matB, matC);
    }

    joint_matrix_apply(sg, matC, [=](T& x) { x *= alpha; });
    store_matrix_sync(inputC + row*N + col, matC, N, mem_row_major);
}
SYCL Matrix Extension: Intel Specific Features
SYCL Matrix Extension: Intel Specific Features
SYCL joint_matrix Indexing with Coordinates

• Element wise ops that apply to a set of elements of the matrix → Mapping is required
• Example: Quantization Calculations
• $A\times B + \text{sum}_{\text{rows}}A + \text{sum}_{\text{cols}}B + \text{scalar}_{\text{zero\_point}}$
• $\text{sum}_{\text{rows}}A$ returns a single row of $A$

```
using namespace sycl::ext::intel::experimental::matrix;

void sum_rows_A(joint_matrix<T, rows, cols>& subA)
{
    joint_matrix_apply(sg, subA, [=](T &val, size_t row, size_t col) {
        global_row = row + global_idx * rows;
        sum_local_rows[global_row] += val;
    });
}
```
Matrix Query Interface
# AMX Supported Combinations

<table>
<thead>
<tr>
<th>A type</th>
<th>B type</th>
<th>C type</th>
<th>M</th>
<th>N</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u)int8_t</td>
<td>(u)int8_t</td>
<td>int32_t</td>
<td>&lt;=16</td>
<td>&lt;=16</td>
<td>&lt;=64</td>
</tr>
<tr>
<td>bf16</td>
<td>bf16</td>
<td>float</td>
<td>&lt;=16</td>
<td>&lt;=16</td>
<td>&lt;=32</td>
</tr>
</tbody>
</table>
## Intel XMX Supported Combinations

<table>
<thead>
<tr>
<th>A type</th>
<th>B type</th>
<th>C type</th>
<th>M</th>
<th>N</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u)int8_t</td>
<td>(u)int8_t</td>
<td>int32_t</td>
<td>&lt;=8</td>
<td>8 (DG2)</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 (PVC)</td>
<td></td>
</tr>
<tr>
<td>fp16</td>
<td>fp16</td>
<td>float</td>
<td>&lt;=8</td>
<td>8 (DG2)</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 (PVC)</td>
<td></td>
</tr>
<tr>
<td>bf16</td>
<td>bf16</td>
<td>float</td>
<td>&lt;=8</td>
<td>8 (DG2)</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16 (PVC)</td>
<td></td>
</tr>
<tr>
<td>tf32</td>
<td>tf32</td>
<td>float</td>
<td>&lt;=8</td>
<td>16 (PVC)</td>
<td></td>
</tr>
</tbody>
</table>
## Nvidia* Tensor Cores Supported Combinations

<table>
<thead>
<tr>
<th>A type</th>
<th>B type</th>
<th>Accumulator type</th>
<th>M</th>
<th>N</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>half</td>
<td>float</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>half</td>
<td>half</td>
<td>half</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>bfloat16</td>
<td>bfloat16</td>
<td>float</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>tf32</td>
<td>tf32</td>
<td>float</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>(u)int8_t</td>
<td>(u)int8_t</td>
<td>int32_t</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>
Matrix Runtime Query

- Tell the set of supported matrix sizes and types on this device via an extended device information descriptor.

```c
struct combination {
    size_t max_msize;
    size_t max_nsize;
    size_t max_ksize;
    size_t msize;
    size_t nsize;
    size_t ksize;
    matrix_type atype;
    matrix_type btype;
    matrix_type ctype;
    matrix_type dtype;
};
```

```c
using namespace sycl::ext::oneapi::experimental::info::device;
std::vector<combination> combinations =
device.get_info<info::device::matrix_combinations>();
for (int i = 0; sizeof(combinations); i++) {
    if (Ta == combinations[i].atype && Tb == combinations[i].btype &&
        Tc == combinations[i].ctype && Td == combinations[i].dtype) {
        // joint matrix GEMM kernel can be called using these sizes
        joint_matrix_gemm(combinations[i].msize,
                          combinations[i].nsize, combinations[i].ksize);
    }
}
```
Tuning for Performance
Example of Performance Kernel
M and N are parallel, k sequential in the kernel

range<2> global{M / MSG, (N / NSG) * SG_SIZE};
range<2> local{MCACHE / MSG, NCACHE / NSG * SG_SIZE};
cgh.parallel_for(nd_range<2>(G, L), [=](nd_item<2> item) {
    for (unsigned int kcache = 0; kcache < K; kcache += KCACHE) {
        for (unsigned int ksg = 0; ksg < KCACHE; ksg += KSG) {
            // load from cache/local memory to registers
            // Joint matrix available sizes (MJM, NJM, KJM) reported by the query
            for (unsigned int mjm = 0; mjm < MSG; msg += MJM) {
                for (unsigned int njm = 0; njm < NSG; nsg += NJM) {
                    for (unsigned int kjm = 0; kjm < KSG; kjm += KJM) {
                        joint_matrix_mad(...);
                    }
                }
            }
        }
    }
}

//Workgroup and cache locality
for (unsigned int mcache = 0; mcache < M; mcache += MCACHE) {
    for (unsigned int ncache = 0; ncache < N; ncache += NCACHE) {
        for (unsigned int kcache = 0; kcache < K; kcache += KCACHE) {
            // load/prefetch from global memory to cache/local memory
            //Sub-group and registers locality
            for (unsigned int msg = 0; msg < MCACHE; msg += MSG) {
                for (unsigned int nsg = 0; nsg < NCACHE; nsg += NSG) {
                    for (unsigned int ksg = 0; ksg < KCACHE; ksg += KSG) {
                        // load from cache/local memory to registers
                        // Joint matrix available sizes (MJM, NJM, KJM) reported by the query
                        for (unsigned int mjm = 0; mjm < MSG; msg += MJM) {
                            for (unsigned int njm = 0; njm < NSG; nsg += NJM) {
                                for (unsigned int kjm = 0; kjm < KSG; kjm += KJM) {
                                    joint_matrix_mad(...);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

// load from cache/local memory to registers
// Joint matrix available sizes (MJM, NJM, KJM) reported by the query
for (unsigned int mjm = 0; mjm < MSG; msg += MJM) {
    for (unsigned int njm = 0; njm < NSG; nsg += NJM) {
        for (unsigned int kjm = 0; kjm < KSG; kjm += KJM) {
            joint_matrix_mad(...);
        }
    }
}
<table>
<thead>
<tr>
<th>Bfloat16 type example</th>
<th>SYCL terms</th>
<th>Compute unit and data</th>
<th>Intel XMX</th>
<th>Intel AMX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workgroup execution hierarchy level/ Cache locality</td>
<td>Local range</td>
<td>• Data in L1 cache</td>
<td>(MCACHExNCACHExKCACHE) =256x256x32 for 4096 GEMM size</td>
<td>(MCACHExNCACHExKCACHE) = 256x256x1024 for 4096 GEMM size</td>
</tr>
<tr>
<td>Subgroup execution hierarchy level</td>
<td>Inside the kernel</td>
<td>• Data in registers</td>
<td>MSGxNSGxKSG = 32x64x16</td>
<td>MSGxNSGxKSG = 32x32x32 to occupy the 8 AMX tiles</td>
</tr>
<tr>
<td>Joint matrix shape</td>
<td>Inside the kernel</td>
<td>Matrix hardware</td>
<td>MJMxNJMxKJM = 8x16x16</td>
<td>MJMxNJMxKJM = 16x16x32</td>
</tr>
<tr>
<td>Additional runtime options</td>
<td></td>
<td></td>
<td>SYCL_PROGRAM_COMPILE_OPTIONS=&quot;-ze-opt-large-register-file&quot;</td>
<td>• DPCPP_CPU_NUM_CUS to set number of threads • DPCPP_CPU_PLACES for threads to core affinity</td>
</tr>
</tbody>
</table>
Conclusion and Next Steps

• Full support of SYCL joint matrix extension on Intel AMX, Intel XMX, Nvidia Tensor Cores, and AMD Matrix Cores
• Matrix extensions to LLVM IR and SPIRV
• Effective usage in MLIR integration and CUDA code migration
• Next steps:
  • Standardization of SYCL joint matrix to Khronos SYCL
  • Performance results to be published, more tuning to be done
  • More features to come: out of bounds checking, joint_matrix_prefetch, etc
• Contact: dounia.khaldi@intel.com
Legal Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.
No product or component can be absolutely secure.
Your costs and results may vary.
© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
Backup Slides
SYCL Matrix Extension: Intel Specific Features
SYCL joint_matrix Indexing with Coordinates

- Element wise ops that apply to a set of elements of the matrix → Mapping is required
- Example: Quantization Calculations
- $A \times B + \text{sum\_rows\_A} + \text{sum\_cols\_B} + \text{scalar\_zero\_point}$
- $\text{sum\_rows\_A}$ returns a single row of $A$

```cpp
using namespace sycl::ext::oneapi::experimental::matrix;

void sum_rows_A(joint_matrix<T, rows, cols>& subA) {
    auto data = ext::intel::experimental::matrix::get_wi_data(sg, subA);
    for (int i = 0; i < data.length(); ++i) {
        auto [row, col] = data[i].get_coord();
        global_index = row + global_idx * rows;
        sum_local_rows[global_index] += data[i];
    }
}
```

```cpp
using namespace sycl::ext::oneapi::experimental::matrix;

void sum_rows_A(joint_matrix<T, rows, cols>& subA) {
    joint_matrix_apply(sg, subA, [=](T &val, size_t row, size_t col) {
        global_row = row + global_idx * rows;
        sum_local_rows[global_row] += val;
    });
}
```
# Current Users of Joint Matrix

<table>
<thead>
<tr>
<th>Code migration from wmma samples</th>
<th>Code migration from wmma samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>- <a href="https://github.com/NVIDIA/cuda_s...samples/tree/master/Samples/3_CUDA_Features/cudaTensorCoreGemm">https://github.com/NVIDIA/cuda_s...samples/tree/master/Samples/3_CUDA_Features/cudaTensorCoreGemm</a></td>
<td>- <a href="https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/cudaTensorCoreGemm">https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/cudaTensorCoreGemm</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Porting code from wmma to joint_matrix</th>
<th>Porting code from wmma to joint_matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Porting an earthquake simulation code that makes direct use of the tensor cores through wmma from CUDA and wmma to SYCL and joint matrix</td>
<td>- Porting an earthquake simulation code that makes direct use of the tensor cores through wmma from CUDA and wmma to SYCL and joint matrix</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SYCL-DNN – By CodePlay</th>
<th>SYCL-DNN – By CodePlay</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Using joint_matrix for enabling Nvidia Tensor Cores in SYCL-DNN</td>
<td>- Using joint_matrix for enabling Nvidia Tensor Cores in SYCL-DNN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SYCL-BLAS – By CodePlay</th>
<th>SYCL-BLAS – By CodePlay</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Using joint_matrix for enabling Nvidia Tensor Cores in SYCL-BLAS GEMM</td>
<td>- Using joint_matrix for enabling Nvidia Tensor Cores in SYCL-BLAS GEMM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SPIRV MLIR Dialect</th>
<th>SPIRV MLIR Dialect</th>
</tr>
</thead>
<tbody>
<tr>
<td>- XMX Support using MLIR SPIRV dialect by adding SPIRV joint_matrix</td>
<td>- XMX Support using MLIR SPIRV dialect by adding SPIRV joint_matrix</td>
</tr>
</tbody>
</table>