Joint Matrix: A Unified SYCL Extension for Matrix Hardware Programming

Dounia Khaldi

Intel Corp.

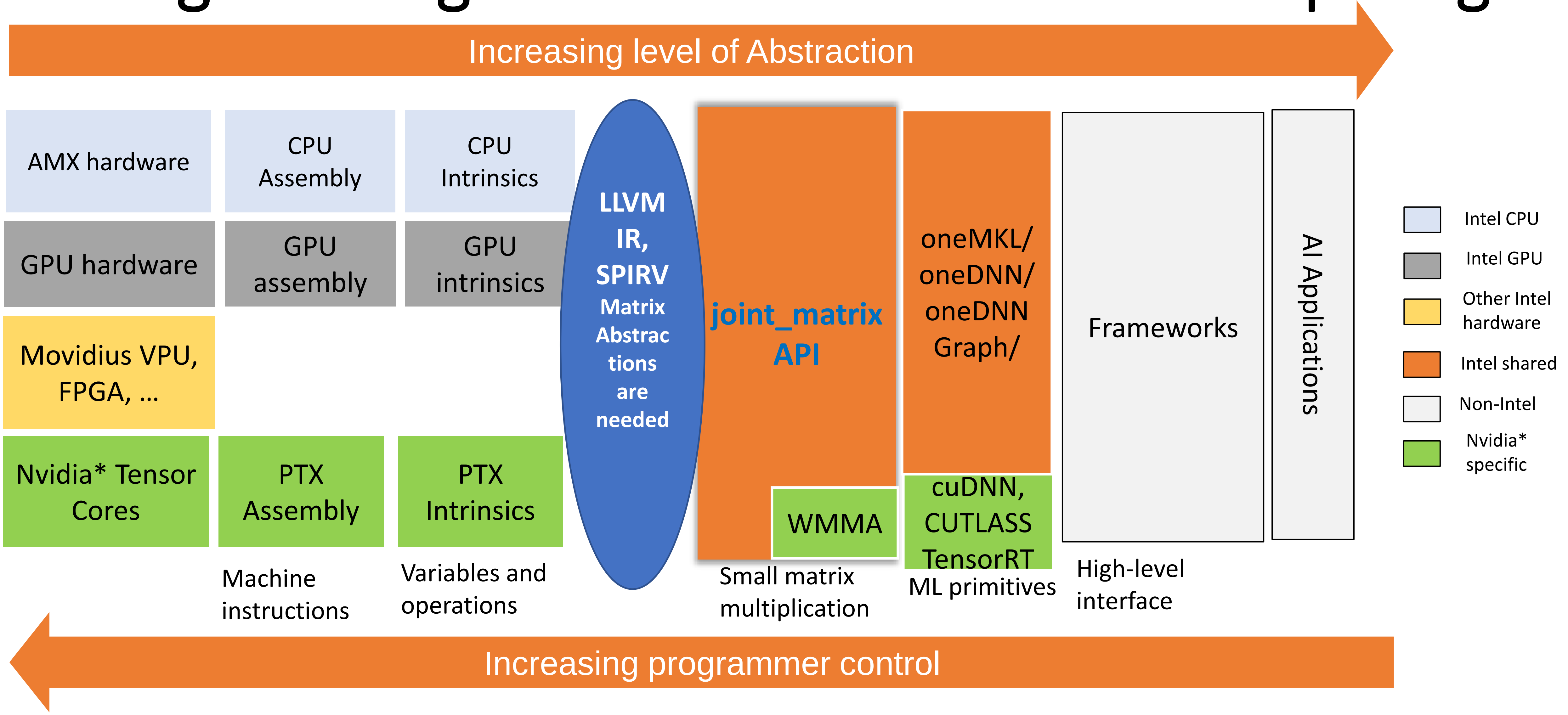
oneAPI Language SIG Forum, June 07th, 2023

Collaborating with many colleagues from Intel and Codeplay

Executive Summary

- Goal
  - Deliver unified SYCL matrix interface across matrix hardware: Intel AMX, Intel XMX and Nvidia Tensor Cores
    - 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 2023.1 release
  - Current Users: Code porting from CUDA wmma, MLIR SPIRV-based joint matrix code generation

Programming Abstractions for Matrix Computing



Ninja programmer - Focus on performance through hardware

Application programmer - Focus on algorithmic improvements

oneAPI

Lead Users

Performance portability across all hardware without extra effort of optimizations for specific hardware

AI Scientists

- New operations such as tensor contractions, BRGEMM, quantized gemm, fused operations

Library Developers

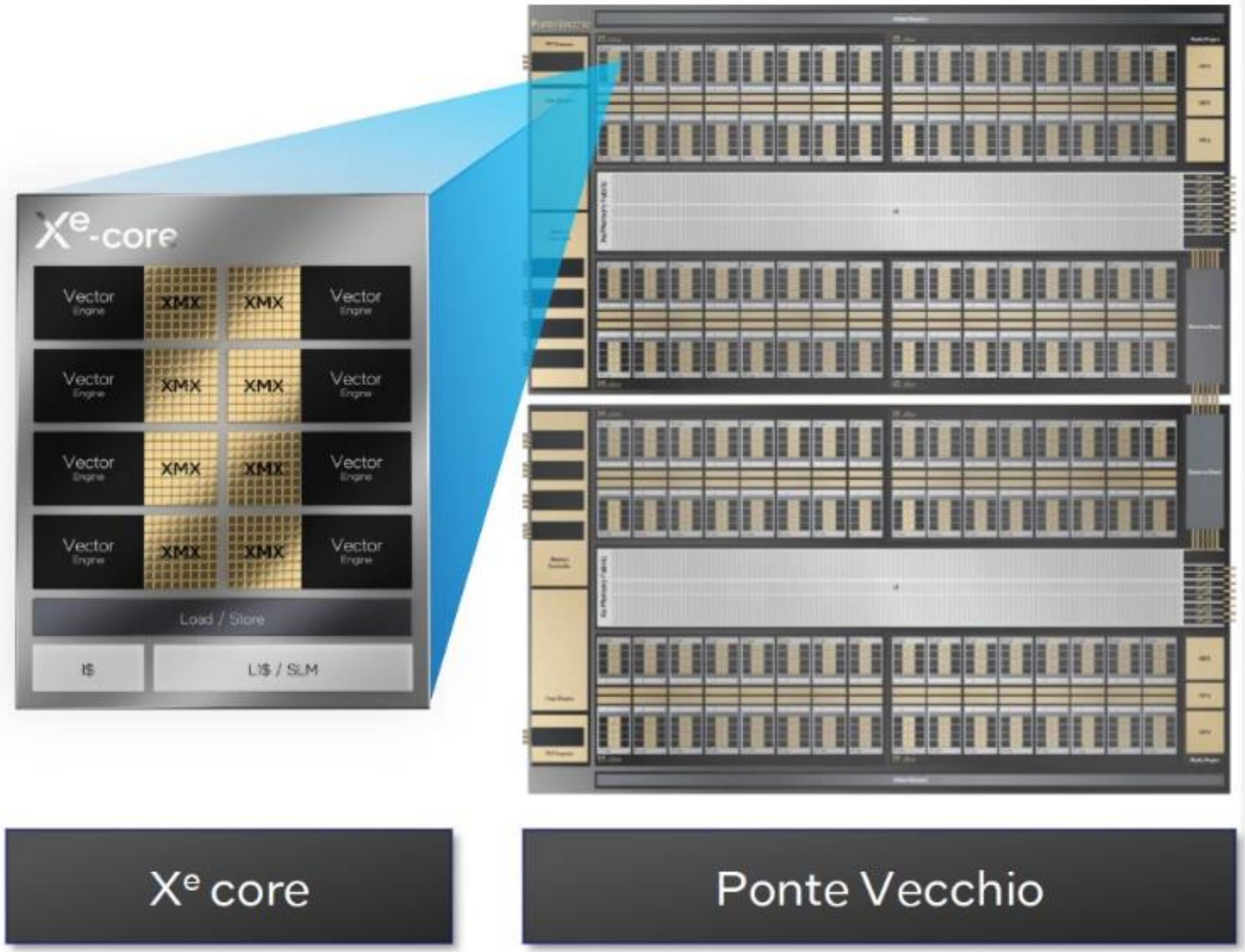
- Different DNN and BLAS libraries

Matrix Hardware

oneAPI

Intel XMX in Intel® Data Center GPU Max Series

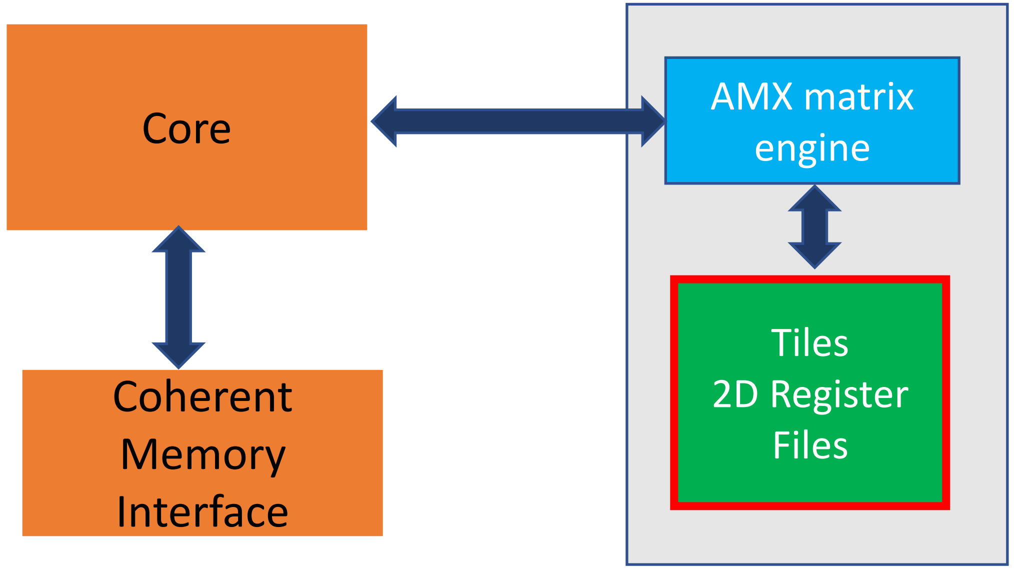
- Code-named Ponte Vecchio (PVC)
- Xe-HPC 2-Stack Ponte Vecchio GPU
- Each Xe-Stack has 4 slices
- Xe-slice contains 16 Xe-core
- An Xe -core contains 8 vector and 8 matrix engines



oneAPI

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

SYCL Matrix Extension

Namespace	namespace sycl::ext::oneapi::experimental::matrix
- New matrix data type with group scope - Defined with a specified type, use (a, b, accumulator), size, and layout	template <typename Group, typename T, use Use, size_t Rows, size_t Cols, layout Layout = layout::dynamic> struct joint_matrix; enum class use { a, b, accumulator};
- Separate memory operations from the compute - enum class layout {row_major, col_major, dynamic }; - Group execution scope → *joint, Group as argument*	• joint_matrix_fill(Group g, joint_matrix<>&dst, T v); • void joint_matrix_load(Group g, joint_matrix<>dst, T *base, unsigned stride, Layout layout); • void joint_matrix_store(Group g, joint_matrix<>src, T *base, unsigned stride, Layout layout);
- Multiply and add - Element-wise ops - Extensible to add more operations	• joint_matrix<> joint_matrix_mad(Group g, joint_matrix<>A, joint_matrix<>B, joint_matrix<>C); • void joint_matrix_apply(Group g, joint_matrix<>A, F&& func);

SYCL joint_*matrix* *Example*

```
using namespace sycl::ext::oneapi::experimental::matrix;
queue q;
range<2> G = {M/tM, N/tN * SG_SIZE};
range<2> L = {1, SG_SIZE};
auto bufA = sycl::buffer{memA, sycl::range{M*K}};
auto bufB = sycl::buffer{memB, sycl::range{K*N}};
auto bufC = sycl::buffer{memC, sycl::range{M*N}};
q.submit([&](sycl::handler& cgh) {
  auto accA = sycl::accessor{bufA, cgh, sycl::read_only};
  auto accB = sycl::accessor{bufB, cgh, sycl::read_only};
  auto accC = sycl::accessor{bufC, cgh, sycl::read_write};
  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, int8_t, use::a, tM, tK, layout::row_major> subA;
     joint_matrix<sub_group, int8_t, use::b, tK, tN, layout::row_major> subB;
     joint_matrix<sub_group, int32_t, use::accumulator, tM, tN> subC;
     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, N);
       subC = joint_matrix_mad(sg, subA, subB, subC);
     }
     joint_matrix_apply(sg, subC, [=](T &x) { Relu(x); });
     joint_matrix_store(sg, subC.get_pointer(), accC + sg_startx * tM * N + sg_starty,
N, row_major);
  });
});
q.wait;
```

oneAPI 2023.1: One Joint Matrix Code to Run on Intel AMX, Intel XMX and Nvidia* Tensor Cores

```
joint_matrix<sub_group, int8_t, use::a, tM, tK, layout::row_major> subA;
joint_matrix<sub_group, int8_t, use::b, tK, tN, layout::row_major> subB;
joint_matrix<sub_group, int32_t, 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);
  subC = joint_matrix_mad(sg, 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);
```

Intel CPUs

Intel GPUs

Nvidia* GPUs

oneAPI

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*B + sum_rows_A + sum_cols_B + scalar_zero_point*
- *sum_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

A type	Btype	Ctype	M	N	K
(u)int8_t	(u)int8_t	int32_t	<=16	<=16	<=64
bf16	bf16	float	<=16	<=16	<=32

Intel XMX Supported Combinations

A type	Btype	Ctype	M	N	K
(u)int8_t	(u)int8_t	int32_t	<=8	8 (ATS-M) 16 (PVC)	32
fp16	fp16	float	<=8	8 (ATS-M) 16 (PVC)	16
bf16	bf16	float	<=8	8 (ATS-M) 16 (PVC)	16
tf32	tf32	float	<=8	16 (PVC)	

oneAPI

Nvidia* Tensor Cores Supported Combinations

A type	Btype	Accumulator type	M	N	K
half	half	float	16	16	16
			32	8	16
			8	32	16
half	half	half	16	16	16
			32	8	16
			8	32	16
bfloat16	bfloat16	float	16	16	16
			32	8	16
			8	32	16
tf32	tf32	float	16	16	8
(u)int8_t	(u)int8_t	int32_t	16	16	16
			32	8	16
			8	32	16

Matrix Query

Static Query

Provide a default shape if user does not provide a combination in a *constexpr* way

```
namespace sycl::ext::oneapi::experimental::matrix
template<sycl::ext::oneapi::experimental::architecture Dev, typename Ta, typename Tb,
  typename Taccumulator>
struct matrix_params {
 static constexpr size_t M = /* implementation defined */;
 static constexpr size_t N = /* implementation defined */;
 static constexpr size_t K = /* implementation defined */;
 template <typename Group, layout Layout>
 using joint_matrix_a = joint_matrix<Group, Ta, use::a, M, K, Layout>;
 template <typename Group, layout Layout>
 using joint_matrix_b = joint_matrix<Group, Tb, use::b, K, N, Layout>;
 template <typename Group>
 using joint_matrix_accumulator = joint_matrix<Group, Taccumulator, use::accumulator, M, N>;
};
```

Runtime Query

Tell the set of supported matrix sizes and types on this device

(Not implemented yet)

```
namespace sycl::ext::oneapi::experimental::info::device::matrix
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);
 }
}
```

SYCL joint_*matrix Using the Default Query*

```
using namespace sycl::ext::oneapi::experimental::matrix;
using myparams = matrix_params<sycl::ext::oneapi::experimental::architecture::intel_gpu_pvc,
                              int8_t, int8_t, int>;
constexpr int tM = myparams::M;
constexpr int tN = myparams::N;
constexpr int tK = myparams::K;
range<2> G = {M/tM, N/tN * SG_SIZE};
range<2> L = {1, SG_SIZE};
// buffers
q.submit([&](sycl::handler& cgh) {
  // accessors
  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();
    myparams::joint_matrix_a<sub_group, layout::row_major> tA;
    myparams::joint_matrix_b<sub_group, layout::row_major> tB;
    myparams::joint_matrix_accumulator<sub_group> tC;
    joint_matrix_fill(sg, tC, 0);
    for (int k = 0; k < K; k += tk) {
     joint_matrix_load(sg, tA, memA + sg_startx * tM * K + k, K);
     joint_matrix_load(sg, tB, memB + k * N + sg_starty, N);
     tC = joint_matrix_mad(sg, tA, tB, tC);
   }
   joint_matrix_store(sg, tC, memC + sg_startx * tM * N + sg_starty, N, row_major);
  });
});
```

# SYCL joint_matrix	# CUDA Fragments

```
// 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<sub_group, 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 < K; 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);
    matC = joint_matrix_mad(sg, 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);

}
```

```
// inputA is MxK, inputB is KxN, inputC is MxN

#define tM=16 tN=16 tK=16


__global__ void wmma_ker(blockidx) {

  fragment<matrix_a, 16, 16, 16, half, col_major> a_frag;
  fragment<matrix_b, 16, 16, 16, half, row_major> b_frag;
  fragment<accumulator, 16, 16, 16, float> c_frag;

  uint row = (blockIdx%x - 1)*tM + 1
  uint col = (blockIdx%y - 1)*tN + 1

  fill_fragment(c_frag, 0.0f);

  for (uint step = 0; step < K; step += matrixDepth) {

      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);

   }

   for(int t=0; t<matC.num_elements; t++)

      matC.x[t] *= alpha;

   store_matrix_sync(inputC+row*N+col, matC, N, mem_row_major);

}
```

oneAPI

Current Users of Joint Matrix

Code migration from wmma samples	• https://github.com/wzsh/wmma_tensorcore_sample/tree/master/matrix_wmma/matrix_wmma • https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/cudaTensorCoreGemm
Porting code from wmma to joint_matrix	• Porting an earthquake simulation code that makes direct use of the tensor cores through wmma from CUDA and wmma to SYCL and joint matrix
SYCL-DNN – By CodePlay	• Using joint_matrix for enabling Nvidia Tensor Cores in SYCL-DNN
SYCL-BLAS – By CodePlay	• Using joint_matrix for enabling Nvidia Tensor Cores in SYCL-BLAS GEMM
SPIRV MLIR Dialect	• XMX Support using MLIR SPIRV dialect by adding SPIRV joint_matrix

Conclusion and Next Steps

- Full support of SYCL joint matrix extension on Intel AMX, Intel XMX, and Nvidia Tensor Cores
- 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
  - Standardization of SPIRV joint matrix to Khronos SPIRV
- Contributions/feedback are welcome

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.