IWOCL 2024

The 12th International Workshop on OpenCL and SYCL

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

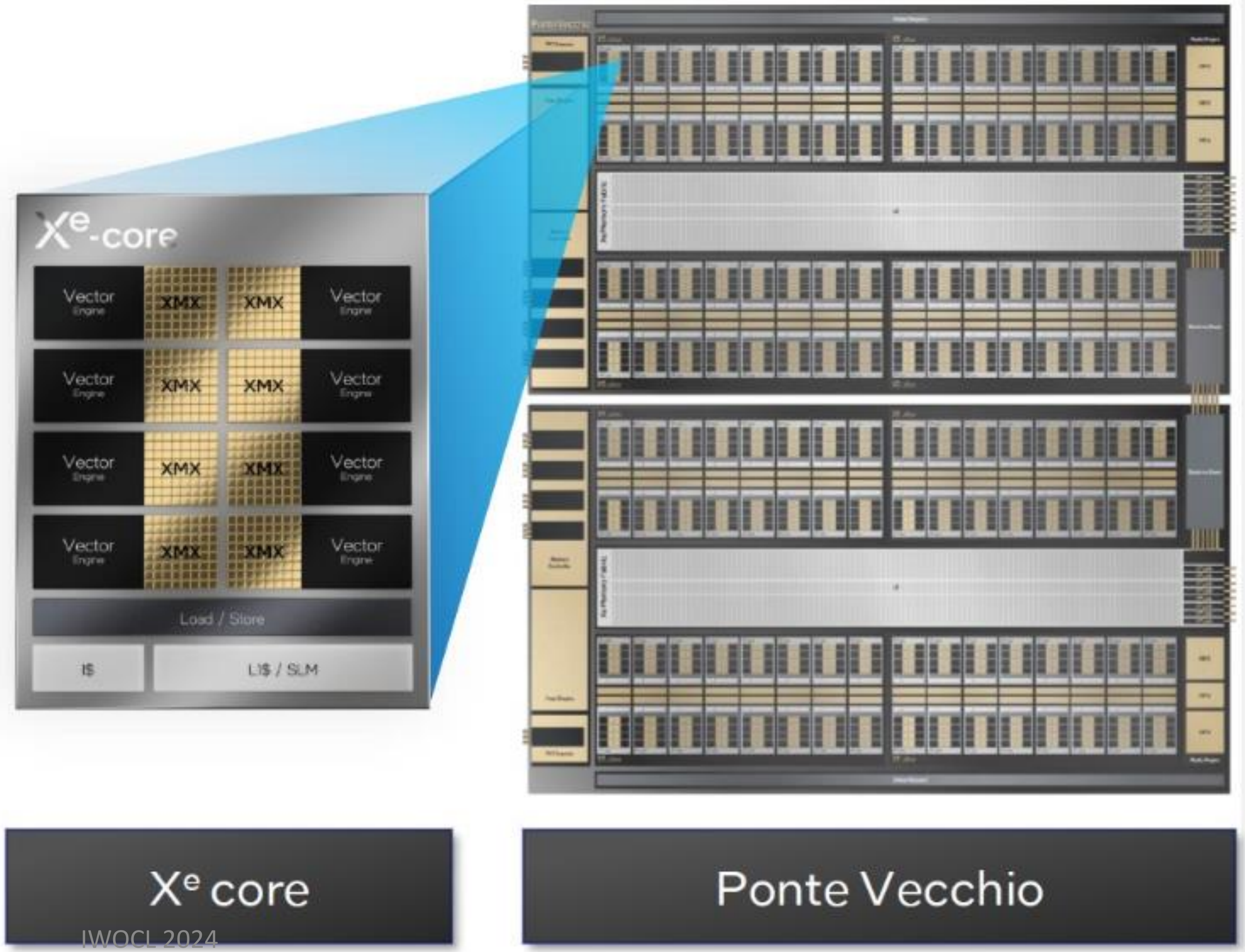
Outline

- 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

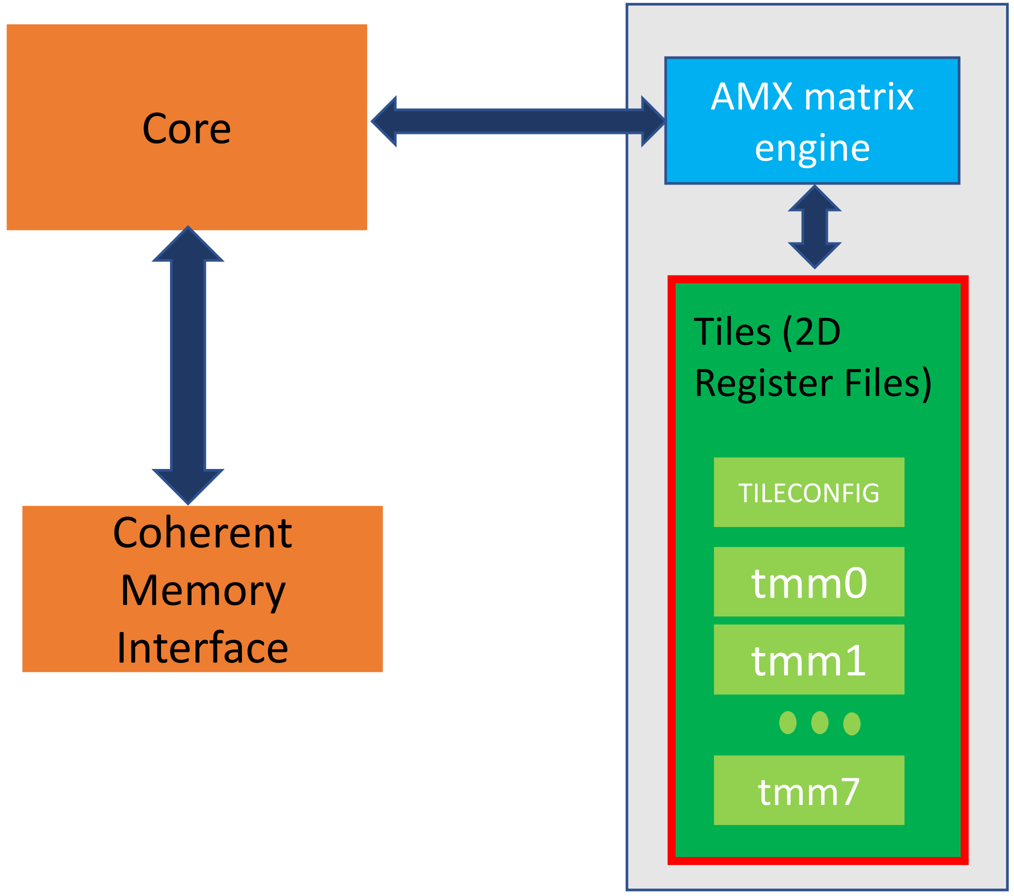
Intel XMX in Intel® Data Center GPU Max Series

- 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

SYCL Matrix Extension

SYCL joint_matrix	Description
namespace sycl::ext::oneapi::experimental::matrix	Namespace
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}; enum class layout {row_major, col_major, dynamic};	- New matrix data type with group scope - Defined with a specified type, use (a, b, accumulator), shape, and layout

joint_matrix_fill(Group g, joint_matrix<>&dst, T v);
void joint_matrix_load(Group g, joint_matrix<>&dst, multi_ptr<> src, size_t stride, Layout layout); // use::accumulator matrix
void joint_matrix_load(Group g, joint_matrix<>&dst, multi_ptr<> src, size_t stride); // use::a and use::b matrices
void joint_matrix_store(Group g, joint_matrix<>src, multi_ptr<> dst, unsigned stride, Layout layout); //use::accumulator matrix
	- Separate memory operations from the compute
	- Group execution scope → *joint, Group as argument*

•joint_matrix_mad(Group g, joint_matrix<>&D, joint_matrix<>&A, joint_matrix<>&B, joint_matrix<>&C); •void joint_matrix_apply(Group g, joint_matrix<>&A, F&& func); •void joint_matrix_copy(Group g, joint_matrix<Group, T1, Use1, Rows, Cols, Layout1> &dest, joint_matrix<Group, T2, Use2, Rows, Cols, Layout2> &src);	- Multiply and add - Element-wise ops - Copy

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};
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

Parameters are not portable → use query

```
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);
```

Intel CPUs

Intel GPUs

Nvidia* GPUs

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<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);
    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) {

  fragment<matrix_a, tM, tN, tK, half, col_major> matA;
  fragment<matrix_b, tM, tN, tK, half, row_major> matB;
  fragment<accumulator, tM, tN, tK, float> matC;

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

  fill_fragment(matC, 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);

}
```

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*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 (DG2) 16 (PVC)	32
fp16	fp16	float	<=8	8 (DG2) 16 (PVC)	16
bf16	bf16	float	<=8	8 (DG2) 16 (PVC)	16
tf32	tf32	float	<=8	16 (PVC)	

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 Runtime Query

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

```
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;
};
```

```
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(…);
```

Performance Tuning: Blocking for data reuse

Bfloat16 type example	**SYCL terms**	**Compute unit and data**	**Intel XMX**	**Intel AMX**
Workgroup execution hierarchy level/ Cache locality	Local range	• Data in L1 cache	(MCACHExNCACHExKCACHE) =256x256x32 for 4096 GEMM size	(MCACHExNCACHExKCACHE) = 256x256x1024 for 4096 GEMM size
Subgroup execution hierarchy level	Inside the kernel	• Data in registers	MSGxNSGxKSG = 32x64x16	MSGxNSGxKSG = 32x32x32 to occupy the 8 AMX tiles
Joint matrix shape	Inside the kernel	Matrix hardware	MJMxNJMxKJM = 8x16x16	MJMxNJMxKJM = 16x16x32
Additional runtime options			SYCL_PROGRAM_COMPILE_OPTIONS="-ze-opt-large-register-file"	• DPCPP_CPU_NUM_CUS to set number of threads • DPCPP_CPU_PLACES for threads to core affinity

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.

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*B + sum_rows_A + sum_cols_B + scalar_zero_point*
- *sum_rows_A* returns a single row of A

```
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];
  }
}
```

```
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;
  });
}
```

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