Extending SYCL's Programming Paradigm with Tensor-based SIMD Abstractions

Wilson Feng, Shucai Yao, Kai Ting Wang, Md Aamir Raihan, Laichun Feng, Chunrong Xu
Huawei Canada Research Centre
Markham, Canada
{wilson.cx.feng,shucai.yao,kai.ting.wang,md.aamir.raihan,fenglaichun,xuchunrong2}@huawei.com

ABSTRACT

Heterogeneous computing has emerged as an important method for supporting more than one kind of processors or accelerators in a program. There is generally a trade off between source code portability and device performance for heterogeneous programming. Thus, new programming abstractions to assist programmers to reduce their development efforts while minimizing performance penalties is extremely valuable.

The Khronos SYCL [2] standard defines an abstract single-program-multiple-data (SPMD) programming model for heterogeneous computing. This paper presents a language extension on top of the SYCL standard to enable flexibility for programmers. We introduce a set of single-instruction-multiple-data (SIMD) abstractions based on multi-dimensional arrays (Tensors) in conjuction with the existing SPMD programming paradigm.

Our work is based on a C++ language and a set new of LLVM intermediate representation (IR) for representing the SIMD programs. This also includes a set of custom optimization passes that performs instruction lowering, automatic address allocation, and synchronization insertion. We show how our work can be used in conjunction with conventional SYCL SPMD programming for various benchmarks such as general matrix multiplication (GEMM) and lower upper (LU) inverse [5] and evaluate its hardware utilization performance.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; • **Computer systems organization** → **Single instruction, multiple data**.

KEYWORDS

SYCL, Tensor, Parallel Computing, LLVM

ACM Reference Format:
Wilson Feng, Shucai Yao, Kai Ting Wang, Md Aamir Raihan, Laichun Feng, Chunrong Xu. 2022. Extending SYCL's Programming Paradigm with Tensor-based SIMD Abstractions . In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3489525.3511681

1 INTRODUCTION

With recent development of various device accelerators in domains such as Deep Neural Network (DNN) [6], SYCL has become one of the programming paradigms of interest for heterogeneous computing. SYCL programming is based on standard ISO C++ with a SPMD-based abstraction, and so developers may face challenges in writing their DNN algorithms. There are various reasons such as having to perform arithmetic operations on multi-dimensional arrays across specific axes. These types of expressions become challenging to realize into source code due to the SYCL language itself only provides C++-based abstractions. Thus, programmers resort to having to write multiple nested for loops with complicated multi-dimensional indexing. There are many SYCL implementations in development with possibly Intel's DPC++ being the most actively developed implementation that is open source [1].

Figure 1 presents the Ascend hardware architecture overview [4], at a per-core level. Each core maps to a logical thread abstraction in the SYCL SPMD programming model. Thus, each core can be asynchronously performing different compute tasks in multiple hardware pipelines.

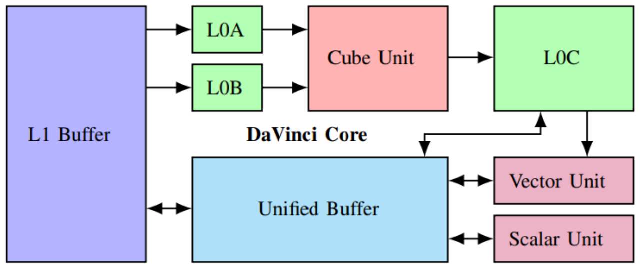

Figure 1: Overview of the Ascend Architecture

Our work proposes a new language extension on top of the existing SYCL abstractions that addresses such domain specific programming challenges. In summary, we make the following new contributions:

- We introduce a new set of language extensions based on a custom multi-dimensional array type called, Tensor, and its operations. These extensions work in conjunction with the existing SYCL abstractions.
- Introduce new LLVM optimization passes that perform various lowering functionalities for the Tensors and its operations such as address allocation, auto-synchronization insertion, and instruction lowering.
- Demonstrate our extension with various benchmark applications and its performance results.

2 LLVM IMPLEMENTATION

2.1 Ascend C++ Language Extension

In this section, we introduce the tensor extensions, including tensor definitions, special tensor concepts, tensor format and tensor assignment operator. Then, we elaborate the tensor operations in the rest part of this section.

2.1.1 Tensor Extensions.

Tensor Definitions. A *Tensor*, shown in Listing 1, is a collection of elements and it could have up to 5 dimensions. Tensor-related attributes are as follow:

- *Shape* specifies the number of the elements in each dimension.
- *Stride* specifies the distance between consecutive elements in a dimension in a linear address space.
- *Coord* specifies the coordinate index in a tensor.
- *Attr* specifies the tensor (or fractal) memory layout information and the double buffering attribute.
- *AS* specifies the address space. It is passed via the template parameter and the default is global

```
template <typename T, AS As = AS::GM> class Tensor final {
  // Define a global memory tensor
  Tensor(__gm__ T *addr, const Shape &shape,
         const uint64_t attr = Attr::CONTIGUOUS){...}
  // Define a local tensor
  Tensor(const Shape &shape,
         const uint64_t attr = Attr::CONTIGUOUS){...}
  ...
}
```

Listing 1: Tensor Definitions

Special Tensor Concepts. We introduce two special tensor concepts:

- *Local* tensor, shown in Listing 1, is created by the constructor `Tensor(const Shape &shape, ...)` and its address is allocated by the compiler. A local tensor is created when we chain the tensor operations together. For example, `dst = tsqrt(tadd(src0, sc1))` is equal to `tmp = tadd(src0, src1); dst = tsqrt(tmp)`. The related tensor operations will be discussed in Section 2.1.3.
- *Virtual* tensor is a **view** of the original tensor. That is, the virtual tensor shares the memory with the original tensor. The related operations will be introduced in Section 2.1.2.

Tensor Format. Tensor format defines the tensor's data layout in memory. By default, the format is contiguous. That is, a tensor is constructed with the shape `Shape(M, K, N)`, the elements can be accessed by `Tensor(m, k, n)`. The address is calculated through: `(datatype *)start_addr + n + k*N + m*K*N`. It also supports the other formats, such as `TENSOR_FORMAT_Nz`, `FRACTAL_Zz`, `FRACTAL_Zn` and `NC1HWC0`.

Tensor Assignment. The following tensor assignment operations, shown in Listing 2, are supported:

(1) The elements of the global memory tensor *src1* are assigned to the UB tensor *tmp1*.
(2) Elements of a slice (virtual tensor) of tensor *src20* are assigned to the UB tensor *src21*.
(3) The elements of the L1 Buffer tensor *src30* are assigned to the UB tensor *src31*.
(4) The elements of the UB tensor *src4* are assigned to the UB tensor *tmp4*.

```
Shape s16x16(16, 16);
Shape s1x16(1, 16);
// 1. Assign a global memory tensor to a local UB tensor
Tensor<half> src1(s16x16);
Tensor<half, AS::UB> tmp1 = src1;
// 2. Assign a virtual tensor to a local UB tensor
Tensor<half, AS::UB> src20(s16x16);
Tensor<half, AS::UB> src21(s1x16);
src21 = src20.strided_slice(s1x16, stride(16, 1), 0);
// 3. Assign a local CBUF tensor to another local UB tensor
Tensor<half, AS::CBUF> src30(s16x16);
Tensor<half, AS::UB> src31(s16x16);
src31 = src30;
// 4. Assign a local UB tensor to another local UB tensor
Tensor<half, AS::UB> src4(s16x16);
Tensor<half, AS::UB> tmp4(s16x16);
tmp4 = src4;
```

Listing 2: Tensor Assignment

2.1.2 Tensor Member Operations.

Slice returns a virtual sub-tensor from the original tensor. Its function prototype is shown in Listing 3, as a member function of the tensor class. For example, `src.slice(Shape(1, 5), Coord(1, 0))`. The return tensor shares memory with the original tensor.

```
// dstS: the shape of destinate sub-tensor.
// dstC: the starting coordinate when slicing.
Tensor slice(const Shape &dstS, const Coord &dstC);
```

Listing 3: Tensor Member Operations

2.1.3 Tensor Element-Wise Operations.

All the tensor operations in this section, shown in Listing 4, are element-wise operations. The operations always returns a tensor which has the same shape as the input tensors. The input tensors should have the same shape if there are two input tensors. To simplify the discussion, we omit the template definition for the following tensor operations in this paper.

```
// Unary Tensor Element-Wise Operations
Tensor<T> tUnaryOp(const Tensor<T> &src)
// Binary Tensor Element-Wise Operations
Tensor<T> tBinaryOp(const Tensor<T> &src0, const Tensor<T> &src1)
// Tensor and Scalar Element-Wise Operations
Tensor<T> tOpScalar(const Tensor<T> &src, T s)
// Tensor Compare and Selection Element-Wise Operations
Tensor<T> tcmp(const Tensor<T> &src0, const Tensor<T> &src1, CmpOp op)
Tensor<T> tcmps(const Tensor<T> &src, T scalar, CmpOp op)
Tensor<T> tsel(const Tensor<T> &src0, const Tensor<T> &src1, MASK mask)
Tensor<T> tcmpsel(const Tensor<T> &lhs, const Tensor<T> &rhs, const Tensor
  <T> &slhs, const Tensor<T> &srhs, CmpOp op)
// Tensor Conversion Element-Wise Operations
Tensor<T1> tconv(const Tensor<T> src, T1 cast_type)
```

Listing 4: Tensor Element-Wise Operations

Unary Tensor Operations. The *tUnaryOp* in the Listing 4 supports following operations: *texp* (exponential), *tln* (natural logarithm), *tsqrt* (square root), *trsqrt* (reciprocal of square root), *trec* (reciprocal), *trelu* (*RELU* operation on the *src* tensor. It is equivalent to `dst[n] = src[n] < 0 ? 0 : src[n]`), *tlrelu* (leaky *RELU* operation on the *src* tensor. It is equivalent to `dst[n] = src[n] < 0 ? src[n] * factor : src[n]`), *tabs* computes the absolute value. *tnot* performs the logical *not* operation.

Binary Tensor Operations. The *tBinaryOp* in the Listing 4 supports the following operations: *tadd* (addition), *tsub* (subtraction), *tmul* (multiplication), *tdiv* (division), *tmax* (max value for each element in the *src* and *dst* tensors), *tmin* (min value for each element in the *src* and *dst* tensors), *tand* (logical *and*), *tor* (logical *or*).

Tensor and Scalar Operations. The *tOpScalar* in the Listing 4 supports following operations:

- *tadds* adds each element of the *src* tensor with the scalar value *s*.
- *tmuls* multiplies each element of the *src* tensor with the scalar value *s*.
- *tmaxs* returns the max value of each element of *src* tensor and the scalar value *s*. It equals to `dst[n] = max(src[n], s)`.
- *tmins* returns the min value of each element of *src* tensor and the scalar value *s*. It equals to `dst[n] = min(src[n], s)`.

Tensor Compare and Selection Operations. The *CmpOp* can be `EQ`, `NE`, `LT`, `GT`, `GE`, `LE`. The following tensor compare and selection operations are supported in the Listing 4:

tcmp (compares each element in *src0* and *src1* tensors), *tcmps* (compares each element in *src* tensor with the scalar value *s*), *tsel* (performs selection between *src0* and *src1* tensors, based on *mask*. It equals to `dst[n] = (mask[n] = = 1)? src0[n] : src1[n]`), *tcmpsel* (compares each element in *lhs* and *rhs* tensors. If the result of an element is true, the element in *slhs* tensor would be selected. Otherwise, the element in *srhs* tensor would be selected).

Tensor Conversion Operation. The following tensor data type conversions are supported:

- *half2float* converts data type from `half` to `float`.
- *float2half* converts data type from `float` to `half`.

2.1.4 *Tensor Reduction Operations.*

Tensor reduction operations are defined in the Listing 5. The *axis* represents the axis that does reduction operation. The *keepDim* is used to check whether to keep the reduction axis or not.

- *treduce_max* gets the maximum element belonging to specific axis of the *src* tensor.
- *treduce_min* gets the minimum element belonging to specific axis of the *src* tensor.

```
// src: the source tensor
Tensor<T> treduce_max(const Tensor<T> &src,
                      const enum axis,
                      bool keepDim = false)
Tensor<T> treduce_min(const Tensor<T> &src,
                      const enum axis,
                      bool keepDim = false)
```

Listing 5: Tensor Reduction Operations

2.1.5 *Matrix Multiplication.*

Tmmad operation does a hardware accelerated matrix-multiplication. It supports two versions of matrix multiplication:

(1) `AccMatrix = LhsMatrix X RhsMatrix + AccMatrix`
(2) `AccMatrix = LhsMatrix X RhsMatrix`

```
// LhsMatrix: left matrix, shape is m x k.
// RhsMatrix: right matrix, shape is k x n.
// AccMatrix: result matrix, shape is m x n
// AccMatrix = LhsMatrix X RhsMatrix
Tensor<DstT> tmmad(const Tensor<LhsT> &LhsMatrix,
                   const Tensor<RhsT> &RhsMatrix)
// AccMatrix = LhsMatrix X RhsMatrix + AccMatrix
Tensor<DstT> tmmad(const Tensor<LhsT> &LhsMatrix,
                   const Tensor<RhsT> &RhsMatrix,
                   const Tensor<DstT> &AccMatrix)
```

Listing 6: Matrix Multiplication

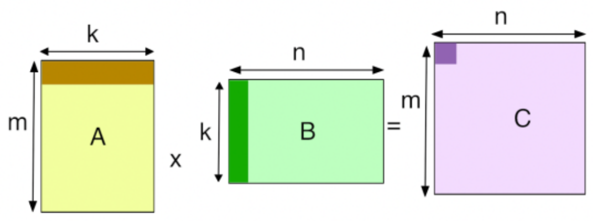

Figure 2: Matrix Multiplication

2.1.6 *Tensor Copy Operations.*

There are three tensor copy operations: *tcpi*, *tcpo* and *tmov*.

- *tcpi* copies a smaller tensor's contents into a bigger tensor starting at a given coordinate, *coord*. The bigger tensor must be able to fit into the smaller tensor's shape.
- *tcpo* copies a sub-tensor's contents, from the bigger tensor, into the smaller tensor starting at a given coordinate,a *coord*. The smaller tensor must be able to fit into the sub-tensor's shape.
- *tmov* copies one tensor into another, the *src* and *dst* tensor should have the same shape.

Tcpi and *tcpo* allow that *src* and *dst* tensors have the same shape, in this case, the semantic is the same as *tmov*.

```
// dst: the destination tensor.
// src: the source tensor.
// coord: the starting coordinate.
void tcpi(const Tensor<T> &dst,
        const Tensor<T> &src,
        const Coord &coord)
void tcpo(const Tensor<T> &dst,
        const Tensor<T> &src,
        const Coord &coord)
void tmov(const Tensor<T> &dst,
        const Tensor<T> &src)
```

Listing 7: Tensor Copy Operations

2.2 Custom Compilation Tool Chain

In order to support the Tensor language extension on top of the existing SYCL language using the Ascend heterogeneous programming compiler, we present a custom modified tool chain. Figure 3 is a system level architecture overview for our custom tool chain done in the clang driver. We leverage the existing clang front end that already contains all the SYCL language specifications implemented, and introduce new Tensor C++ classes along with intrinsic instructions. The output of clang's code generation is LLVM-IR with the new Tensor intrinsic instructions. This IR module is then passed to a custom LLVM project where it contains a series of Ascend specific optimization passes. These custom optimization passes converts the Tensor-based level of abstraction in the IR to become intrinsic instructions closer to the actual Ascend hardware instructions. Finally, the IR is passed to the back end to be lowered into a binary. The rest of the compilation flow follows the existing SYCL compilation tool chain.

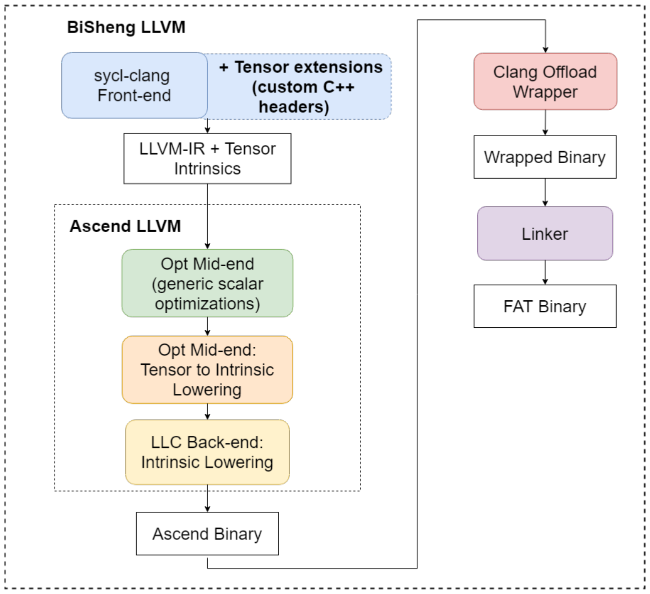
BiSheng LLVM
sycl-clang Front-end
+ Tensor extensions (custom C++ headers)
LLVM-IR + Tensor Intrinsics
Ascend LLVM
Opt Mid-end (generic scalar optimizations)
Opt Mid-end: Tensor to Intrinsic Lowering
LLC Back-end: Intrinsic Lowering
Ascend Binary
Clang Offload Wrapper
Wrapped Binary
Linker
FAT Binary

Figure 3: Overview of the Custom Compilation Tool Chain

2.3 Custom Optimization Passes

In this section, we first introduce the core data structures which are used to maintain the tensor related information during the IR transformation. Second, we describe the All-in-One module level transform pass which first runs all the analysis passes and then performs the IR transformation. Third, we elaborate the Ascend specific LLVM function analysis passes which the all-in-one transform requires.

2.3.1 *Data Structures.*

Tensor Context. A module-level singleton *TensorContext* (shown in Listing 8) is maintained in the backend. It contains *TensorTable* and *TensorGraph* for each function.

```
class TensorContext {
public:
  static TensorContext &getInstance() {
    static TensorContext TLGlobal;
    return TLGlobal;
  }
  ...
private:
  Map<Function *, TensorTable *> Func2TensorTable;
  Map<Function *, TensorGraph *> Func2TensorGraph;
  ...
  TensorContext() {
    ...
  }
}
```

Listing 8: Tensor Context

Tensor Table. TensorTable, shown in Listing 9, collects all the tensor information (except for CFG-related) in the function. For example, tensor intrinsic lowering information.

```
class TensorTable {
public:
  TensorTable(const Function &F) { Func = &F; }
private:
  SmallPtrSet<Tensor *, 16> Tensors = {};
  // Mapping between tensor level intrinsic
  // and lowered Basic Blocks
  Map<const Instruction *, pair<BasicBlock *, BasicBlock *>>
    Intrinsic2BBsMap;
  ...
}
```

Listing 9: Tensor Table

Tensor Graph. TensorGraph, shown in Listing 10, records the tensor CFG-related information, such as instruction insertion position.

```
class TensorGraph {
public:
  TensorGraph(const Function &Func) : F(&Func){};
  ...
private:
  // One-to-one map between BasicBlock and Vertex
  Map<const BasicBlock *, TLVertex *> BB2VertexMap;
};
```

Listing 10: Tensor Graph

2.3.2 *All-in-one Transform Pass.*

All-in-one module-level transform pass contains four function level transform passes: TLOplowering, Address Allocation, Double Buffering and Auto synchronization. Instead of modifying the IR in place, all-in-one transform pass follows a *Copy-Update-Replace* style. There are two copies for each kernel function: *OldF* and *NewF*. *OldF* is immutable while *NewF* is modified during the transform pass.

2.3.3 *Tensor Def-Use and Liveness Analysis.*

The tensor Def-Use analysis pass first collects all the tensor *def* and *use* information. Then, tensor liveness analysis pass applies the standard variable liveness algorithm, to compute tensor liveness.

2.3.4 *Tensor OpLowering Analysis.*

Tensor OpLowering analysis pass converts the tensor level operations into lower level intrinsic. For example the *tadd* in Listing 11, it takes two input tensors, *in1* and *in2* and returns the output tensor *dst*. Based on the tensor's type, it setups the configuration parameters for the lower level intrinsic, such as, repeat times. It could generate multiple intrinsic if the repeat number exceeds the limit. Tensor OpLowering analysis pass creates all the lowered intrinsic code and collects pipeline information for each tensor-level operation, which will be used by auto synchronization analysis pass.

```
  ; Tensor-Level operator
  dst = tadd(in1, in2);

  ; Low-Level Intrinsic
  call void llvm.tl.vadd(i64 1, i64 3, i64 0,
        i64 %17, ... ; dst Tensor Parameters
        i64 %13, ... ; in1 Tensor Parameters
        i64 %14, ...); in2 Tensor Parameters
```

Listing 11: Tensor OpLowering Analysis Pass

2.3.5 Tensor Address Allocation Analysis. Tensor address allocation analysis pass collects all the tensor address space information and assign the address for the tensors (shown in Listing 12). It also allows two tensors to use the same memory space if there is no overlap between these two tensors' live range.

```
%13 = rt.malloca() ; Address starts 0x1000
%14 = rt.malloca() ; Address starts 0x2000
%17 = rt.malloca() ; Address starts 0x3000
; Low-Level Intrinsic
call void llvm.tl.vadd(i64 1, i64 3, i64 0,
                i64 %17, ... ; dst Tensor Parameters
                i64 %13, ... ; in1 Tensor Parameters
                i64 %14, ...); in2 Tensor Parameters
```

Listing 12: Tensor Address Allocation Analysis Pass

2.3.6 Auto Synchronization Analysis. Ascend C++ provides the users with an synchronous programming model while the Ascend hardware has multiple asynchronous hardware pipelines. That is, the compiler needs to analyze the synchronization information and automatically insert the synchronization instructions into the LLVM IR. Based on the analysis results of Tensor liveness analysis pass, the synchronization analysis pass determines the appropriate location to insert the synchronization instructions. Low level intrinsic `sync_event` is inserted by the backend to guarantee the synchronization between different hardware pipelines. For example, `ubTensor` in Listing 13 has a Read-After-Write (RAW) hazard and `sync_event` is inserted by the backend.

```
ubTensor = tmuls(ubTensor, scalar);
// sync_event()
gmTensor = ubTensor;
```

Listing 13: Auto Synchronization

2.3.7 Double Buffering Analysis. Ascend hardware supports memory read and write operations at the same time. Therefore it is possible for the compiler to overlap read and write operations in order to improve the overall performance. The example in Listing 14 demonstrates how to use the double buffering feature. We can see in Figure 4 read and write `ubTensor` operations are overlapped among adjacent iterations.

```
// Double buffering is enabled: Attr::DB
Tensor<half, AS::UB> ubTensor(ShapeUB, Attr::DB);
for (....){
  // read ubTensor
  ... = op(..., ubTensor);
  // write ubTensor
  ubTensor = ...;
}
```

Listing 14: Double Buffering Example

3 EXPERIMENTS

In order to demonstrate our language extension, we present the following benchmarks written in the SYCL specification with our Tensor language extension, validate its correctness and measure its performance counters on the Ascend hardware. The Ascend accelerator contains 32 physical cores, clocked at 1GHz, with 32GB of HBM memory. The clock speed of the server, the host CPU is 2.6GHz with 64 KB L1 data cache, 512 KB L2 cache and 24M L3 cache.

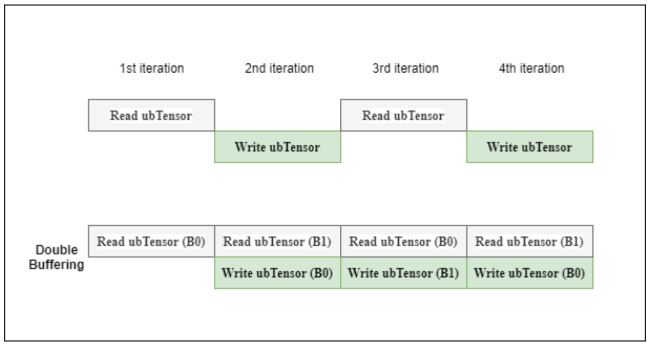

Figure 4: Double Buffering

3.1 Benchmarks

The following are various benchmarks written using SYCL and our Tensor extension. We describe the language extensions used and how it maps to our Ascend hardware under the hood.

3.1.1 GEMM. The following example is based on a General Matrix Multiply (GEMM) kernel. Both the input matrices are of element type half precision floating point. We vary the input sizes to evaluate how the size impacts the performance.

In this example, we are using the *single_task* SYCL abstraction which maps to a single core in the Ascend accelerator. The loop nest iterates through the input tiles to be accumulated on the output tile. Note that the *if-statement* is used to control whether the *tmadd* instruction is performing accumulation mode or not. Furthermore, inorder to achieve maximum performance from the *tmadd* instruction, the input tensors should be in special formats, *FRACTAL_Zz* and *FRACTAL_Zn.*

3.1.2 LU Inverse. This LU Inverse algorithm [5] is comprised of finding the inverse for the lower (L) and upper (U) parts of an input matrix, namely A. The pseudocode for finding the U inverse is shown below. The corresponding SYCL parallel_for implementation using our proposed tensor extension is shown in Figure 6. The pseudocode and implementation for finding the L inverse are conceptually similar and thus we omit them here. Each row update is an iterative process. For instance, updating row *i* of Uinv requires reads from row *i*, *i+1*, *i+2*, and etc. as well as from A. There are clearly data dependencies from one row to others which then require the update to be performed in a certain direction, and as shown in the pseudocode, it is from bottom up.

```
Uinv = eye(n);
Uinv(n,:) /= A(n,n);
for $i = n - 1 : 1 : -1$ do
    for $j = i + 1 : n$ do
        Uinv(i,:) = Uinv(i,:)-A(i,j)*Uinv(j,:);
    end
    Uinv(i,:) /= A(i,i);
end
```

Algorithm 1: U Inverse Algorithm

```

#define M0 16
#define K0 16
#define N0 16

constexpr uint64_t M = ...;
constexpr uint64_t N = ...;
constexpr uint64_t K = ...;

auto e1 = Queue.submit([&](cl::sycl::handler &cgh) {
  auto kern = [IN_feature_acc, IN_weight_acc, OUT_acc]() {

    using namespace ANONYMOUS::TL;
    __gm__ half *outBuf = (__gm__ half *)OUT_acc;
    __gm__ half *fmBuf = (__gm__ half *)IN_feature_acc;
    __gm__ half *wBuf = (__gm__ half *)IN_weight_acc;

    Tensor<half> gm_in_feature_map(fmBuf, Shape(M, K), Attr::FRACTAL_Zz);
    Tensor<half> gm_in_weight(wBuf, Shape(K, N), Attr::FRACTAL_Zn);
    Tensor<half> gm_out_feature_map(outBuf, Shape(M, N));

    for (uint64_t n = 0; n < N; n += N0) {
      for (uint64_t m = 0; m < M; m += M0) {
        Tensor<half, AS::CC> accMatrix(Shape(M0, N0), Attr::FRACTAL_Nz);
        for (uint64_t k = 0; k < K; k += K0) {

          Tensor<half, AS::CA> lhsMatrix(Shape(M0, K0), Attr::FRACTAL_Zz);
          Tensor<half, AS::CB> rhsMatrix(Shape(K0, N0), Attr::FRACTAL_Zn);

          tcpo(lhsMatrix, gm_in_feature_map, Coord(m, k));
          tcpo(rhsMatrix, gm_in_weight, Coord(k, n));

          if (k > 0) {
            accMatrix = tmmad(lhsMatrix, rhsMatrix, accMatrix);
          } else {
            accMatrix = tmmad(lhsMatrix, rhsMatrix);
          }
        }

        tcpi(gm_out_feature_map, accMatrix, Coord(m, n));
      }
    }

  };
  cgh.single_task<class TGEMM>(kern);
});
```

Figure 5: SYCL single_task kernel GEMM

```
sycl::range<1> BlockRange{numCore};
sycl::range<1> LocalRange{1};

auto e1 = deviceQueue.submit([&](cl::sycl::handler &cgh) {
  auto kern = [IN, OUT](sycl::nd_item<1> ids) {
    int64_t N = 384, ldA = 384, ldRes = 384, numTile = 6;
    int64_t tileSize = N / numCore;

    using namespace ANONYMOUS::TL;
    __gm__ float *inA = (__gm__ float *)IN;
    __gm__ float *result = (__gm__ float *)OUT;

    Shape shapeInput(N, ldA), shapeOutput(N, ldRes), ShapeTile(N, tileSize);
    Tensor<float> input(inA, shapeInput), out(result, shapeOutput);
    Tensor<float, AS::UB> TrigInv(ShapeTile);
    TrigInv.set(0);

    int64_t tile = ids.get_global_id(0) * tileSize;

    for (int64_t j = tile; j < tile + tileSize; ++j) {
      TrigInv.set(1, j * tileSize + j - tile);
    }

    for (int64_t col = tile; col < N; col++) {
      Shape tileAShape(N);
      Tensor<float, AS::UB> A(tileAShape);
      A = input.slice(tileAShape, Coord(col, 0));
      for (int64_t row = tile; row < col; row++) {
        float scalar = A.get(row);
        scalar = -1 * (float)scalar;
        Tensor<float, AS::UB> tmp =
            tmuls(TrigInv.slice(Shape(tileSize), Coord(row, 0)), scalar);
        TrigInv.slice(Shape(tileSize), Coord(col, 0)) =
            tadd(TrigInv.slice(Shape(tileSize), Coord(col, 0)), tmp);
      }
      float scalar2 = A.get(col);
      scalar2 = (float)1.0 / (float)scalar2;
      TrigInv.slice(Shape(tileSize), Coord(col, 0)) =
          tmuls(TrigInv.slice(Shape(tileSize), Coord(col, 0)), scalar2);
    }
    out.slice(Shape(N, tileSize), Coord(0, tile)) = TrigInv;
  };
  cgh.parallel_for<class TLINV_DYNAMIC>(nd_range(BlockRange, LocalRange),
    kern);
});
```

Figure 6: SYCL parallel_for kernel Finding Uinv

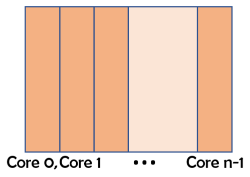

Figure 7: Parallelizing Uinv

The dependencies are from elements of a row to elements of other rows within the same columns. This allows parallelization along the horizontal (or column) dimension as shown in Figure 7 by partitioning the work into n tiles where each core (or thread) is responsible for computing the results for a single tile. This further allows vectorization of computations (e.g. element-wise multiply, subtract, and divide) within each tile. This is where our tensor extension showcases itself.

3.2 Performance Evaluation

3.2.1 GEMM. Table 1 shows the relative slow down as we increase the sizes of the input matrices, on a single Ascend core. Note that for all input sizes, there is some fixed amount of performance overhead incurred due to various reasons such as kernel launch ABI entry code being executed. We observe that for the first few relatively smaller shapes, the overall slow down was not significant. This can be explained by the fact that the amount of data payload is most likely able to fit into the inefficient slacks that were already incurred by the 16x16 case. Hence, no additional time incurred up until a certain point where the data payload got large enough such that additional data transfer and compute was required by the hardware pipeline.

Table 2 shows the relative slow down as we increase the sizes of the input matrices, on a single CPU core. Note that we used the standard clang *O2* optimization pipeline. Comparing the rate of the increased slow down between the host CPU implementation and the one for the single Ascend core, we see that the CPU version is

Shape Size	Slow down
16x16	1
32x32	1.017
64x64	1.043
128x128	1.079
256x256	2.255
512x512	9.416
1024x1024	56.42

Table 1: Ascend single_task GEMM Slowdown

Shape Size	Slow down
16x16	1
32x32	6.379
64x64	48.75
128x128	386.6
256x256	3083
512x512	24686
1024x1024	197400

Table 2: Host CPU single_task GEMM Slowdown

significantly more sensitive to the input matrices' sizes becoming larger.

3.2.2 LU Inverse. Figure 8 shows performance speedup normalized against a single core performance for finding Uinv and Linv for A being a 64x64 single precision matrix. While the CPU performance continues to scale beyond 4 cores for both the Linv and Uinv kernels, AICORE performance begins to drop after 4 cores. The key reason is the design of the AICORE's vector unit is variable in vector width with a minimal width requirement of 32 bytes (or 8 elements of fp32). Parallelizing 8-ways AICORE means the width of each tile, or each row, comprises of 8 elements of fp32, thus using the vector unit in the minimal possible way. Synchronization and data transfer overheads can no longer be easily amortized by the benefit of the vectorization leading to a slowdown in performance. Constrasting with parallelizing 2-ways AICORE where each vector operation operates on 32 elements of fp32, a speedup of 1.151x (Linv) and 1.326x (Uinv) are observed. A key lesson is optimizing for the Ascend architecture requires careful balance between parallelization and vectorization.

Figure 9 shows performance speedup normalized against 4-core, for the size of A being a 384x384 single precision matrix. We have also observed a similar pattern where the AICORE performance continues to rise and peak at 6 AICORES with 1.083x (Linv) and 1.11x (Uinv), and begins to decline afterwards.

4 RELATED WORK

Triton [7] builds a *tile* or *block* programming abstraction on top of CUDA. CUDA is a SIMT based programming language for GPU and by imposing users to program in a *block* style, Triton convenietly ensures no divergence of SIMT threads within a GPU block. Triton's shared memory allocation and synchronization are analogous to our tensor liveness and auto-synchronization optimizations described in section 2.3.

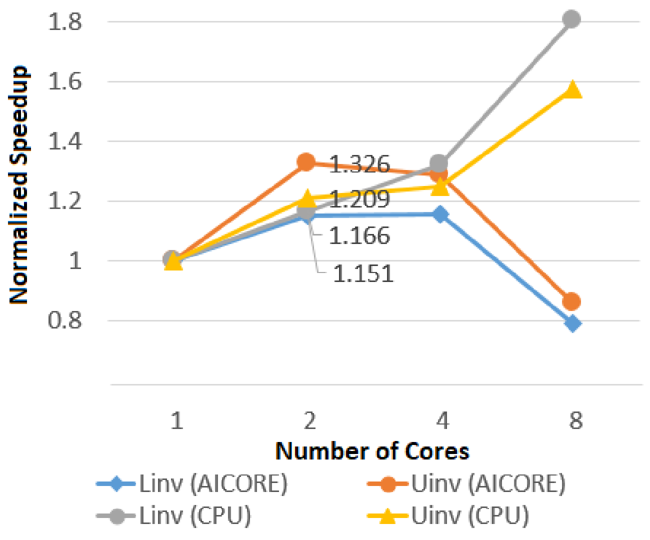

Figure 8: Scalability Study A=64x64

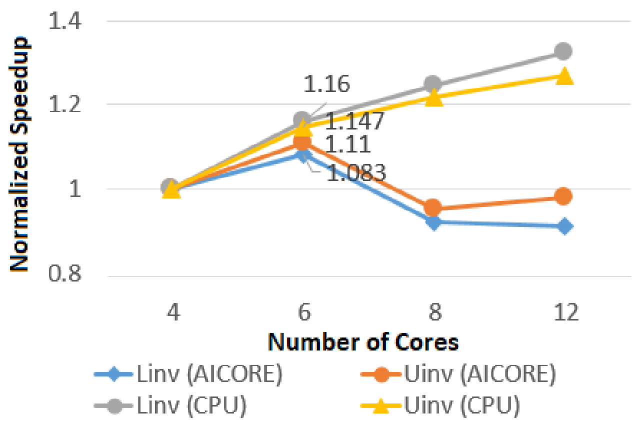

Figure 9: Scalability Study A=384x384

An alternative for supporting native SYCL programs is to use the AKG converter compilation path [3]. Instead of having SIMD-based extensions, it takes native SYCL code and pass the LLVM-IR into a custom optimization pass which generates an intermediate loopy IR. Then it is taken in by AKG and generates native Ascend source code. Similar to our approach, it takes the Ascend code and utilizes the Ascend compiler to generate an executable binary.

5 CONCLUSION

We have introduced a set of custom language extensions of the SYCL standard specifications by adding it on top of the existing clang front end. We take the clang generated LLVM-IR and pass it through a series of custom LLVM optimization passes that transforms the IR into an Ascend executable binary. We are looking into optimizing our custom LLVM passes to improve the run time performance of Anonymous binaries compiled by this compilation path.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions from our colleagues: Rasool Maghareh, Ali Nouri, Ibrahim Faruk Yalciner, Peixu Ren, Weijia Sheng, Cheng Qin, Wenqiao Chen, Ziyao Zhu, Hui Long, Yinglun Jia, Jingzhe Hu, Yiling Gao, Zhaogang Shi, Qiao Hu, Siyuan Zou, Tao Wang, Ye Tian, Ronglin Jiang, Liucheng Zhou and Xinyuan Xu.

REFERENCES

[1] 2021. Intel's LLVM Project. https://github.com/intel/llvm/.
[2] 2021. SYCL Khronos Group. https://khronos.org/sycl/.
[3] Wilson Feng, Rasool Maghareh, and Kai-Ting Amy Wang. 2021. Extending DPC++ with Support for Huawei Ascend AI Chipset. In *International Workshop on OpenCL* (Munich, Germany) *(IWOCL'21)*. Association for Computing Machinery, New York, NY, USA, Article 13, 4 pages. https://doi.org/10.1145/3456669.3456684
[4] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. 2019. DaVinci: A Scalable Architecture for Neural Network Computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–44. https://doi.org/10.1109/HOTCHIPS.2019.8875654
[5] Florent Lopez and Theo Mary. 2020. *Mixed Precision LU Factorization on GPU Tensor Cores: Reducing Data Movement and Memory Footprint.* Technical Report ICL-UT-20-13.
[6] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2021. AI Accelerator Survey and Trends. arXiv:2109.08957 [cs.AR]
[7] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10âĂŞ19. https://doi.org/10.1145/3315508.3329973