SAMSUNG

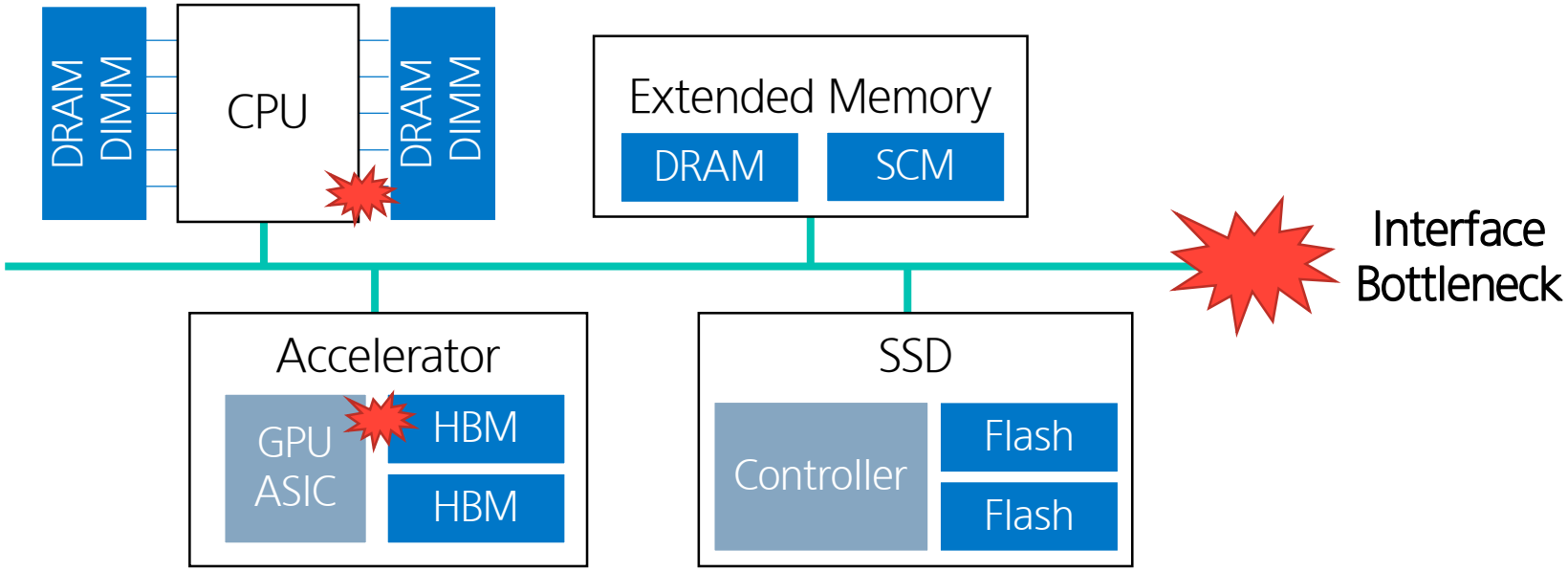
SYCL Extension Proposal for PIM/PNM

Sep 19. 2023

Hyesun Hong
Samsung Advanced Institute of Technology (Suwon, S.Korea)

Traditional Approach to Overcome Memory Bottleneck

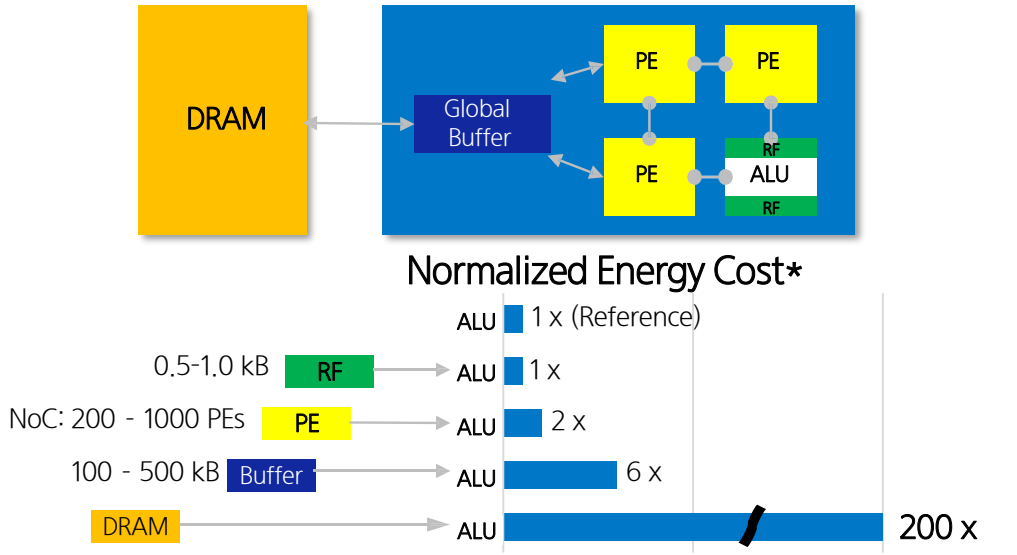
- DRAM latency is much greater than the computing latency
  - Bottlenecked by the speed/access of the DRAM
- Many techniques have been implemented to hide the latency
  - ex. Pipelining, Prefetching



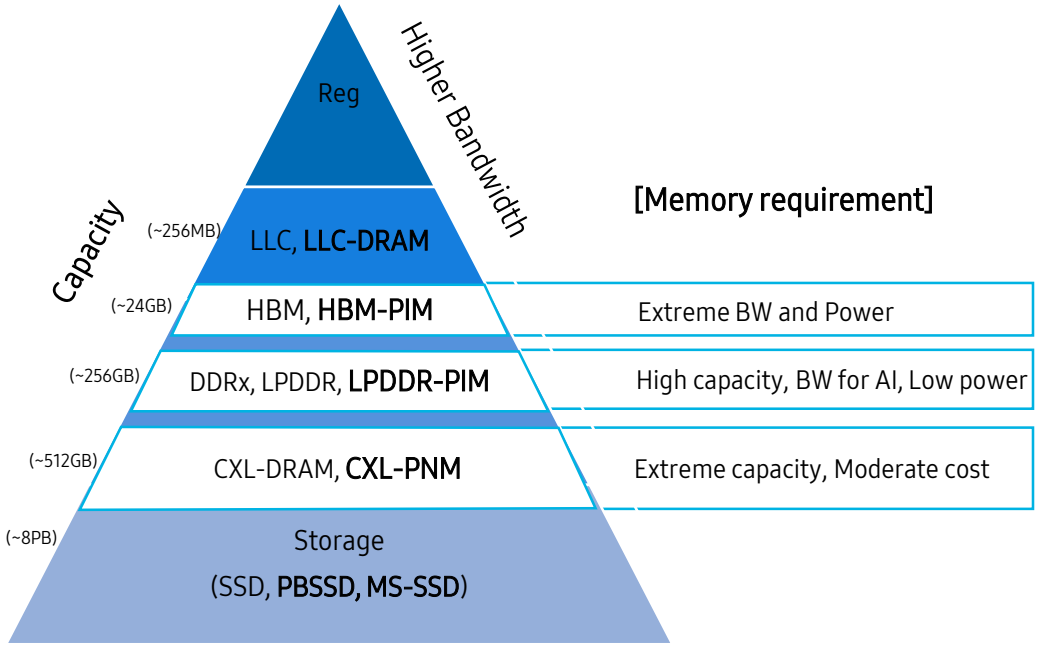
PIM/PNM on Memory Hierarchy and Energy Reduction

Source: Samsung PIM/PNM for Transformer based AI, 2023 HotChips 35

- Data movement consumes a lot of energy even for simple computation
- Processing-in-Memory (PIM) / Processing-near-Memory (PNM) technology can reduce energy consumption within a typical memory hierarchy
- PIM/PNM device for each layer must meet specific requirements: bandwidth (BW), power, capacity, etc

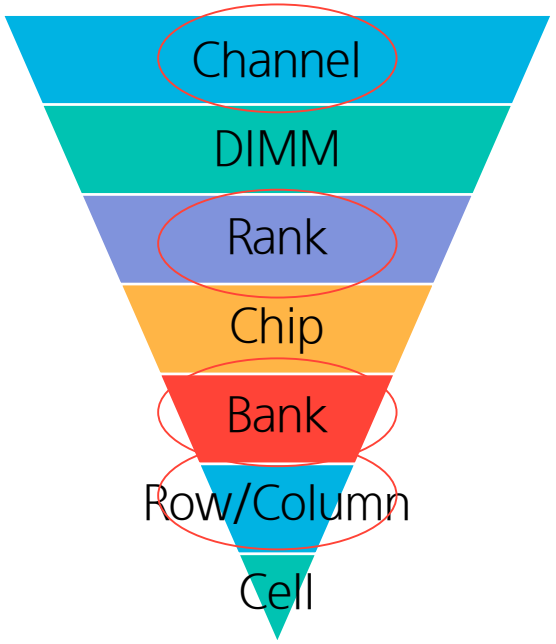


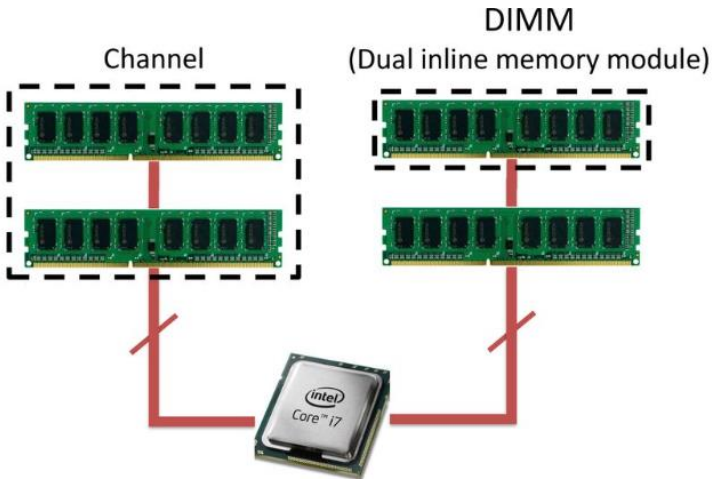
Source: * Y.-H. Eyeriss, 2016 ISCA

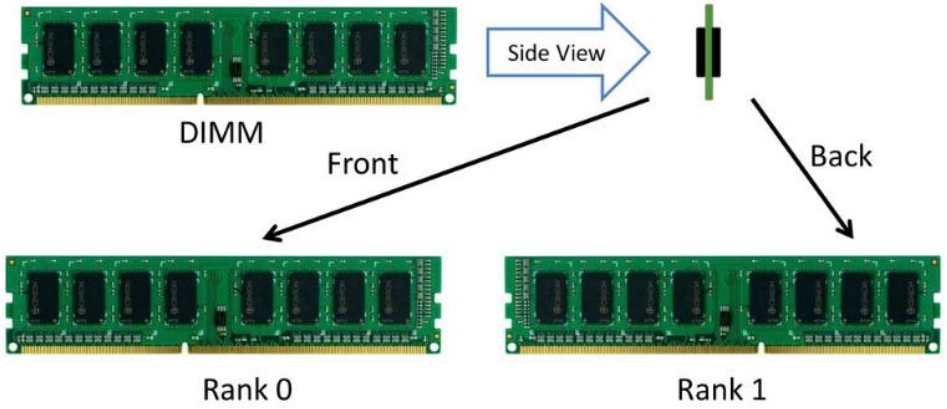


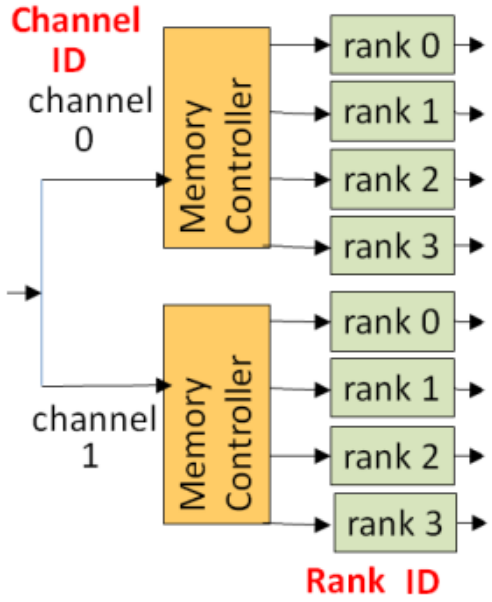
Inside the Memory

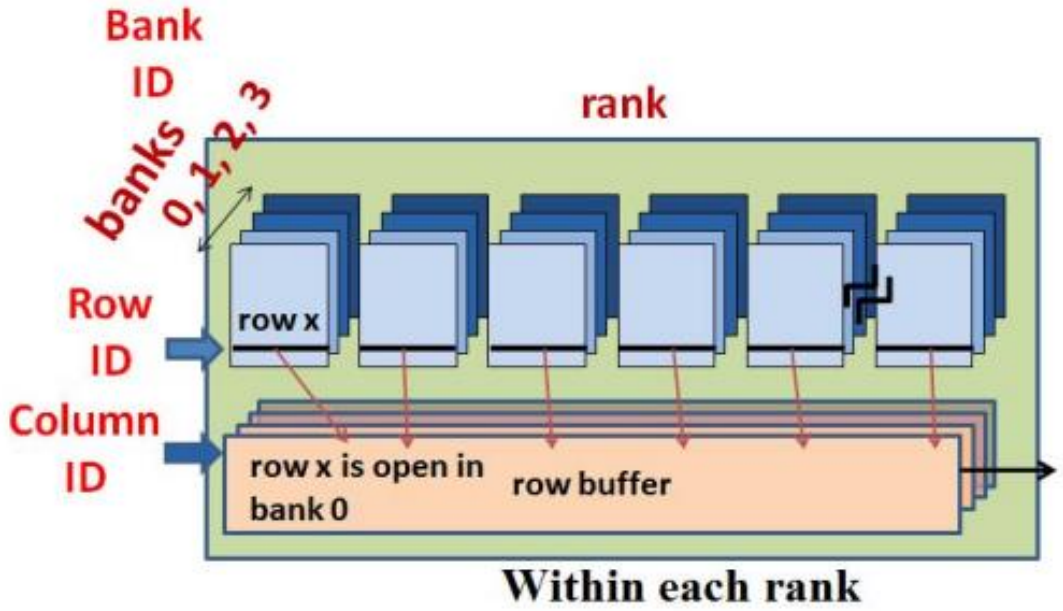
Source: The Main Memory system: DRAM organization: https://slideplayer.com/slide/14569058/
Source: Kumar, Karthik, et al. "Memory energy management for an enterprise decision support system." IEEE/ACM International Symposium on Low Power Electronics and Design. IEEE, 2011.







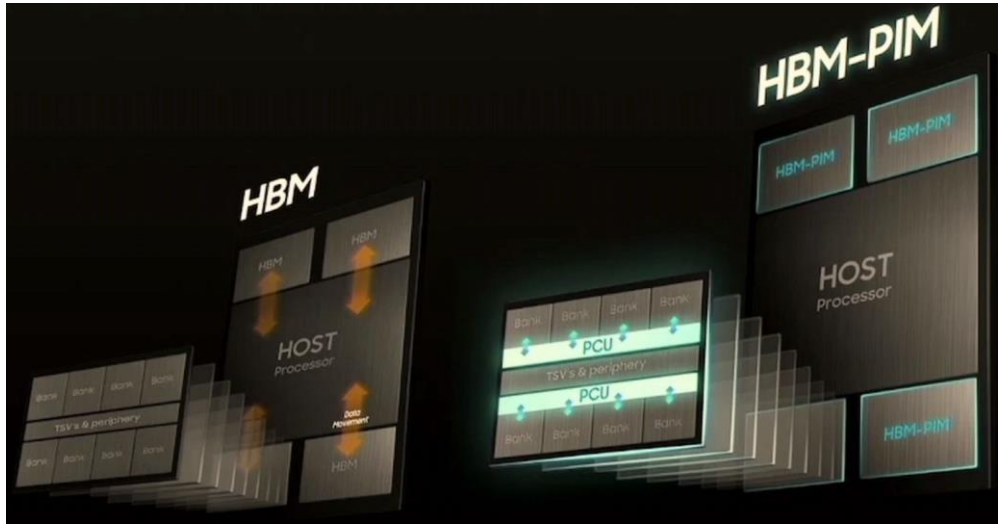




Samsung Aquabolt-XL, System-level 1st PIM memory

Lee, Sukhan, et al. "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," 2021 ISCA

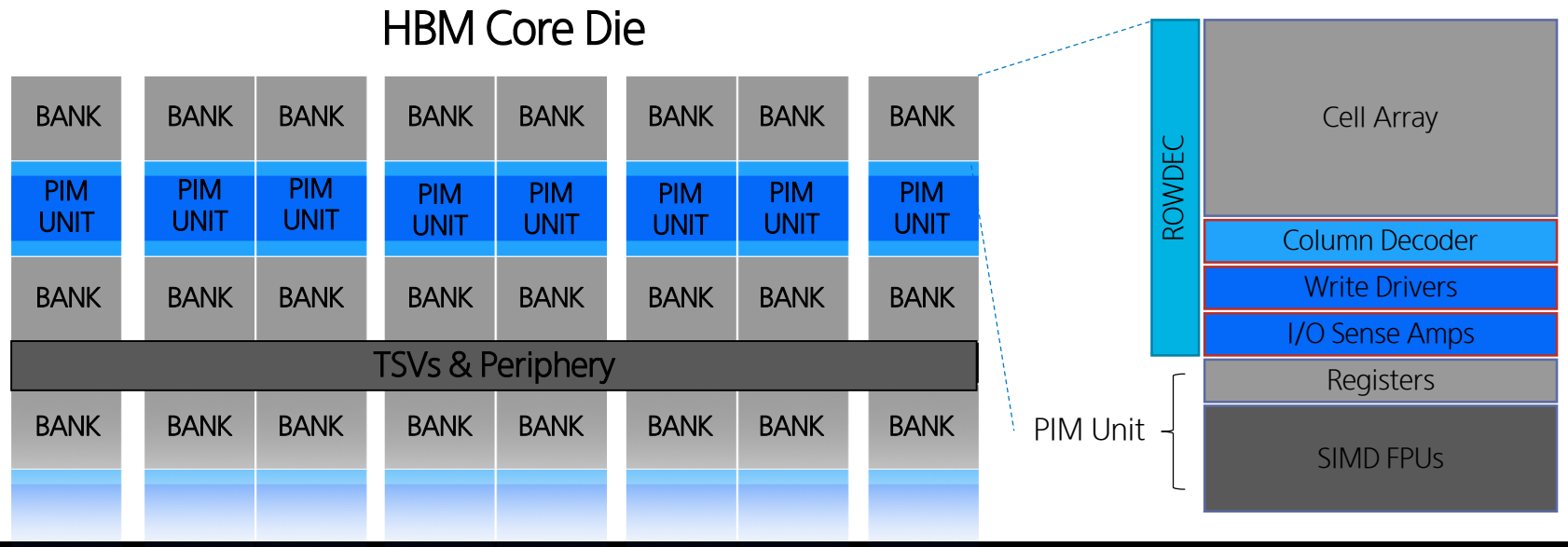
- The first demonstrator vehicle of PIM is based on HBM2 Aquabolt, which is used in leading edge AI and HPC systems
- Improve performance of bandwidth-intensive workloads
- Improve energy efficiency by reducing computing-memory data movement



Aquabolt-XL HBM2-PIM Architecture

Lee, Sukhan, et al. "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," 2021 ISCA

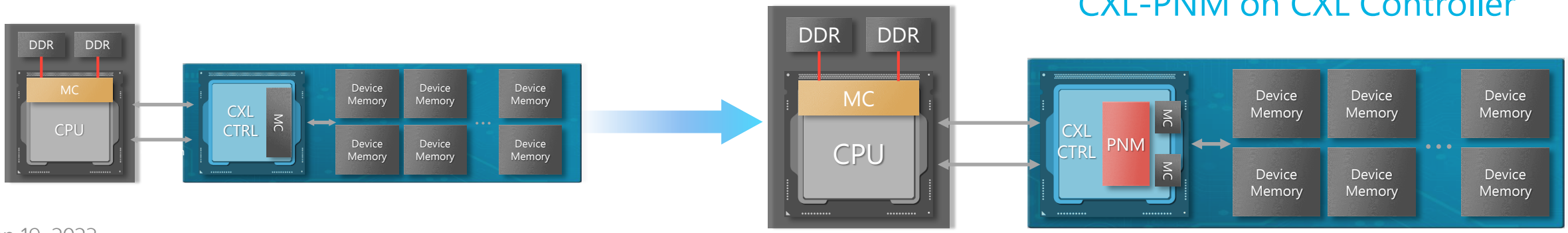
- Placed a PIM execution unit at the I/O boundary of a bank
  - Support both standard DRAM and PIM-DRAM modes for versatility
  - Exploit bank-level parallelism: access multiple banks/FPUs in a lockstep manner
- Maintained the same form-factor and timing parameters
  → Enable to facilitate drop-in replacement
  - DRAM RD/WR command triggers execution of a PIM instruction in PIM mode



CXL-PNM Architecture

Source: Samsung PIM/PNM for Transformer based AI, 2023 HotChips 35

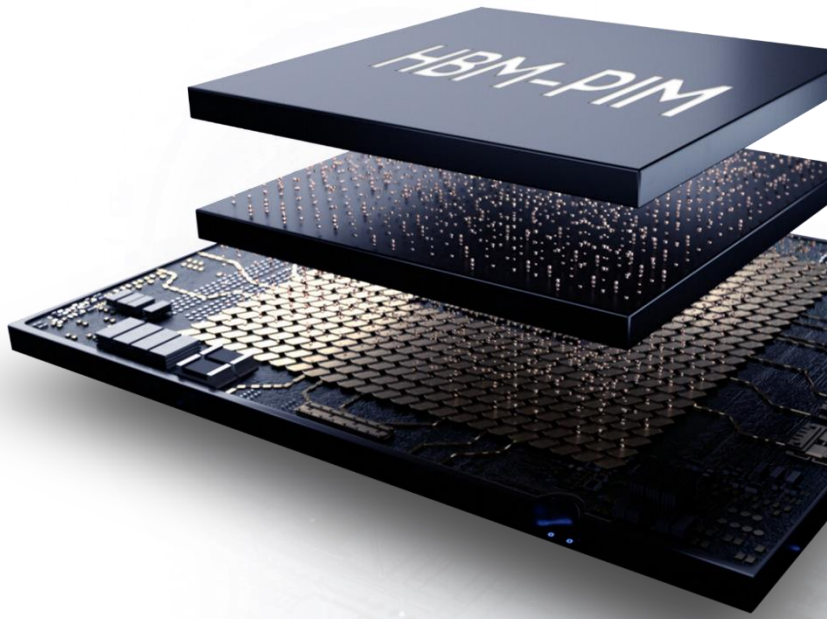
- A CXL-based Processing-near-Memory (PNM) Solution
  - Processing engine on the top of CXL memory module
  - Multiple internal memory channel
  - Energy-efficient data processing with reduced data movement distance
  - CXL-PNM is able to be used for a wider range of systems including AI/ML accelerators
    - CXL-PNM can provide 512GB capacity and 1.1TB/s bandwidth
  - GEMV operation can be fully offloaded to CXL-PNM, fully utilizing DRAM bandwidth and boosting PIM technology (up to 8TB/s)



SAMSUNG

SYCL extensions for PIM/PNM

SYCL™



Design Goal

- Seamlessly integrate PIM/PNM operations into the SYCL programming model to make them available to users in an easy-to-use and comprehensible manner
- Allow combination of xPU and PIM/PNM operations in same SYCL device kernel
- Vendor-neutral design, not specific to the specific PIM / PNM

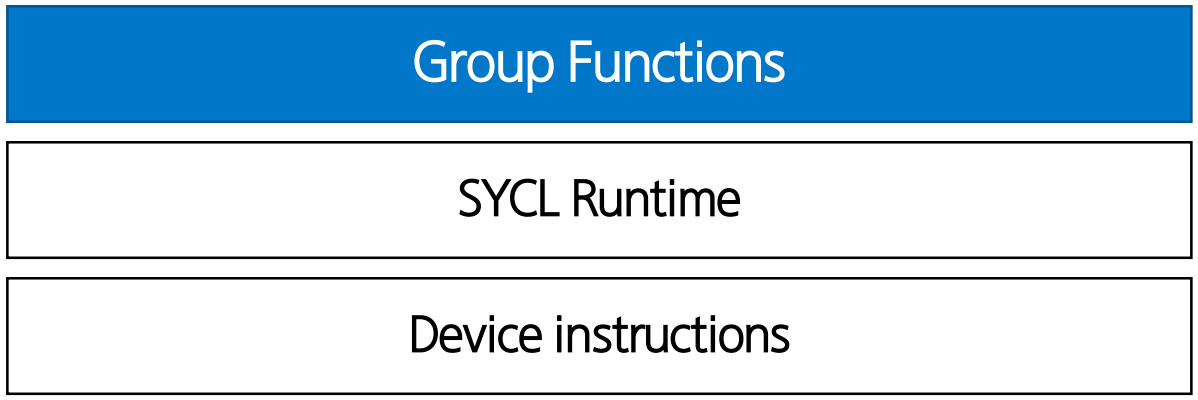
Design

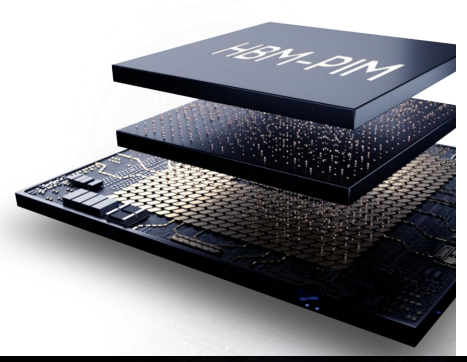
- Vector operations seem like a natural fit for the SIMD, but doubt over its suitability
  - No convergence guarantee
  - Vector size explicitly need to match the alignment → Restrict the portability

- Model a special function unit
  - Aligns with trends to model special functional units inside accelerators
    - Mapping from user code to specialized blocks by compiler desirable, but often not possible
    - Instead, expose programming interface with suitable abstraction for user to leverage specialized blocks
  - Ex. joint_matrix extension: Intel AMX/XMX, Nvidia Tensorcore inside GPU

Design

- Group functions mapping to PIM / PNM
  - Easy to use
  - Easily be combined with other device code in the same SYCL kernel
  - Give the necessary convergence criteria





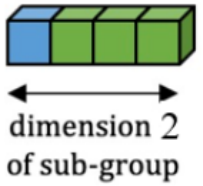
Recap: Group

Source: https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/sycl-thread-mapping-and-gpu-occupancy.html

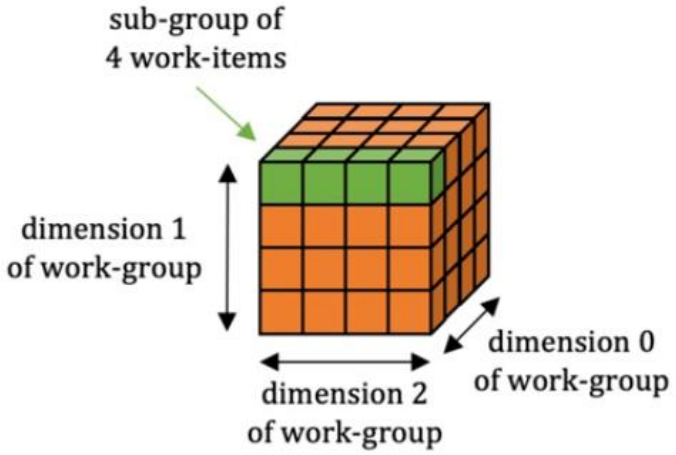
- SYCL basic unit of work: *work-item*
- Work-items are organized in *work-groups*
  - Number of work-items in a work-group specified by user or chosen by SYCL RT during kernel launch
- Work-groups split into one or multiple *sub-groups*
  - Sub-group size determined by device

Work-item

sub-group



Work-group



Recap: Group Functions

- Group functions tied to (sub-) groups of work items
  - Must be encountered in converged control flow
    - If one work-item in group-scope calls function, all work-items in group-scope must call function
    - Call must happen under the same conditions (same branch, same iteration of loop)
  - Can specify additional restrictions

- Ex. group_broadcast, group_barrier, shift_group_right, joint_reduce, reduce_over_group, select_from_group, joint_exclusinve_scan, …

Extensions for PIM

- Semantics similar to std::reduce / std::transform / std::inner_product

```
template<size_t BlockSize, typename Group, typename Ptr, typename BinaryOperation>
std::iterator_traits<Ptr>::value_type joint_reduce(Group g, Ptr data, BinaryOperation binary_op,
                                                   size_t num_blocks);
```

```
template<size_t BlockSize, typename Group, typename Ptr, typename BinaryOperation>
void joint_transform(Group g, const Ptr operand1, const Ptr operand2, Ptr result,
                     BinaryOperation binary_op, size_t num_blocks);
```

```
template<size_t BlockSize, typename Group, typename Ptr>
std::iterator_traits<Ptr>::value_type joint_inner_product (Group g, const Ptr operand1,
                                                           const Ptr operand2, size_t num_blocks);
```

Extensions for PIM

- Extend existing group algorithm

```
template<size_t BlockSize, typename Group, typename Ptr, typename BinaryOperation>
std::iterator_traits<Ptr>::value_type joint_reduce(Group g, Ptr data, BinaryOperation binary_op,
                                                   size_t num_blocks);
```

  - Semantics similar to std::reduce
  - Remove the second pointer which points one element past the end of data
  - Add another overload of the existing joint_reduce group function
    - BlockSize template argument
    - num_blocks argument
    - [data, data + (num_blocks * BlockSize)) use the operator binary_op (sycl::mul / sycl::add)
  - If PIM supports horizontal operations, perform reduction completely in PIM
    - Otherwise, partially perform reduction in PIM and finalize on the underlying device (ex. GPU)

Extensions for PIM

- Add new group algorithm

```
template<size_t BlockSize, typename Group, typename Ptr, typename BinaryOperation>
void joint_transform(Group g, const Ptr operand1, const Ptr operand2, Ptr result,
                     BinaryOperation binary_op, size_t num_blocks);
```

  - Semantics similar to std::transform
  - Map to element-wise addition / multiplication in PIM
  - Preserve the original order of the element
    - [result, result + (num_blocks * BlockSize)) = [operand1, operand1 + (num_blocks * BlockSize)) *binary_op* [operand2, operand2 + (num_blocks * BlockSize))

Example of SYCL Application with PIM


```
1  #include <sycl.hpp>
2  using namespace sycl;
3  int main() {
4      constexpr size_t dataSize = 1280;
5      sycl::half in1[dataSize];
6      sycl::half in2[dataSize];
7      sycl::half out[dataSize];
8      queue q{ pim_selector{} };
9      {
10         buffer<sycl::half> bIn1{in1, range{dataSize}};
11         buffer<sycl::half> bIn2{in2, range{dataSize}};
12         buffer<sycl::half> bOut{out, range{dataSize}};
13         q.submit([&](handler &cgh) {
14             auto accIn1 = bIn1.get_access<access_mode::read>(cgh);
15             auto accIn2 = bIn2.get_access<access_mode::read>(cgh);
16             auto accOut = bOut.get_access<access_mode::write>(cgh);
17             cgh.parallel_for<class Kernel>(nd_range<1>{range<1>{10}, range<1>{2}},
18                 [=](nd_item<1> item)
19             {
20                 auto groupID = item.get_group(0);
21                 auto* in1 = &accIn1[groupID * 128];
22                 auto* in2 = &accIn2[groupID * 128];
23                 auto* out = &accOut[groupID * 128];
24                 joint_transform<64>(item.get_group(), in1, in2, out, sycl::plus<>(), 2);
25             });
26         });
27     }
28     return 0;
29 }
```

Select a SYCL device **supporting PIM operations**

Call a group function **supporting PIM operations**

Extensions for PIM

- Add new group algorithm

```
template<size_t BlockSize, typename Group, typename Ptr>
std::iterator_traits<Ptr>::value_type joint_inner_product (Group g, const Ptr operand1,
                                const Ptr operand2, size_t num_blocks);
```

  - Semantics similar to std::inner_product
  - Can be mapped to MAC/MAD operation to perform multiplication and reduction in PIM

GEMV Example of SYCL Application with PIM


```
1  sycl::half vec[numRows] = ...;
2  /* Matrix stored in *column-major* order */
3  sycl::half mat[numCols * numRows] = ...;
4  sycl::half out[numCols];
5  queue q{pim_selector{}};
6  {
7      buffer<sycl::half, 1> bVec{in1, range{numRows}};
8      /* Matrix stored in *column-major* order */
9      buffer<sycl::half, 2> bMat{in2, range{numCols, numRows}};
10     buffer<sycl::half, 1> bOut{out, range{numCols}};
11     q.submit([&](handler &cgh) {
12         auto accVec = bVec.get_access<access_mode::read>(cgh);
13         auto accMat = bMat.get_access<access_mode::read>(cgh);
14         auto accOut = bOut.get_access<access_mode::write>(cgh);
15         cgh.parallel_for<class Kernel>(nd_range<1>{range<1>{numCols*numRows}, range<1>{numRows}},
16             [=](nd_item<1> item) [[sycl::reqd_work_group_size(8)]]
17         {
18             auto groupID = item.get_group(0);
19             auto vecPtr = &accVec[0];
20             id<2> colIdx{groupID, 0};
21             auto matPtr = &accMat[colIdx];
22             auto result = joint_inner_product<numRows>(item.get_group(), vecPtr, matPtr, 1);
23             if(item.get_group().leader()){
24                 accOut[groupID] = result;
25             }
26         });
27     });
28 }
```

Select a SYCL device **supporting PIM operations**

Call a group function **supporting PIM operations**

Extensions for PNM

- Semantics similar to std::reduce / std::exclusive_scan

```
template<size_t BlockSize, typename Group, typename InPtr, typename OutPtr,
         typename BinaryOperation>
OutPtr joint_exclusive_scan(Group g, InPtr first, InPtr last, OutPtr result,
                            BinaryOperation binary_op, size_t num_blocks);
```

```
template<size_t BlockSize, typename Group, typename InPtr, typename OutPtr,
         typename BinaryOperation>
OutPtr joint_inclusive_scan(Group g, InPtr first, InPtr last, OutPtr result,
                            BinaryOperation binary_op, size_t num_blocks);
```

```
template<size_t BlockSize, typename Group, typename Ptr, typename BinaryOperation>
Ptr reduce_over_group(Group g, Ptr data, BinaryOperation binary_op, size_t num_blocks);
```

Example of SYCL Application with PNM

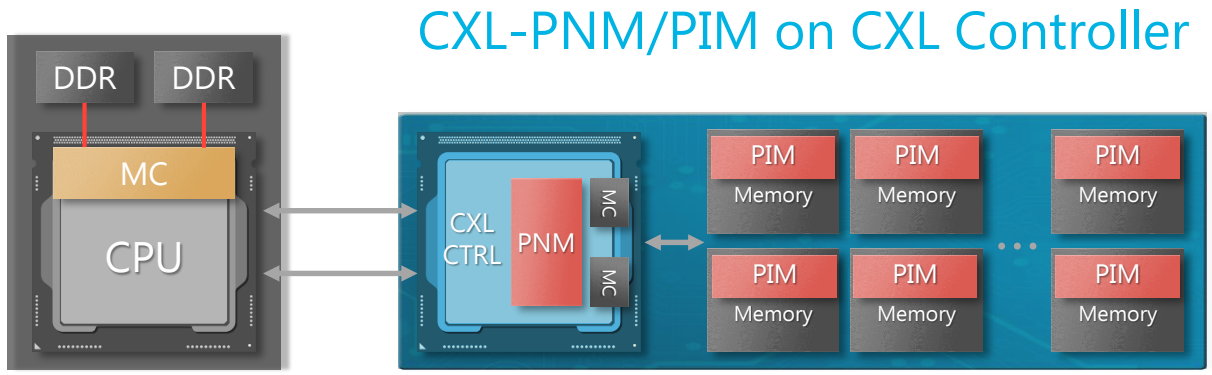

```
1  buffer<int> inputBuf { 1024 };
2  buffer<int> outputBuf { 2 };
3  {
4      // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
5      host_accessor a { inputBuf };
6      std::iota(a.begin(), a.end(), 0);
7  }
8
9  queue myQueue{pnm_selector{}};
10 myQueue.submit([&](handler& cgh) {
11     accessor inputValues { inputBuf, cgh, read_only };
12     accessor outputValues { outputBuf, cgh, write_only, no_init };
13
14     cgh.parallel_for(nd_range<1>(range<1>(16), range<1>(16)), [=](nd_item<1> it) {
15         int partical_sum = reduce_over_group<16>(it.get_group(),
16                                 inputValues[it.get_global_linear_id()], plus<>(), 1);
17         outputValues[0] = partical_sum;
18     });
19 });
20
21 host_accessor a { outputBuf };
22 assert(a[0] == 120);
```

Select a SYCL device **supporting PNM operations**

Call a group function **supporting PNM operations**

Extensions for PIM & PNM

SAMSUNG



- Work-Group Semantics
  - Memory commands generated by PNM
- PIM mode
  - Different PIM blocks can operate independently
  - Can select how many blocks are active
- PNM mode
  - Need to consume results from all PIM blocks
    - Need to be synchronized

Extensions for PIM & PNM

- Potential mapping
  - Every PIM block is one work-item
  - PNM with all attached PIM blocks forms one work-group

- Execution
  - Work-item operations map to PIM operation
  - Group-functions map to PNM operation

Example of SYCL Application with PIM & PNM


```
1  cgh.parallel_for<class Norm>(nd_range<1>{N}, [=](nd_item<1> item)
2  {
3      auto partialSum = 0;
4      for(size_t j = 0; j < N; j++)
5          partialSum += A[j];
6
7      auto sum = reduce_over_group (i.get_group(), partialSum, std::plus<>());
8
9      A[i] /= sum;
10 });
```

Specify the number of PIM blocks

Call a group function, executing on PNM
- Compute overall result

Execute on PIM
- Compute partial result in every PIM block
- Compute element-wise operation

Conclusion

- SYCL Support for PIM / PNM
  - Extend the existing group functions
  - Add new group functions similar to std library

Thank you