SAND2022-13103C

Data Transfers and Host/Device Communication using OneAPI for FPGA

A Presentation for the Intel eXtreme Performance Users Group

Phillip Lane, Christopher Siefert, Stephen Olivier, *Clayton Hughes*, Gwendolyn Voskuilen, Kevin Pedretti, and James Elliott

IXPUG 2022, Programming Track, 11:30 AM CST

Argonne National Laboratory

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Outline

- Why FPGA?
- OneAPI and Data-Parallel C++
- Unified Shared Memory Performance
- Kernel Launch Latencies
- Denial-of-Service Vulnerability
- Case Studies

About FPGAs

- CPUs and GPUs: **stored-program computer**
  - Program stored in memory; instructions executed by dedicated fetch/decode/execute hardware
- FPGAs: reconfigurable hardware, **spatial computing**
  - At compile time, code is translated into a physical hardware layout
  - FPGA reconfigures itself at runtime
  - Program translated to arithmetic look-up tables (ALUTs) and block RAM (BRAMs)
- The FPGA die is connected to more traditional RAM components
  - Usually, DDR or HBM

Benefits? Drawbacks?

- No overhead from fetch/decode
  - Able to sustain much higher FLOPs per clock cycle than CPUs or GPUs
- Early adopter of newer memory technologies
  - High-bandwidth memory (HBM)
- Low power usage
  - Cost: much lower clock speed, maxing out at ~400 MHz
- Good branching support
  - Unlike GPUs (or CPUs if the branch pattern can't be predicted)
- Painfully long synthesis times
  - Can take 2-6 hours in our experience

Outline

- Why FPGA?
- OneAPI and Data-Parallel C++
- Unified Shared Memory Performance
- Kernel Launch Latencies
- Denial-of-Service Vulnerability
- Case Studies

OneAPI

- Intel OneAPI is a suite of tools for heterogeneous programming
  - Such tools include the Intel Fortran, C, and C++ compilers, the Data-Parallel C++ compiler, VTune, the Math Kernel Library (OneMKL), among others
- Data-Parallel C++ is an implementation of SYCL for heterogeneous, single-source programming
  - Write once, run anywhere (in theory)
- DPC++ is capable of targeting manycore CPUs, GPUs, and FPGAs

Data-Parallel C++ for FPGA

- Data-Parallel C++ (DPC++) is commendable for making high-level synthesis (HLS) for FPGA more accessible than ever
- Built off OpenCL, abstracts away many of the more difficult or tedious requirements present in OpenCL development
- However, high-performance software needs evaluation
    - Software needs to be robust, performant, and accessible
- We analyze many of the phenomena related to data transfer using DPC++ on a Bittware Stratix 10 MX FPGA

Outline

- Why FPGA?
- OneAPI and Data-Parallel C++
- **Unified Shared Memory Performance**
- Kernel Launch Latencies
- Denial-of-Service Vulnerability
- Case Studies

Phenomenon 1 – USM Performance

- Our FPGA supports Explicit Unified Shared Memory (Explicit USM)
- Explicit USM allows developers to manually allocate and move memory between the device and host
  - Similar to the CUDA programming model
  - In contrast to the buffer/accessor model
- We've noticed that the use of explicit USM can slow performance by up to 4x on our FPGA

```
template<class T>
class Vector_usm {
public:
  Vector_usm(int mysize) : size(mysize) {
    host_buf = (T*)aligned_alloc(64, sizeof(T) * size);
    buf      = sycl::malloc_device<T>(size, *workq);
  }

  ~Vector_usm() {
    sycl::free(buf, *workq);
    workq->wait();
    free(host_buf);
  }

  void d_to_h() {
    workq->memcpy(host_buf, buf, size*sizeof(T));
    workq->wait();
  }

  void h_to_d() {
    workq->memcpy(buf, host_buf, size*sizeof(T));
    workq->wait();
  }
```

Underlying Cause

- USM isn't inherently bad
  - When written properly, it is faster than buffer/accessor
- USM chokes on **frequent, small transfers**
  - Buffer/accessor does, too, but to a less severe degree
- Transfer 80 kB of data 100,000 times… averages 30 seconds, up to **2 minutes**
  - Buffer/accessor averages 18 seconds, up to 30 seconds
- Transfer 8 MB of data 1,000 times… consistently takes 4.1 seconds using USM
  - Buffer/accessor consistently takes 4.3 seconds!

Takeaway

- Prefer **larger, fewer transfers** over smaller, frequent transfers
    - Problem is not limited to just USM, happens to buffer/accessor too
- If small, frequent transfers are necessary, prefer buffer/accessor
- Peak performance of both paradigms are roughly equal!

- Would really like this performance limitation patched

Outline

- Why FPGA?
- OneAPI and Data-Parallel C++
- Unified Shared Memory Performance
- Kernel Launch Latencies
- Denial-of-Service Vulnerability
- Case Studies

Motivation

- Many of our apps at Sandia are based on spawning a **high number** of **lightweight** device kernels
  - Kokkos is built with this design principle in mind
- Launch latencies play a critical role in the execution time of these types of applications
- The lower, the better!

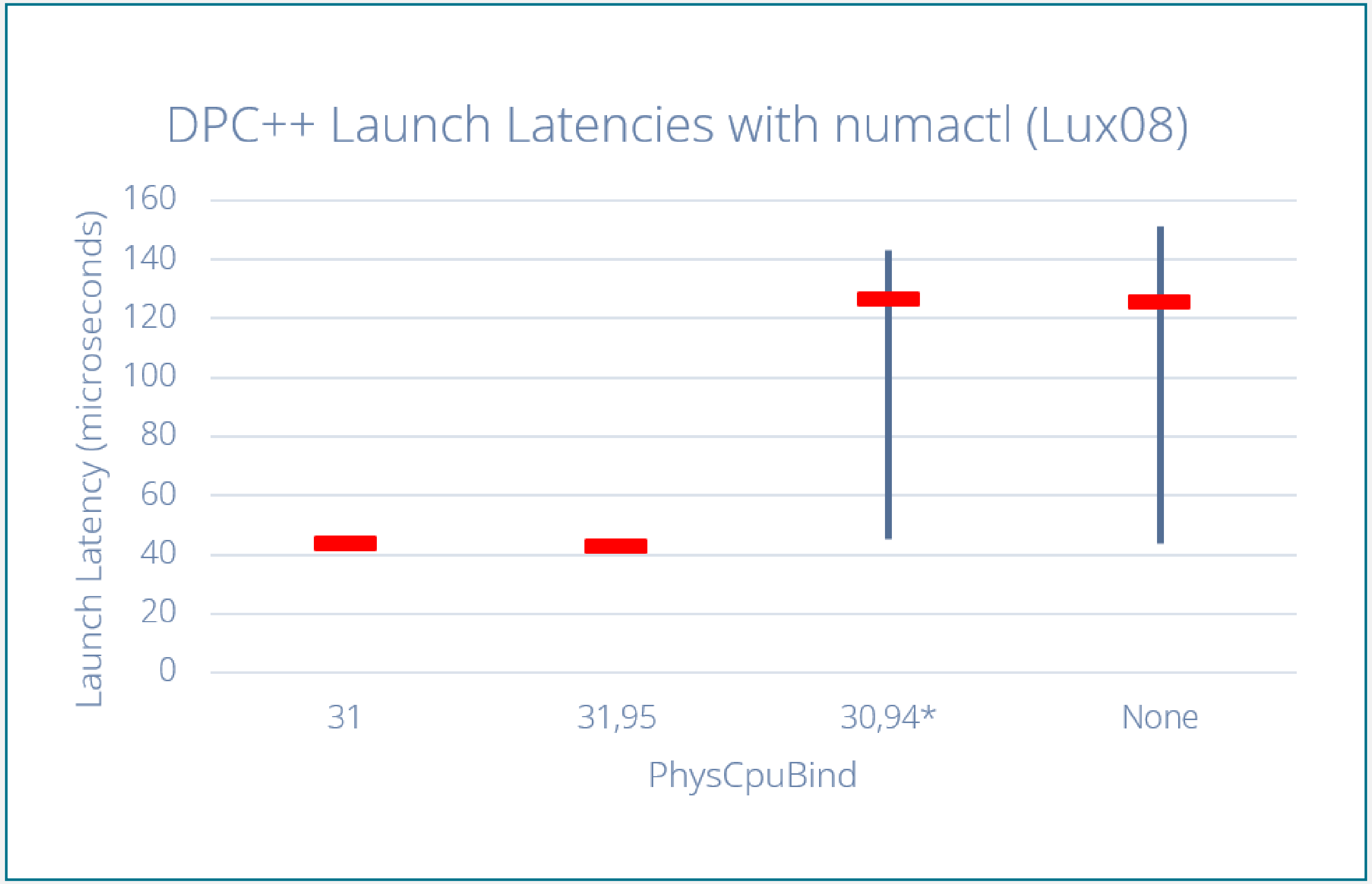
Test Methodology

- Synthesize two empty kernels
  - FPGA kernel with nothing to do (zero code, zero data transfers)
  - One in DPC++, one in OpenCL
- Call FPGA kernel, then immediately synchronize
  - Timing starts before kernel launch and ends after synchronization
- Repeat 100,000 times
- Numactl used in some tests
  - Linux tool to bind a process to physical thread/core

Systems Tested

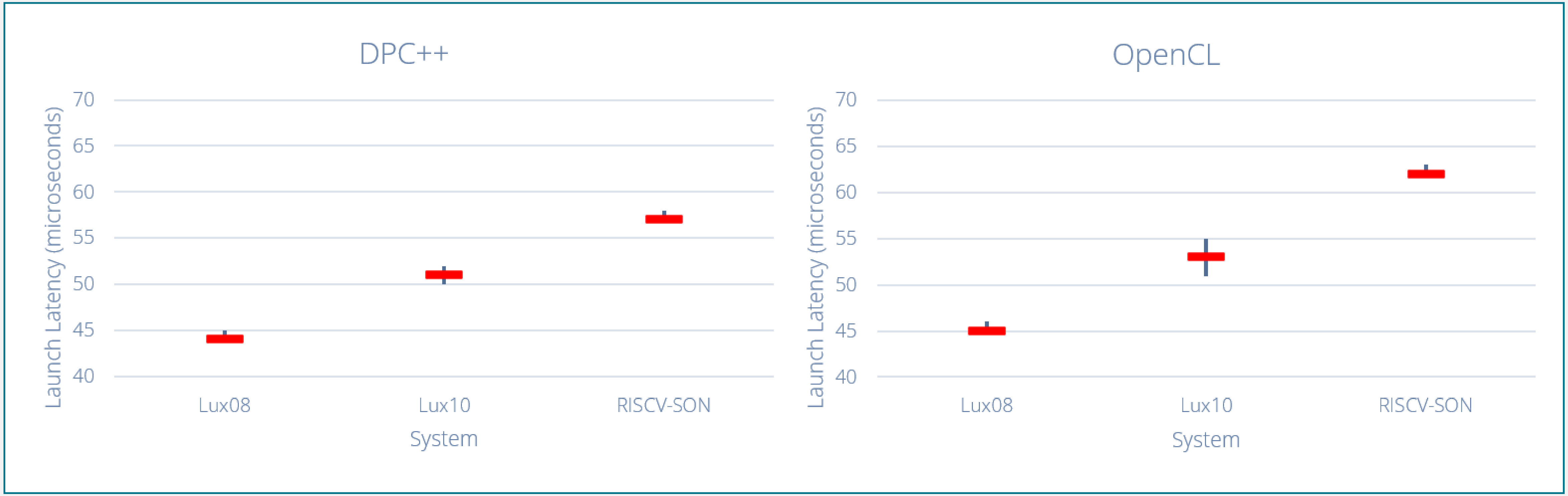
- Lux08
  - Node in Lux cluster supporting 2x Intel Xeon Platinum 8352Y CPUs
  - Dynamic frequency scaling **enabled, performance governor**
- Lux10
  - Node in Lux cluster supporting 2x Intel Xeon Platinum 8352Y CPUs
  - Dynamic frequency scaling **disabled**
- RISCV-SON
  - Node supporting 1x Intel Xeon Silver 4216 CPUs
  - Dynamic frequency scaling **enabled, powersave governor**
- All systems use the same Bittware Stratix 10 MX FPGA with Bittware driver

Using numactl to Bind to CPUs



*all other cores were tested, results approximately the same

DPC++ vs. OpenCL on Different Systems



Lux08 – Performance governor
Lux10 – Disabled governor
RISCV-SON – Powersave governor

Takeaways

- For the Bittware Stratix 10 MX, binding to the **highest physical core** on **socket 0** was crucial for performance
  - Lowest variance and average launch latency
- System configuration can play a significant part in launch latency
  - The performance frequency governor seems to help
- DPC++ usually **faster** than OpenCL by 2-9%

An Aside

- We noticed that if the DPC++ kernel had an accessor, there was a **20 microsecond overhead**, even if no data is being transferred
    - Suppose data is already on device and is in sync with host
    - 20 us overhead to every kernel launch to do this "check"
- Does this scale based on number of accessors? Does USM carry the same penalty?
    - Likely: yes and no respectively… left for future testing

Outline

- Why FPGA?
- OneAPI and Data-Parallel C++
- Unified Shared Memory Performance
- Kernel Launch Latencies
- Denial-of-Service Vulnerability
- Case Studies

Direct Memory Access

- Direct memory access (DMA) allows much faster transfers of data between the host and device
- Requires arrays to be aligned to a 64-byte boundary
- Basic routine for DMA transfer
  - OS will pin relevant pages on CPU side
  - DMA transaction will occur without risk of pages migrating around RAM
  - Once transaction is over, OS should un-pin pages

Denial of Service

- If DMA is used for a device-to-host transfer, there is the possibility of a kernel panic
    - OS unable to pin the page(s) in memory
    - OS alerts FPGA driver, driver doesn't catch error, so... kernel panic
    - Has brought down several of our nodes
- Fixing not as easy as rebooting node
    - By default, FPGA boots into unusable state
    - Reboot process on next slide

Reboot Process

```
setpci -s 4b:0.0 ECAP_AER+0x08.L=0xFFFFFFFF
setpci -s 4b:0.0 ECAP_AER+0x14.L=0xFFFFFFFF
setpci -s b1:0.0 ECAP_AER+0x08.L=0xFFFFFFFF
setpci -s b1:0.0 ECAP_AER+0x14.L=0xFFFFFFFF

setpci -s 4a:02.0 ECAP_AER+0x08.L=0xFFFFFFFF
setpci -s 4a:02.0 ECAP_AER+0x14.L=0xFFFFFFFF
setpci -s b0:02.0 ECAP_AER+0x08.L=0xFFFFFFFF
setpci -s b0:02.0 ECAP_AER+0x14.L=0xFFFFFFFF

echo 1 > /sys/bus/pci/devices/0000:4b:00.0/remove
echo 1 > /sys/bus/pci/devices/0000:b1:00.0/remove

quartus_pgm -c 1 -m jtag -o "p;blinky.sof"

quartus_pgm -c 1 -m jtag -o "p;base.sof"
```

Suspect Cause

- DPC++ is not good with requesting DMA-ready memory
- Only one engineer on our team has caused this kernel panic
  - Used `posix_memalign` for aligned memory
  - All other engineers used `sycl::malloc`
- Suspect `posix_memalign` is not working nicely with DMA
  - Unconfirmed hypothesis… left for further testing
- Despite unconfirmed hypothesis, recommend `sycl::malloc`

Outline

- Why FPGA?
- OneAPI and Data-Parallel C++
- Unified Shared Memory Performance
- Kernel Launch Latencies
- Denial-of-Service Vulnerability
- Case Studies
    - STREAM
    - Sparse Matrix-Vector Multiplication

STREAM

- STREAM is a collection of four memory-intensive benchmarks
- We implement COPY to test raw throughput of the HBM channels
    - COPY – `a[i]=b[i]`
    - `a` and `b` are both kept within the same memory channel
    - Transfer 96 MB of memory per channel and time kernel execution

Our FPGA's Memory Hierarchy

- Our Stratix 10 MX has 32 discrete HBM memory ports
- Data is **not interleaved across ports** per Bittware's design decisions
    - Supposedly data interleaving would make clock speed too slow, so they disable it outright
- Sustained total bandwidth through all ports theoretically 410 GB/s
- Since no data interleaving, template metaprogramming used to generate 32 kernels
    - We test each HBM port individually, then total combined throughput

Test Results

```
h.parallel_for<>(range<1>(size / sizeof(uint64_t)), [=](id<1> i)
                [[intel::num_simd_work_items(NUM_SIMD_WORK_ITEMS),
                sycl::reqd_work_group_size(1,1,REQD_WORK_GROUP_SIZE)]]{
        to[i] = from[i];
});
```

- Per-channel throughput: ~10 GB/s
- Combined throughput: ~316 GB/s
- Roughly 31.6x speedup
    - Perfect is 32x
    - Very good speedup
- Only 77% of theoretical throughput

Lessons Learned

- Each memory load is 256-bit
  - Unused data is **discarded…** not cached, unless explicitly requested
  - Sequential memory access with 64-bit stride = 4x more memory transfers than necessary
- Solution: **vectorization**
  - Manual loop unrolling works!
  - `#pragma ivdep` works!
  - SYCL SIMD attributes work!
  - Simple for loop **does not work—very slow!**

Outline

- Why FPGA?
- OneAPI and Data-Parallel C++
- Unified Shared Memory Performance
- Kernel Launch Latencies
- Denial-of-Service Vulnerability
- Case Studies
    - STREAM
    - Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication

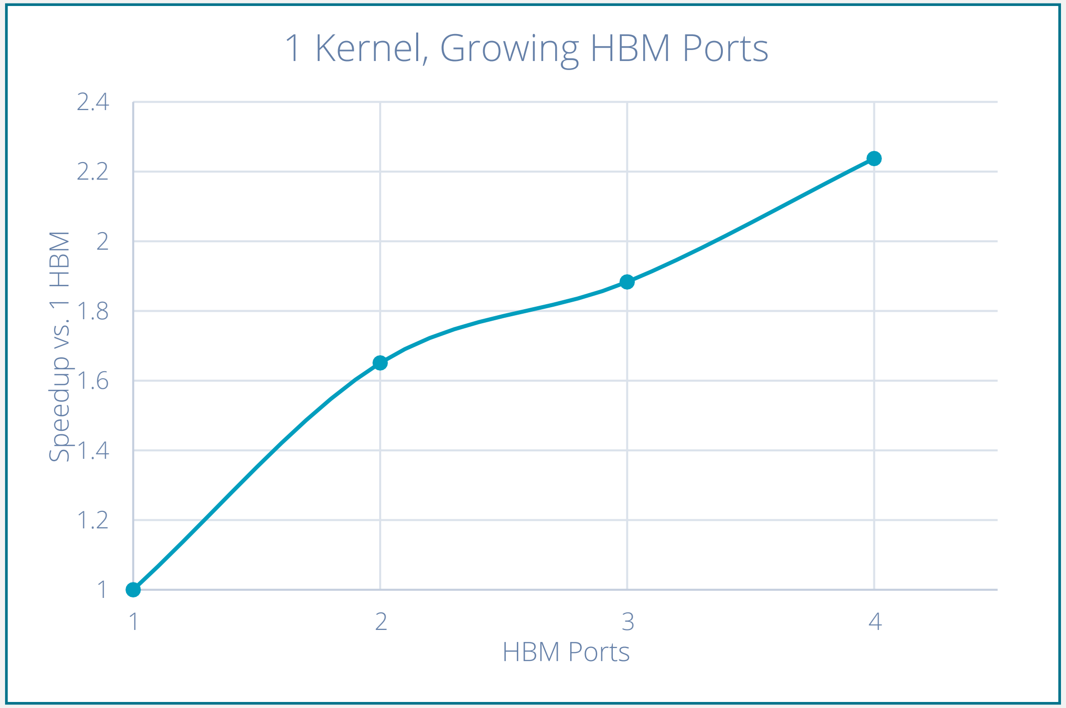
- Sparse matrix-vector multiplication (SpMV) is a common operation in many iterative solvers
  - Conjugate Gradient (CG) and generalized minimal residual method (GMRES)
  - Useful for solving partial differential equations (PDEs)
- Difficult kernel to optimize due to irregular memory access
  - GPU can help speed up SpMV due to significantly higher memory bandwidth
- On both CPU and GPU, majority of time is spent awaiting memory loads
  - 3 memory loads and 1 store must occur **per multiply-add (FMA) operation**

Using SpMV to Analyze DPC++ Programming Practices

- We implement a simple SpMV and tinker with hardware duplication
- Hardware duplication can increase performance by increasing data parallelism in your design
- Try increasing load-store units (LSUs) for more bandwidth
- Try duplicating the actual kernel for more compute power

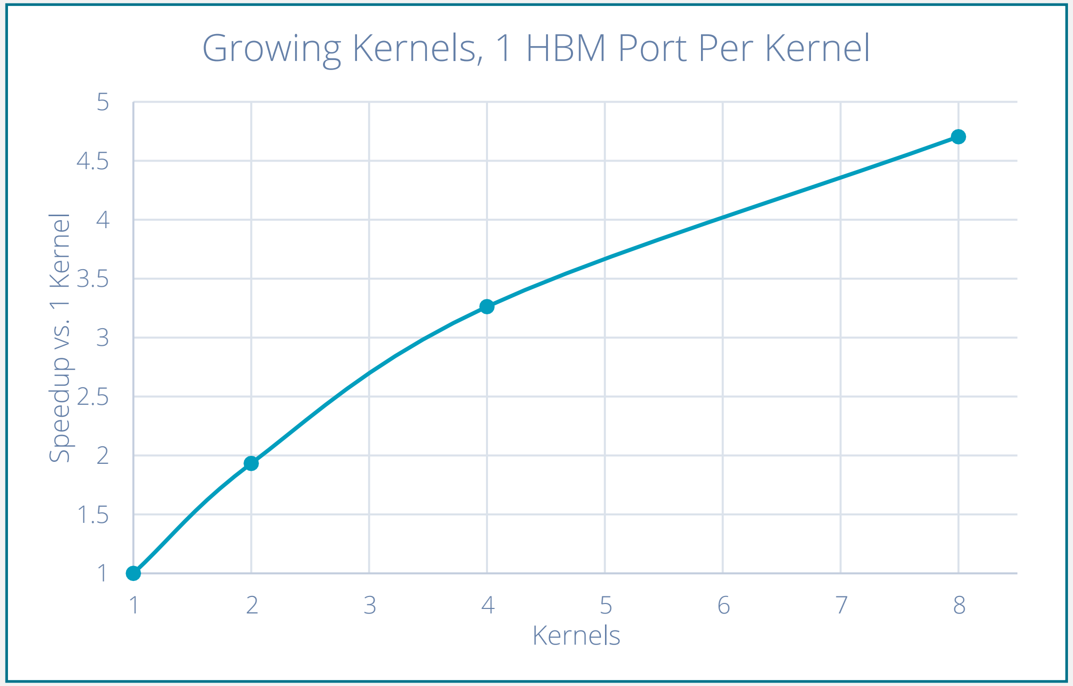
1 Kernel, Growing HBM Ports

- In this test, we increase the number of HBM ports available to the kernel
- Because we can't interleave data, arrays are "assigned" to HBM ports
  - Do this intelligently to balance load as much as possible
- Using 4 HBM ports we achieve a maximum speedup of 2.24x



Growing Kernels, 1 HBM Port Per Kernel

- In this test, we physically duplicate the entire execution kernel using template metaprogramming
- We give each kernel its own HBM port
  - Making all kernels use the same HBM port was tested—memory bound, so there was no speedup
- Using 8 kernel duplicates, we achieve a maximum speedup of 4.7x



Lessons Learned

- Kernel duplication should be used where possible to increase data parallelization in your kernels
  - We encountered area limits with 8 kernels
- Use as much memory bandwidth as possible
  - Combining 8 kernels + 4 HBM ports per kernel, achieved speedup of 7.5x
  - Roughly 4x faster than a single Skylake core running SpMV
- Not confident that we've extracted maximum performance at kernel level
  - We are disappointed that we couldn't outperform a whole Skylake CPU

Conclusion

- Lots of useful information learned through trial and error
  - Would like better documentation of these "best practices"
- DPC++ compiler not where we want it to be
  - Automatically detect SIMD potential in loops
  - Better ways to incorporate hardware duplication
  - Better optimization?

- However, we have not had to use Verilog at all... kudos to Intel
- Huge thanks to Gwen's LDRD team, which has allowed this research

Questions?

Thank you!