V1-24: High-Productivity Computing in Heterogeneous Systems

Mission-Critical Computing
NSF CENTER FOR SPACE, HIGH-PERFORMANCE, AND RESILIENT COMPUTING (SHREC)

SHREC Annual Workshop (SAW23-24)

University of Pittsburgh · BYU Brigham Young University · Virginia Tech · UF University of Florida

January 17-18, 2024

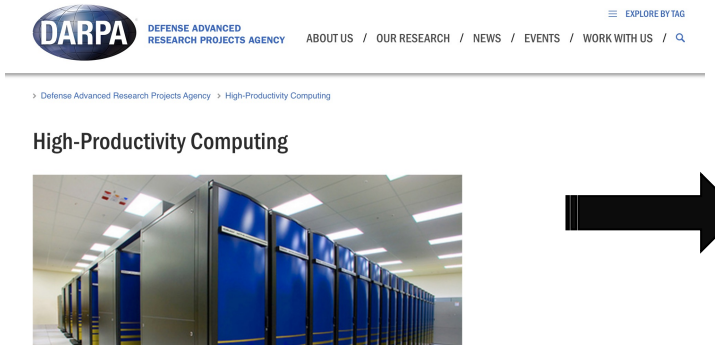
Faculty: Wu Feng

Students: Patrick Dewey, Atharva Gondhalekar, Ritvik Prabhu, Rashmi Ravindranath, Paul Sathre, Frank Wanye, Mukund Yadav

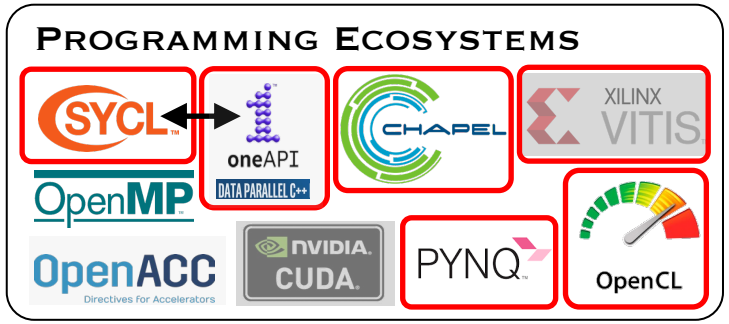
Number of requested memberships ≥ 6

Goal

- Enable ***high-productivity computing*** in ***heterogeneous*** computing systems: CPU + {CPU, GPU, FPGA, TPU, …}
  - Similar to DARPA High-Productivity Computing Systems program for homogeneous systems (e.g., Chapel, Fortress, X10) but for heterogeneous systems (e.g., Chapel, OpenCL, SYCL, oneAPI, VITIS)
  - Preferred Vehicle: Modern, Open Standard Languages & Runtimes
  - Case Studies: Applications and Benchmarks, e.g., Berkeley Dwarfs → OpenDwarfs (@ VT)

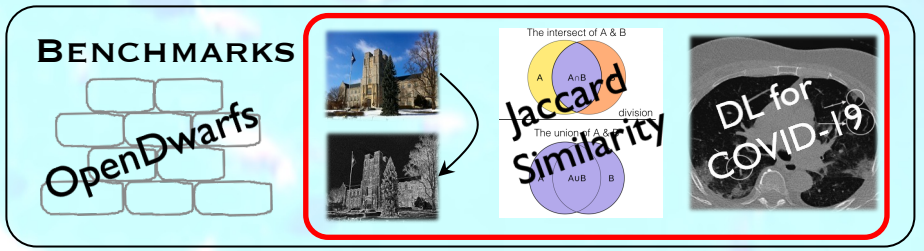


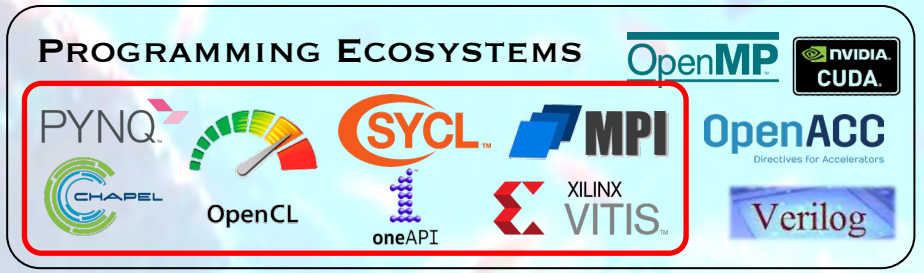
Write ONCE, run ANYWHERE!



Background & Motivation

- Extend our R&D to create and analyze an ecosystem of *high-productivity tools, environments, and benchmarks for heterogeneous computing*



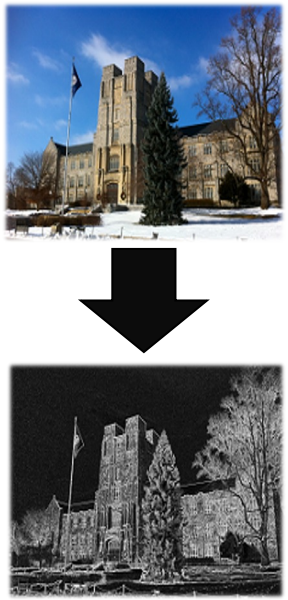


- **Challenges**: How to ***productively ...***
  - Program an application so it runs on many platforms?
  - Evaluate a processor architecture & compare it to others?
  - Develop back-end optimizations & know that they will work well?

} Application-dependent

Recent Work: Performance & Productivity

- Sobel Filter on Intel Arria 10, AMD Alveo U250, and NVIDIA RTX 3090



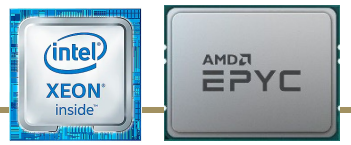
Sobel Filter on 3840 × 2160 Image	Device								
	Intel Arria 10 (FPGA)			AMD Alveo U250 (FPGA)			NVIDIA RTX 3090 (GPU)		
Language	fps	SLOC	Dev Time (hrs)	fps	SLOC	Dev Time (hrs)	fps	SLOC	Dev Time (hrs)
Verilog	132.6	429	305	*Not implemented in Verilog*			*Not functional on GPU*		
OpenCL	85.6	270	50	6.6*	275	–	141.4	254	–
oneAPI → SYCL	21.4	139	20	*No support for oneAPI*			133.1	135	–

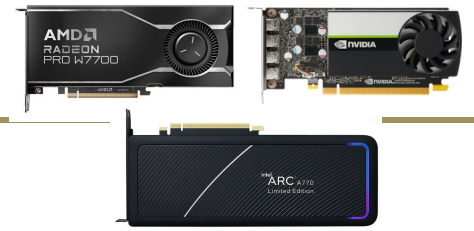
fps: frames per second

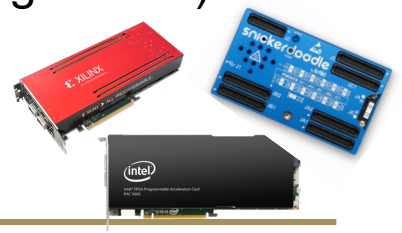
OpenCL & SYCL: Write once, run anywhere

* Evaluated the ***same*** OpenCL kernel written for Arria 10 on U250 without any vendor-specific optimizations

- Rigorous ***Performance & Productivity*** Evaluation of ***Representative Apps*** (FFT, Jaccard similarity, biconjugate gradient stabilized method – BiCGSTAB, and graph algorithms) in ***Different Languages*** on ***Different Devices*** (CPUs, GPUs, and FPGAs)



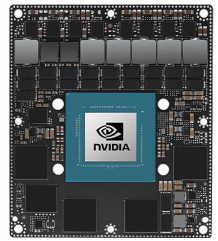


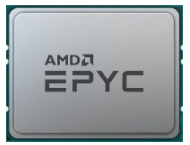


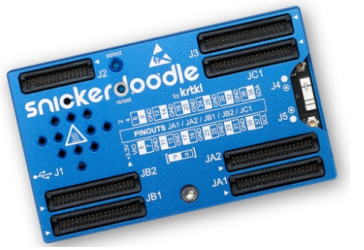
V1

Approach

- Implement a diverse set of ***application benchmarks***
  - Regular vs irregular. Floating point vs integer. CPU- vs memory-intensive
- Characterize the ***productivity*** of a heterogeneous system
  - Kernel Development Time (KDT) → Wall Clock Time
  - Source Lines Of Code (SLOC) and Code Convergence (CC)
- Characterize the ***performance-vs-productivity*** tradeoff
  - Performance Portability (Φ)
  - Performance-Productivity Product (Π)
- ***Identify the best platform*** and associated ecosystem for productivity, performance, or both (across many apps)
- ***Enable further high-productivity research***: *automated co-scheduling at runtime* and *auto-generation of translators*







Open Source

Closed Source

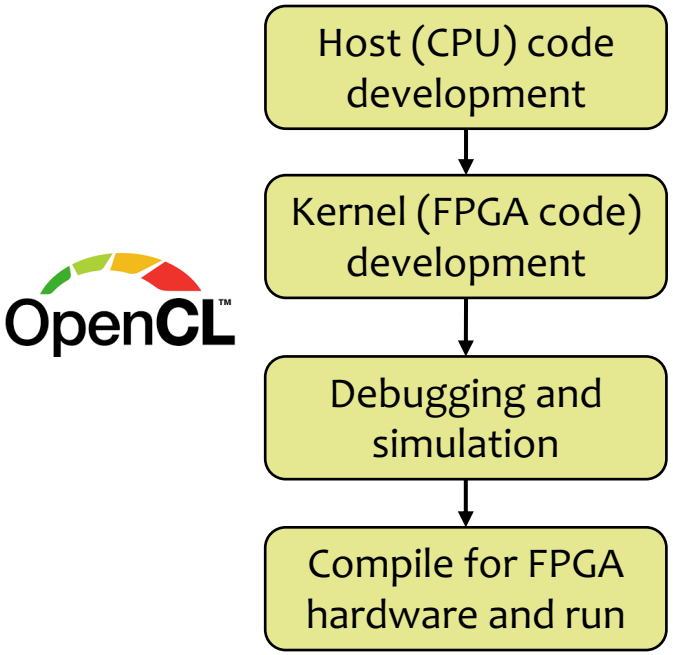
Proposed Tasks for V1-24

(Memberships: Mandatory + Optional, e.g., 2+1)

- Task 1: FPGA Productivity and Performance (Speed/Space/Power) (2+1)
- Task 2: GPU Productivity and Performance (Speed/Power) (2+1)
- Task 3: Chapel GPU Parallel & Distributed Programming: Study, Analysis, Outreach (2+1)
- Task 4: Simultaneous Co-scheduling of Heterogeneous Devices: CPU+GPU+FPGA (0+2)
- Task 5: Auto-Generation of Source-to-Source Translators (0+2)
  - One of {CUDA-to-OpenCL, CUDA-to-OpenMP, CUDA-to-Chapel, CUDA-to-SYCL, …}
  - Baseline: Update of CU2CL: CUDA-to-OpenCL Source-to-Source Translation

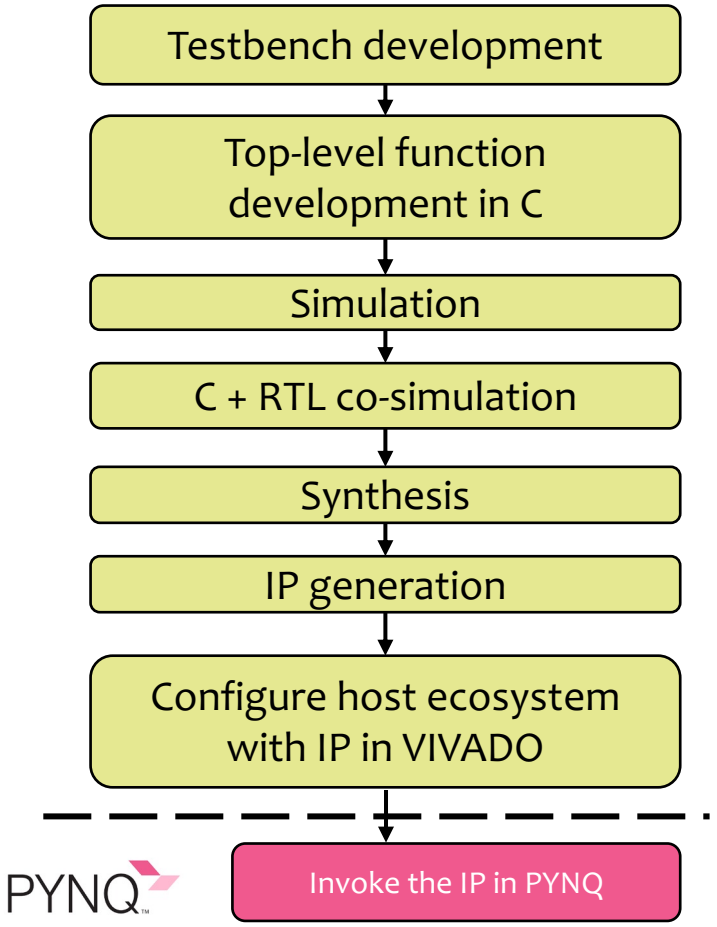
Task 1: FPGA Productivity and Performance (Speed/Space/Power)

OpenCL Development Flow

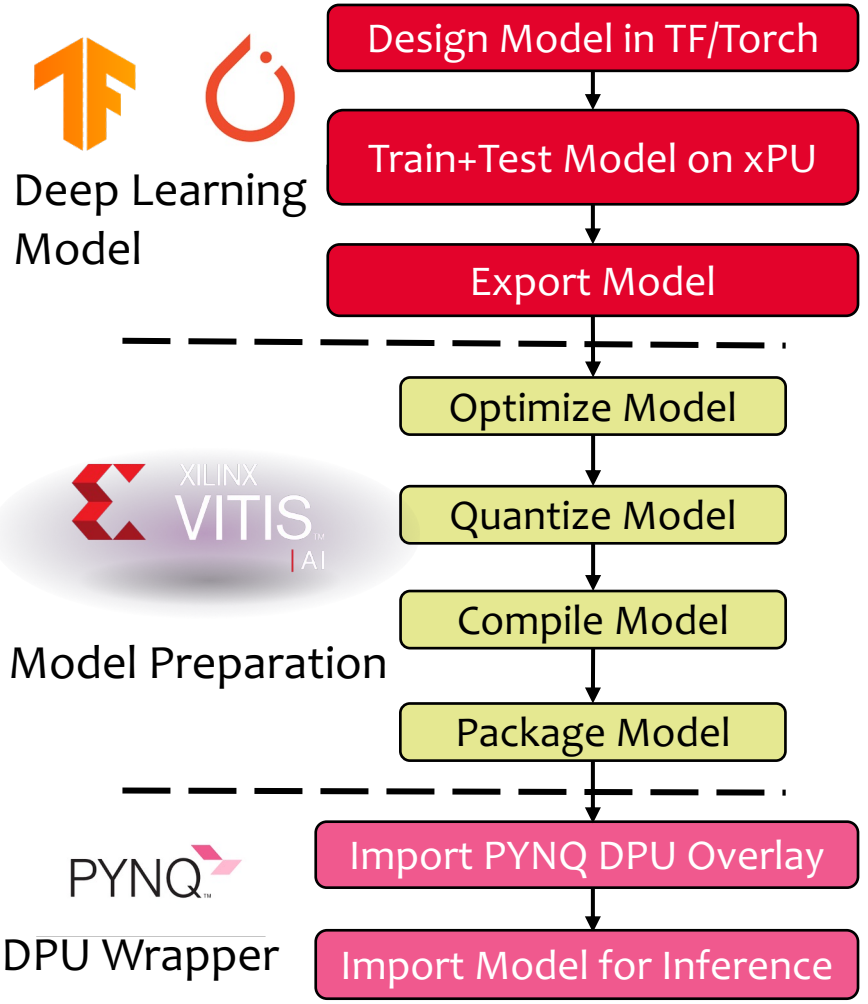


OpenCL dev → less manual intervention needed compared to VITIS HLS and VITIS AI

VITIS HLS Development Flow



VITIS AI Development Flow



V1

Task 1: FPGA Productivity and Performance (Speed/Space/Power)

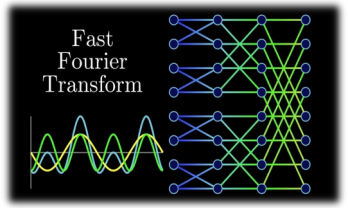
- Motivation
  - Multiple design-entry options (e.g., OpenCL, VITIS HLS, VITIS AI)
  - Performance & productivity tradeoffs of FPGA languages and tools relatively unknown
- Approach
  - Develop target apps via open-source & vendor toolchains
  - Evaluate performance & productivity of FPGA design options
- Tasks 1a, 1b, 1c: Productivity & performance of …
  a) Fast Fourier transform (FFT) with custom-length input via VITIS HLS
     - Target: Snickerdoodle board (ARM Cortex-A9 CPU + AMD Artix-7 FPGA)
  b) Existing reinforcement learning (RL) model via VITIS AI toolchain
     - Target: Alpha Data PA100 (2x ARM Cortex-A72, 2x ARM Cortex-R5, 400x AI Engines)
  c) FFT & RL via open-source toolchains, e.g., SYCL (currently unsupported)
  d) "To-be-provided" RL model for inference

FPGA Vendor and Type	Language & Tools		
	OpenCL	VITIS HLS	VITIS AI
AMD server-grade	✓	✓	✗
AMD embedded	✗	✓	✗
AMD with deep-learning processor units (DPUs)	✗	✓	✓
Intel server-grade	✓	✗	✗

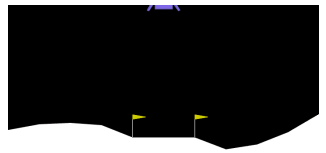
✓ Supported ✗ Unsupported

Write once, run anywhere

FFT: Power of 2 vs Non-power of 2



Reinforcement Learning – LunarLander

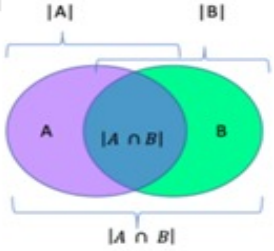


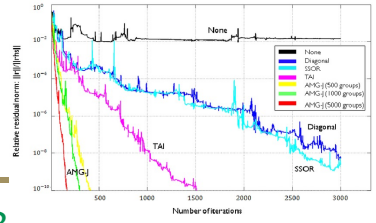
Task 2: GPU Productivity and Performance (Speed/Power)

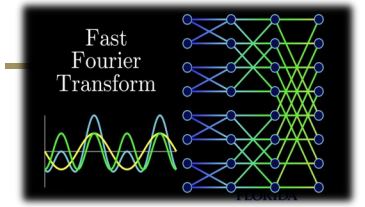
- Motivation
  - High Performance ***but*** Low Productivity: Low-level C/C++-based languages → CUDA, OpenCL, SYCL
  - High Productivity ***but*** Less Performant: (1) Vendor libraries & tools. (2) High-Productivity Lang.
    - Issues: (1) Non-portable libraries, e.g., CUDA libraries *not* portable to AMD hardware. (2) Lack of hardware-software interface control to achieve high performance. (3) Rigidity of pipelines and data abstractions
- Approach

  Think AMD MI300 CPU+GPU and NVIDIA GH200 CPU+GPU
  - Evaluate 1st & 3rd-party solutions to reach more devices with less code (*without sacrificing performance*)
  - First-party solutions? Translators, e.g., Intel's *dpct* (CUDA→oneAPI/SYCL) or AMD's *hipify* (CUDA→HIP)
  - Third-party? Languages, e.g., **Chapel, OpenCL, SYCL.** Libraries, e.g., PyTorch, Apache Arrow, C++ std::par
- Tasks 2a, 2b, 2c: Productivity & performance of OpenCL SYCL CHAPEL oneAPI MPI
  a) Jaccard similarity & other graph apps (e.g., Gunrock) → parallel & distributed CHAPEL MPI
  b) Biconjugate gradient stabilized method (BiCGSTAB) OpenCL SYCL
  c) Community detection → graph clustering (see V3-24) oneAPI SYCL
  d) FFT with custom-length (non-power-of-two) input SYCL

$$js(A,B) = \frac{|A \cap B|}{|A \cup B|}$$







V1

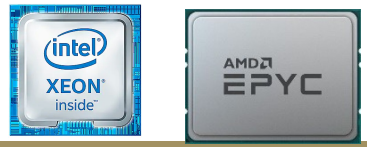
Task 3: Study, Analysis, and Outreach for Chapel GPU

- Motivation
  - Chapel 1.0 (2009) to Chapel 1.26 (2022)
    - High-Productivity Computing for Parallel & Distributed ***CPU*** Computing → Alternative to MPI or MPI+OpenMP
  - Rise of GPU Computing (since 2007)
    - Low-Level Languages (CUDA, OpenCL, …) + High-Level but Rigid/Non-Portable (PyTorch, cuBLAS, …) Libraries
- Approach
  - Study, analyze, & educate community on Chapel GPU for parallel & distributed ***CPU+GPU*** computing
- Tasks 3a, 3b, and 3c

  a) Stress-test and evaluate updated and new features of the Chapel 1.33+ releases

  b) Profile & analyze performance of Chapel 1.33+ on NVIDIA GPU with Jaccard similarity & OpenDwarfs
    - Chapel runtime overheads, Intra-node parallelism and **inter-node parallelism** (*a la* OpenMP/OpenACC/CUDA and MPI, respectively)
    - Conduct on AMD/Intel CPU and AMD/Intel GPU, as time & access permits

  c) Testing, deployment, and education with Chapel 1.33+
  … for parallel computation class(es) at VT

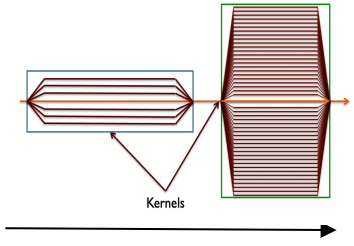


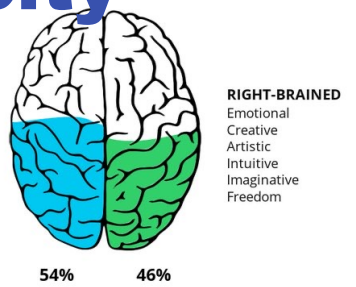
V1

Task 4: Simultaneous Co-scheduling of Heterogeneity
(CPU+GPU+FPGA) – Details in Appendix

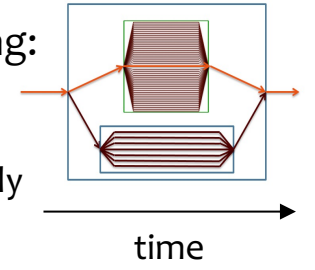
- Motivation: Why Traditional *Processor* Scheduling?
  - Physiologically, left & right brain used simultaneously
  - Silicon-wise, use CPU & GPU brain simultaneously … and even FPGA. (CPU+GPU integration, too!)
  - CoreTSAR: Past R&D on CPU+GPU co-scheduling of regular apps via OpenMP

Traditional Scheduling: CPU kernel, *then* GPU kernel





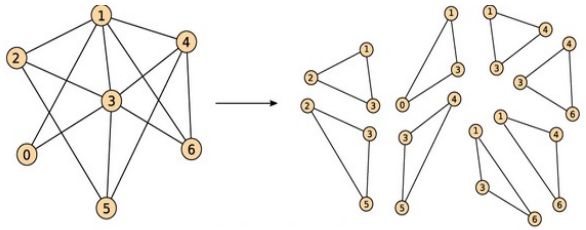
Co-Scheduling: CPU & GPU kernels simultaneously



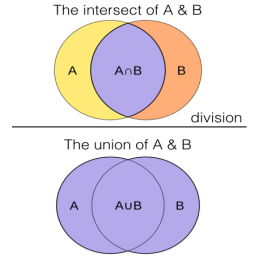
OpenMP OpenACC Directives for Accelerators

- Approach — Think AMD MI300 and NVIDIA GH200 CPU+GPU
  - CoreTSAR++: Generalize co-scheduling for multi-heterogeneity (CPU+GPU+FPGA) and to support regular & irregular workloads with data-dependent memory access patterns

Triangle counting

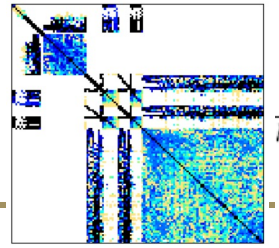


Jaccard similarity



- Task 5a, 5b, 5c: CoreTSAR++ Exploration — SYCL
  - a) Identify & implement appropriate irregular apps to co-schedule
  - b) Manually implement & evaluate co-scheduled irregular apps
  - c) Automate co-scheduling on heterogeneous system (CPU+GPU+ …)
  - d) Investigate simultaneous co-scheduling using Chapel — CHAPEL

Sparse linear solver → Biconjugate gradient stabilized



$$\frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 1 & -2 & 1 \\ 0 & \dots & \dots & \dots & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-3} \\ u_{N-2} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-3} \\ f_{N-2} \end{pmatrix}$$

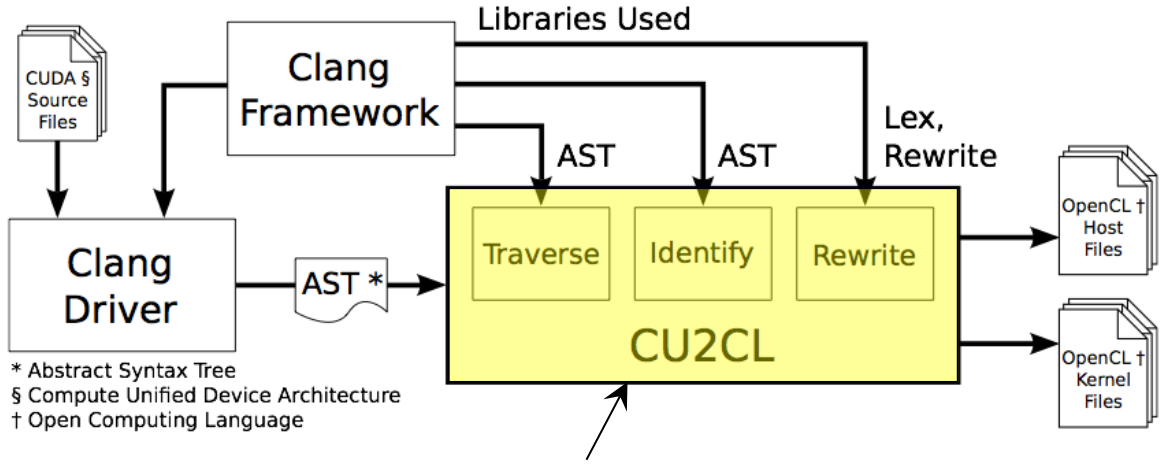
V1

Task 5: Auto-Generation of Source-to-Source Translators

- Motivation
  - Too many heterogeneous programming languages & devices in legacy scientific code. Non-portable.
  - Alternative to "write once, run anywhere" languages? *Source-to-source translation*
    - Examples: CUDA to OpenCL; OpenMP to CUDA; etc., but such translators are *MANUALLY* coded.
  - Goal: Automatically generate such automated source-to-source translators

Write Once, Run Anywhere via Source-to-Source Xlation

- Approach
  - **Leverage abstract syntax trees (ASTs) to "structure" CUDA code,** ensuring that translations respect domain knowledge and best practices.
  - **Combine neural machine translation (NMT) with neuro-symbolic AI** to decipher underlying logic of domain-specific CUDA code, enabling more accurate translation.



Instead of manually creating CU2CL (from V1-12) and manually updating it when the clang ecosystem changes, *auto-generate the CU2CL source-to-source translator.*

- Task 5a, 5b, 5c

a) Update CU2CL source-to-source translator from V1-12

b) Train NLP to auto-generate CUDA-to-OpenCL translator

c) Evaluate accuracy of original CU2CL to auto-generated CU2CL on pre-existing suite of application codes

d) Assess efficacy of auto-generator for other languages

V1

Milestones, Deliverables, and Budget

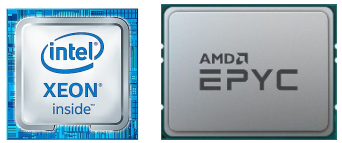
- **Major Milestones (Tasks: T1-T6)**
  - T1: FPGA Productivity: ***FFT via VITIS HLS+PYNQ and RL via VITIS AI***
  - T2: GPU Productivity: ***Jaccard Similarity & Other Graphs via OpenCL/SYCL***
  - T3: Chapel 1.32: ***Synthesis and analysis of Jaccard similarity – parallel and distributed***
  - T4: Simultaneous Co-scheduling of Heterogeneity: ***CPU+GPU co-scheduling of irregular app***
  - T5: Auto-Generation of Source-to-Source Translation: ***AI-generated CUDA-to-OpenCL translator***

- **Deliverables**
  - Monthly progress reports, along with mid-year and end-of-year full reports
  - 2-3 publications at top-tier conference venues or journals

- **Recommended Budget**
  - Minimum: 6 memberships (300 votes)
  - Maximum: 13 memberships (650 votes)

Conclusion

- Enable ***high-productivity computing*** in ***heterogeneous*** computing systems: CPU + {CPU, GPU, FPGA, TPU, …} via open standards: OpenCL, SYCL, Chapel, and to a lesser degree, oneAPI
- Evaluate ***performance & productivity*** of ***representative apps*** (OpenDwarfs, FFT, Jaccard similarity, biconjugate gradient stabilized method – BiCGSTAB, and graph algorithms) on ***different devices*** (CPUs, GPUs, and FPGAs)



Member Benefits

- Direct influence over processors & frameworks studied and apps & datasets used
- Direct benefit from new methods, tools, datasets, codes, models, and insights created as well as new metrics of evaluation
- Direct insights from R&D and analysis

APPENDIX

1. FPGA: Proprietary (RTL & HLS) vs. C/C++ (OpenCL/SYCL/oneAPI)
2. GPU: Low-Level SIMD (CUDA/OpenCL/SYCL) vs. High-Level SIMD (Chapel)
3. Open-Source Approach Leveraging Open Source: Evaluation Metrics
4. Open-Source Approach Leveraging Open Source
   - Enabling Further High-Productivity Research
5. Details for Task 2c: Graph Clustering via oneAPI/SYCL (Related to V2-24)
6. Details for Task 3: Study, Analysis, and Outreach for Chapel GPU
7. Details for Task 4: Simultaneous Co-scheduling of Heterogeneity (CPU+GPU+FPGA)
8. Details for Task 4: Simultaneous Co-scheduling of Heterogeneity (CPU+GPU+FPGA): Irregular Apps

FPGA: Proprietary (RTL & HLS) vs. C/C++ (OpenCL …)

V1

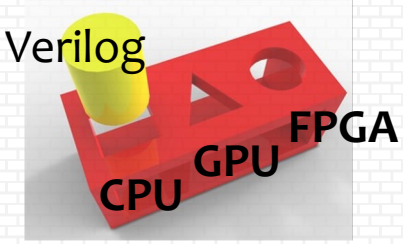
Low-Level RTL (Verilog/VHDL)

Pros

- Direct control over timing requirements and design
- Maximum performance via via HW-specific optimization

Cons

- SLOW development time
- Tedious to read & understand
- More error-prone
- ***NOT portable***

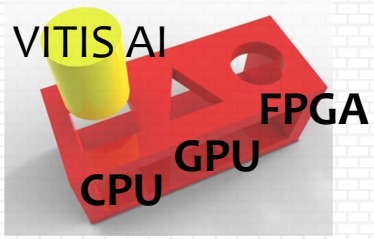


"machine language"

Mid-Level HLS (VITIS HLS & AI)

Pros and **Cons**

- Fairly slow development time
- Less tedious to read & understand
- Fairly error-prone
- ***NOT portable***
- Still requires low-level knowledge to prepare and invoke the kernel

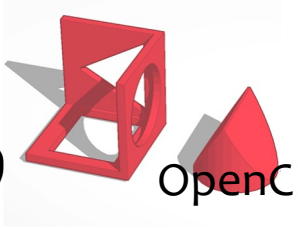


"assembly language"

High-Level C/C++ (OpenCL/SYCL)

Pros

- Faster development time
- Easier to read and understand
- Less error-prone
- ***Portable (CPU, GPU, FPGA)***



Cons

- Limited control over timing requirements
- Performance loss due to overhead of high-level abstractions
- Mixed vendor support for open standards

"high-level language"

GPU: CUDA/OpenCL/SYCL vs. Chapel

(Parallel) (Parallel+Distributed)

Low-Level SIMD (CUDA/OpenCL/SYCL)

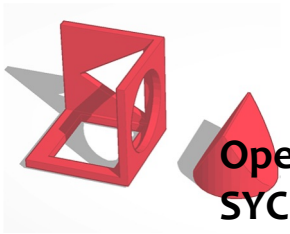
Pros

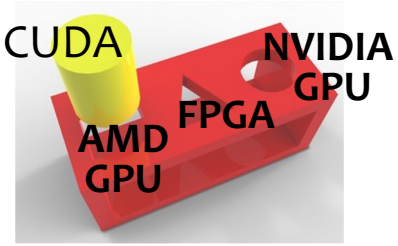
- Direct control over mapping. registers, and memory

- Maximum performance via hardware-specific optimization

Cons

- Slower development time
- Tedious to read & understand
- Manual device/thread decomposition
- Manual distribution via MPI





High-Level SIMD+MIMD (Chapel)

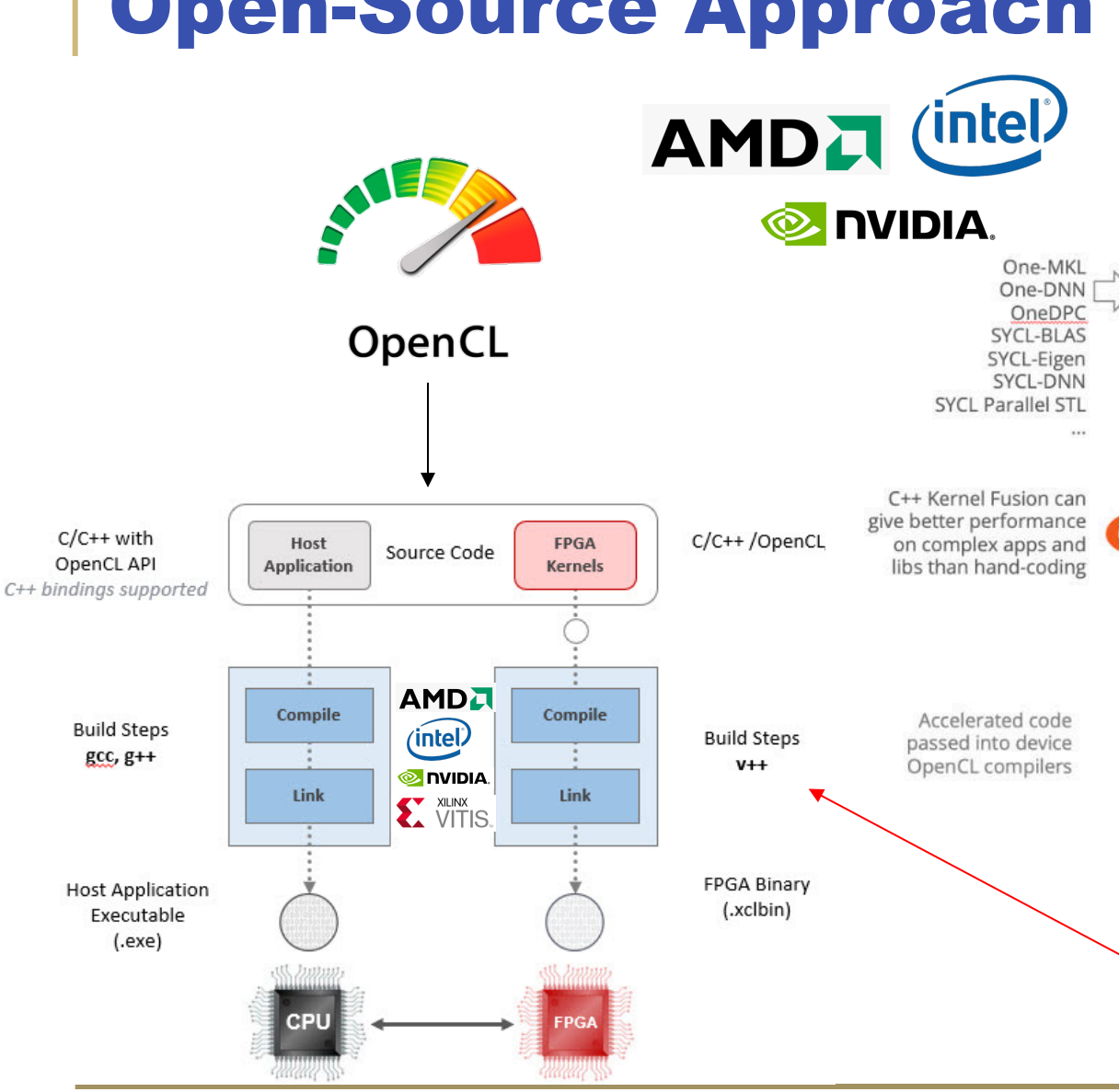
Pros

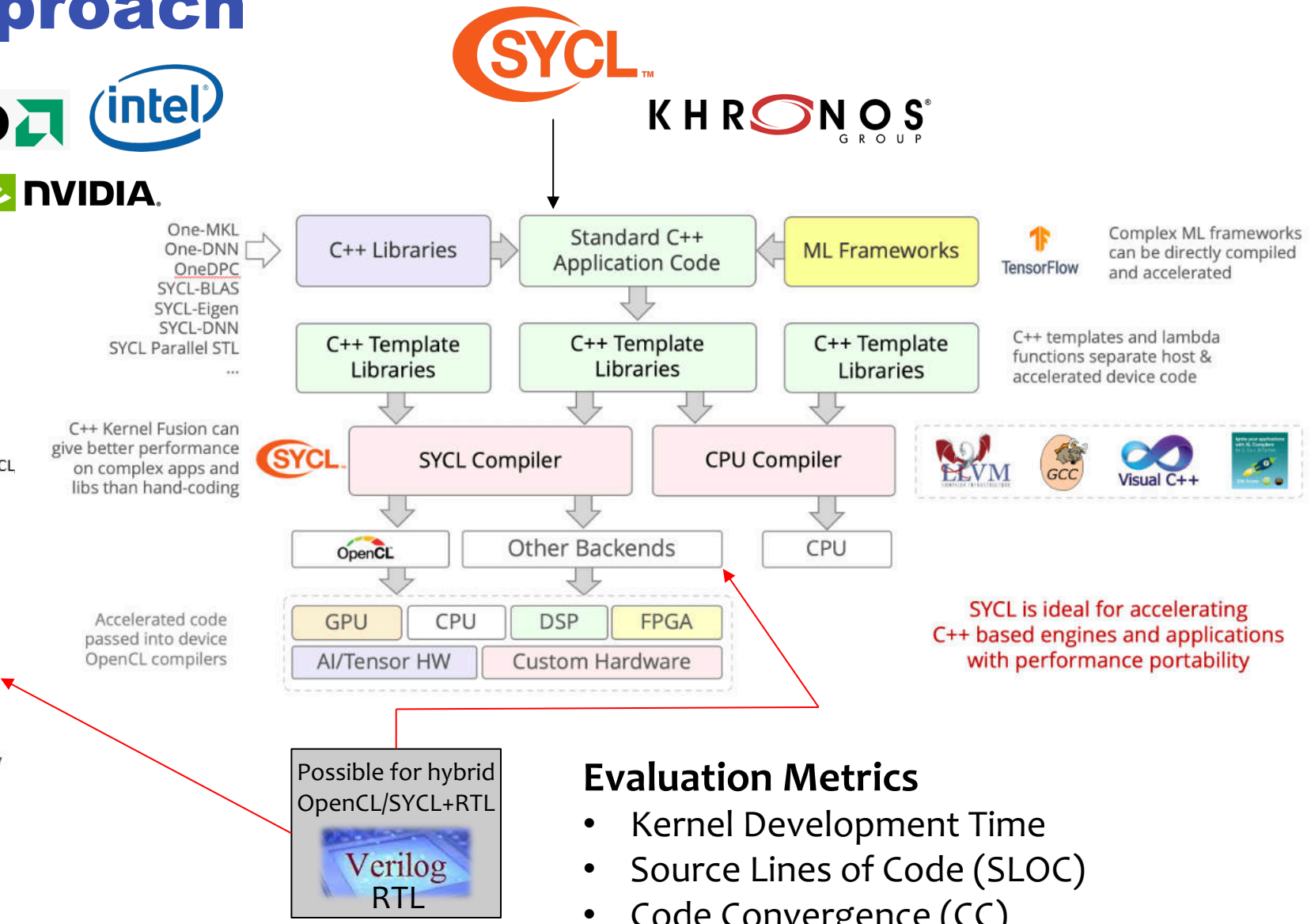
- Faster development time
- Easier to read and understand
- Automatic array/domain decomposition
- Natively extensible from *CPU-parallel* to *heterogeneous-parallel* and *distributed*

Cons

- Limited control over mapping, registers, and memory
- Performance loss due to overhead of high-level abstractions
- Limited developer resources and community

Open-Source Approach

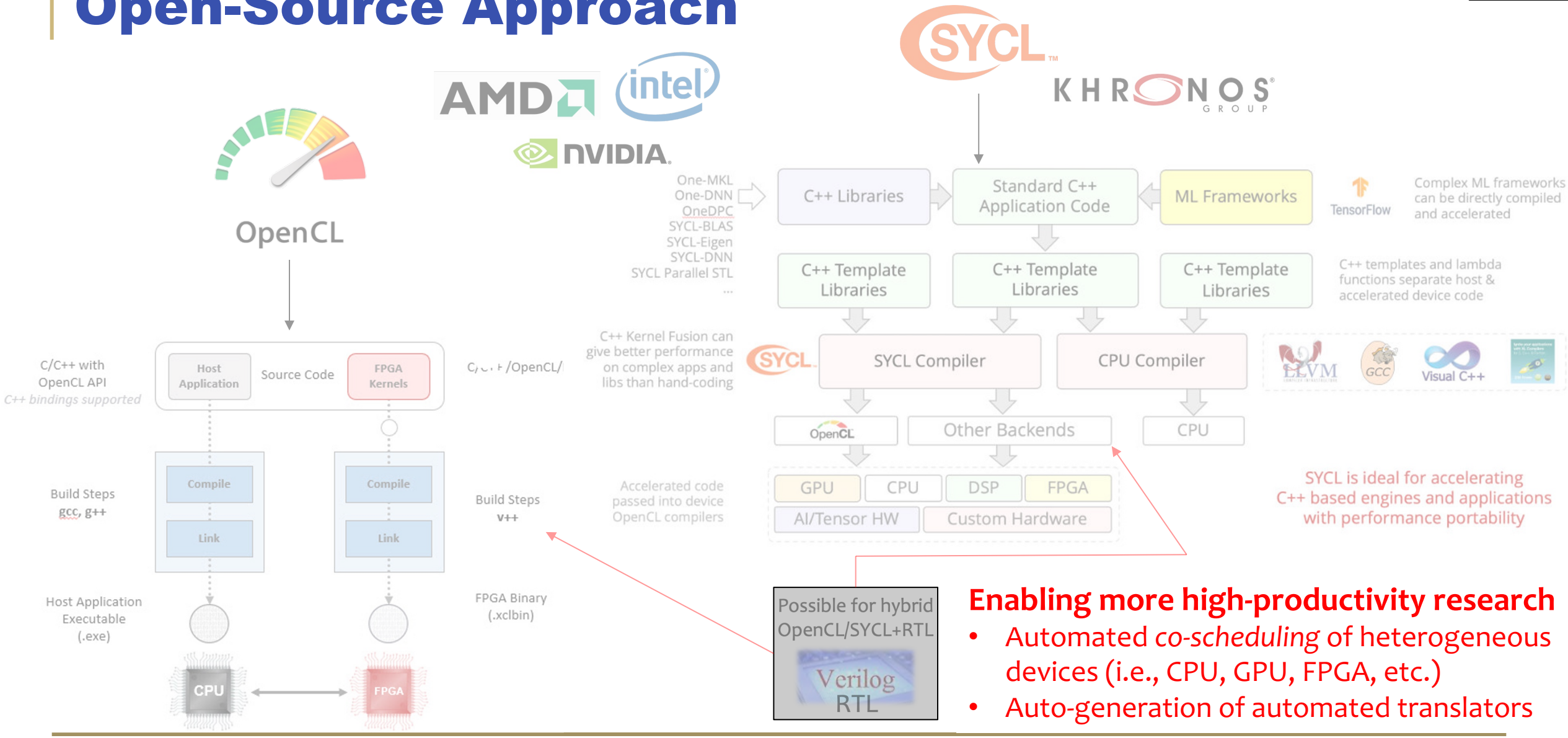




Evaluation Metrics

- Kernel Development Time
- Source Lines of Code (SLOC)
- Code Convergence (CC)
- Performance-Productivity Product (PPP)
- Performance Portability ($\mathcal{P}$)

Open-Source Approach

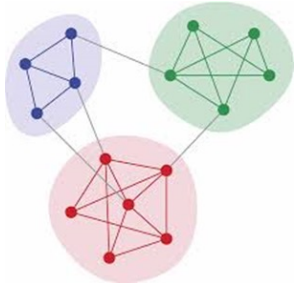


Enabling more high-productivity research

- Automated *co-scheduling* of heterogeneous devices (i.e., CPU, GPU, FPGA, etc.)
- Auto-generation of automated translators

Task 2c: Graph Clustering via oneAPI/SYCL

V1



- Motivation
  - Accurate graph clustering → **computationally expensive ✗**
  - Available implementations of statistically robust graph clustering algorithms? **CPU-oriented ✗**
    - Frontier supercomputer: 99% of FLOPS from GPUs
- Observation
  - CPU-only implementations sacrifice runtime gains of massively parallel GPUs (and low-power gains of FPGAs)

Graph clustering → use cases across many domains ✓

Networking/Finance

Intrusion/Fraud Detection

Bioinformatics

Epidemiology & Drug Discovery

Social media

Recommender Systems

- Approach
  - Translate C++ graph clustering code (V2-24) to oneAPI/SYCL to run heterogeneously on CPU+GPU
  - Evaluate and compare runtime performance of C++ optimized CPU code to oneAPI/SYCL translation on different architectures: CPU-only, GPU-only, CPU+GPU, CPU+GPU with co-scheduling)

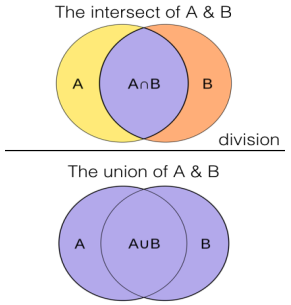
Task 3: Study, Analysis, and Outreach for Chapel GPU

V1

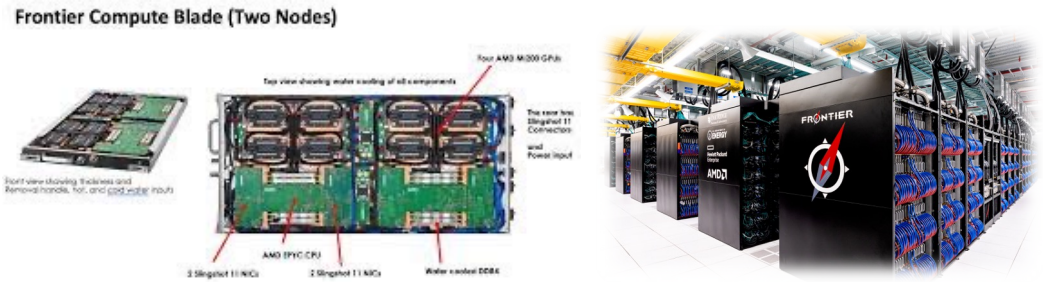
- Motivation
  - Chapel 1.30 (2023): Intro of GPU Support
  - Need for a *portable high-level* approach that provides flexibility, interoperability, and performance of C-based solutions with the more productive and intuitive syntax of library- and Python-based solutions

- Approach
  - Study, analyze, & educate on Chapel GPU for parallel & distributed ***CPU+GPU*** computing
    - Intra-node parallelism across CPUs ***and*** GPUs from ***multiple*** vendors (unlike CUDA/HIP)
    - Multi-node distribution of tasks ***without*** an add-on library like MPI (unlike CUDA/HIP/SYCL/OpenCL)
    - Intuitive parallel abstractions like forall and *automatically promoted and distributed array* ops

- Tasks 3a, 3b, and 3c
  - a) Stress-test and evaluate updated and new features of the Chapel 1.32 release
  - b) Profile & analyze performance of Chapel 1.32
    - Device: NVIDIA GPU
      (No support for other GPUs yet.)
    - Apps: Jaccard similarity →
      OpenDwarfs for Chapel
      (i.e., ChapelDwarfs)



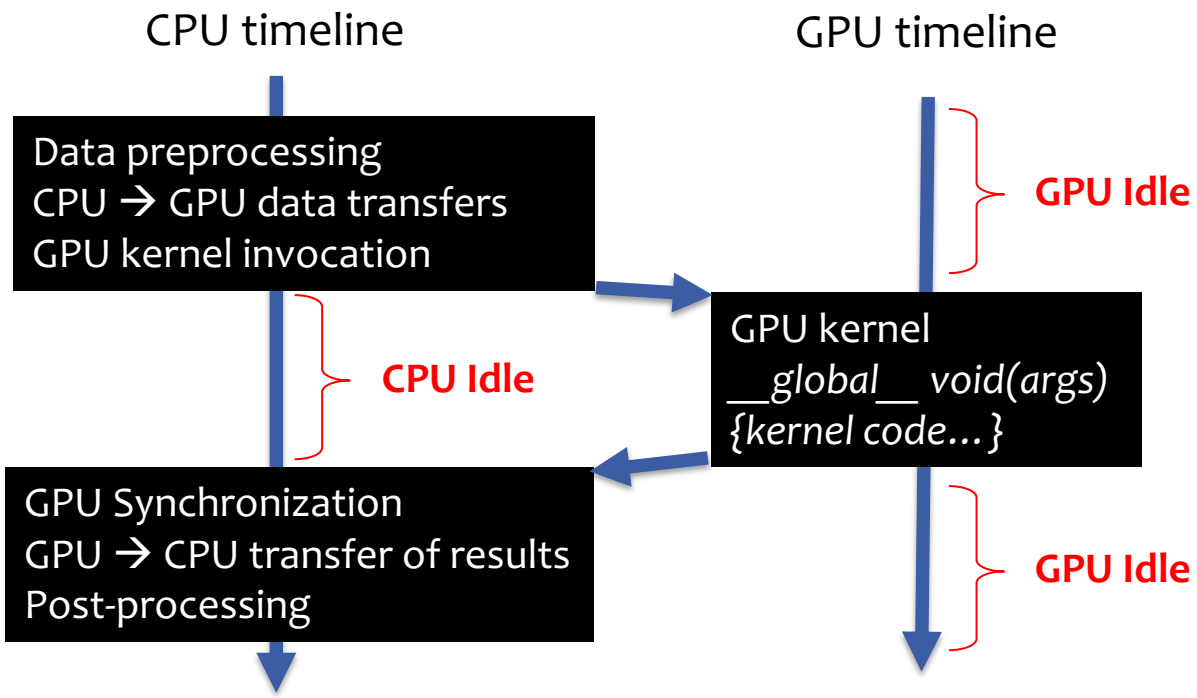
    - Environments: Intra-node and inter-node



  - c) Testing, deployment, and education with Chapel 1.32 for parallel computation class(es) at VT

Task 4: Simultaneous Co-scheduling of Heterogeneity (CPU+GPU+FPGA)

- Motivation (in Detail)
  - CPU typically idle when GPU executes kernel
    - Why not distribute a portion of workload to CPU?!
- Our Prior Work
  - Automated distribution of workload of ***regular*** apps for better performance
  - Context: OpenMP OpenACC
- Proposed Research
  - Automated distribution of workload for ***irregular*** apps for better performance
  - Context: SYCL
  - Heterogeneity: CPU+GPU, CPU+FPGA, CPU+GPU+FPGA

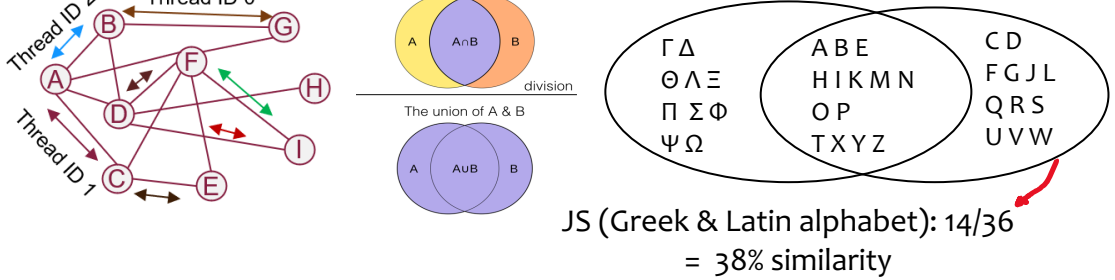


V1

Task 4: Simultaneous Co-scheduling of Heterogeneity (CPU+GPU+FPGA): Irregular Apps

Graph Apps from IARPA AGILE Program [1]

- **Jaccard similarity (JS) in graph datasets**
  - Parallel JS computation across all vertex pairs that form edges of the graph



- **Triangle counting (TC) in graph datasets**
  - Evaluate the total number of unique triangles formed by the edges in the graph

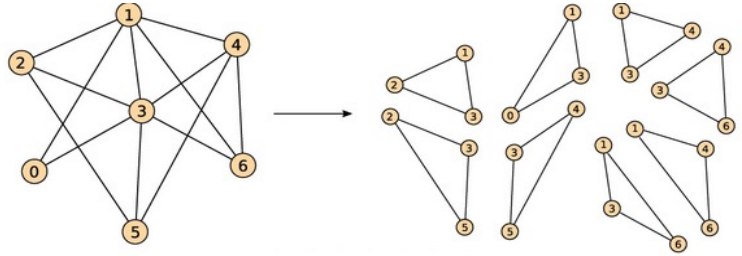
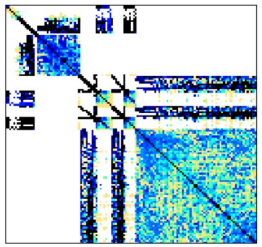


Image source: https://www.computer.org/csdl/journal/td/2017/12/08000612/13rRUxYINeZ

[1]: https://www.iarpa.gov/images/PropsersDayPDFs/AGILE/AGILE_Program_Workflows_FINAL.pdf

Scientific Computing App

- **Biconjugate gradient stabilized (BiCGSTAB)**
  - Sparse linear system solver with irregular memory access pattern
  - Used in computational fluid dynamics



$$\frac{1}{h^2}\begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 1 & -2 & 1 \\ 0 & \dots & \dots & \dots & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-3} \\ u_{N-2} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-3} \\ f_{N-2} \end{pmatrix}$$

Task 5: Auto-Generation of Source-to-Source Translators

V1

- Motivation
  - Too many heterogeneous programming languages & devices in legacy scientific code. Non-portable.
  - Alternative to "write once, run anywhere" languages? *Source-to-source translation*
    - Examples: CUDA to OpenCL; OpenMP to CUDA; etc., but such translators are *MANUALLY* coded.
  - Goal: Automatically generate such automated source-to-source translators
- Approach
  - **Leverage abstract syntax trees (ASTs) to "structure" CUDA code,** ensuring that translations respect domain knowledge and best practices.
  - **Combine neural machine translation (NMT) with neuro-symbolic AI** to decipher underlying logic of domain-specific CUDA code, enabling more accurate translation.

- Existing Source-to-Source Translators
  - OpenMP → CUDA, OpenCL, OpenACC, ISPC, MPI
  - OpenACC → OpenMP
  - CUDA → OpenCL, OpenMP, OpenACC, HIP
  - OpenCL → CUDA
- Task 6a, 6b, 6c
  a) Update CU2CL source-to-source translator from V1-12
  b) Train NLP to auto-generate CUDA-to-OpenCL translator
  c) Evaluate accuracy of original CU2CL to auto-generated CU2CL on pre-existing suite of application codes
  d) Assess efficacy of auto-generator for other languages