codeplay®

Enabling AI to be Open, Safe & Accessible to All

Improving performance of SYCL applications on CPU architectures using LLVM-directed compilation flow

Pietro Ghiglio, Uwe Dolinsky, Mehdi Goli, **Kumudha Narasimhan**

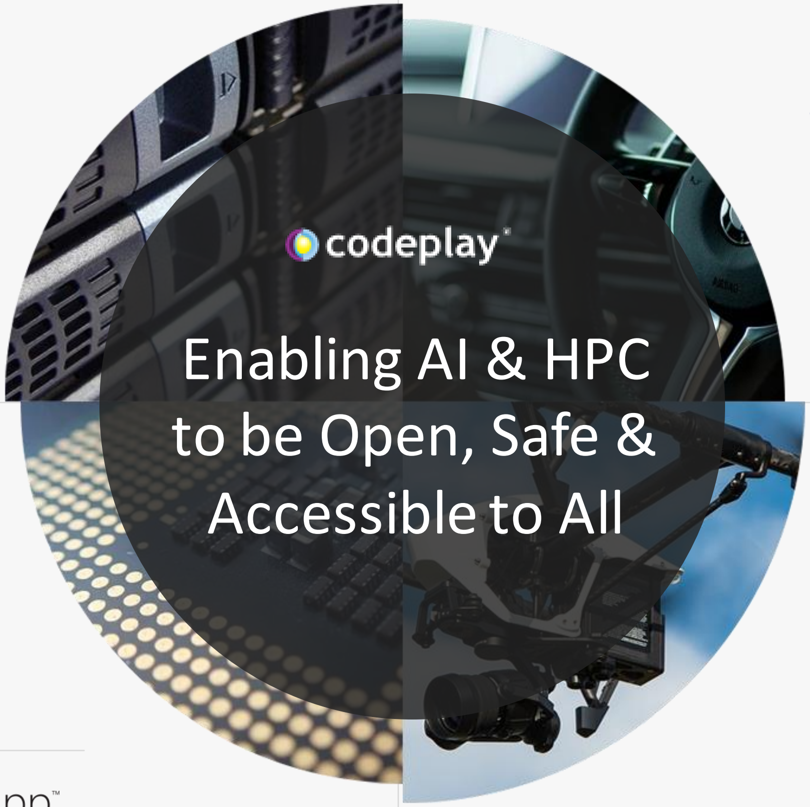
2nd April 2022

Company

Leaders in enabling high-performance software solutions for new AI processing systems

Enabling the toughest processors with tools and middleware based on open standards

Established 2002 in Scotland with ~80 employees



codeplay

Enabling AI & HPC to be Open, Safe & Accessible to All

Partners

intel
BROADCOM
SYNOPSYS
CEVA
Imagination
RENESAS
KMC Kyoto Microcomputer Co., Ltd.
NSI-TEXE
BERKELEY LAB
OAK RIDGE National Laboratory
Argonne NATIONAL LABORATORY

And many more!

Products

Acoran

Integrates all the industry standard technologies needed to support a very wide range of AI and HPC

ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR-V™, HSA™ and Vulkan™

ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

Markets

High Performance Compute (HPC)
Automotive ADAS, IoT, Cloud Compute
Smartphones & Tablets
Medical & Industrial

Technologies: Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

Agenda

1. Motivation: Why SYCL on CPU?
2. Introduction to SYCL and its compilation flow.
3. SYCL host compilation.
4. Performance Results.
5. Conclusions and future work.

Motivation: Why SYCL on CPU?

- **SYCL** mostly targets **heterogeneous systems** with accelerators, but many systems have CPU [only].
- **CPUs** used **alongside** the main accelerator.
- Achieve **performance portability** of SYCL application on CPUs.
- Allow to support platforms for which an **OpenCL implementation is not available**.
- **Remove overheads** introduced by OpenCL/other backends.

SYCL: An open standard for portable software acceleration.

- C++ based open standard API introduced by Khronos.
- Provides single source programming model for heterogenous systems.
- Abstraction layer initially designed on top of OpenCL, now supports several different backends.
- Multiple implementations:
  - ComputeCpp (Codeplay)
  - DPC++ (Intel)
  - triSYCL (AMD)
  - hipSYCL (Heidelberg University)
  - neoSYCL (Tohoku University)

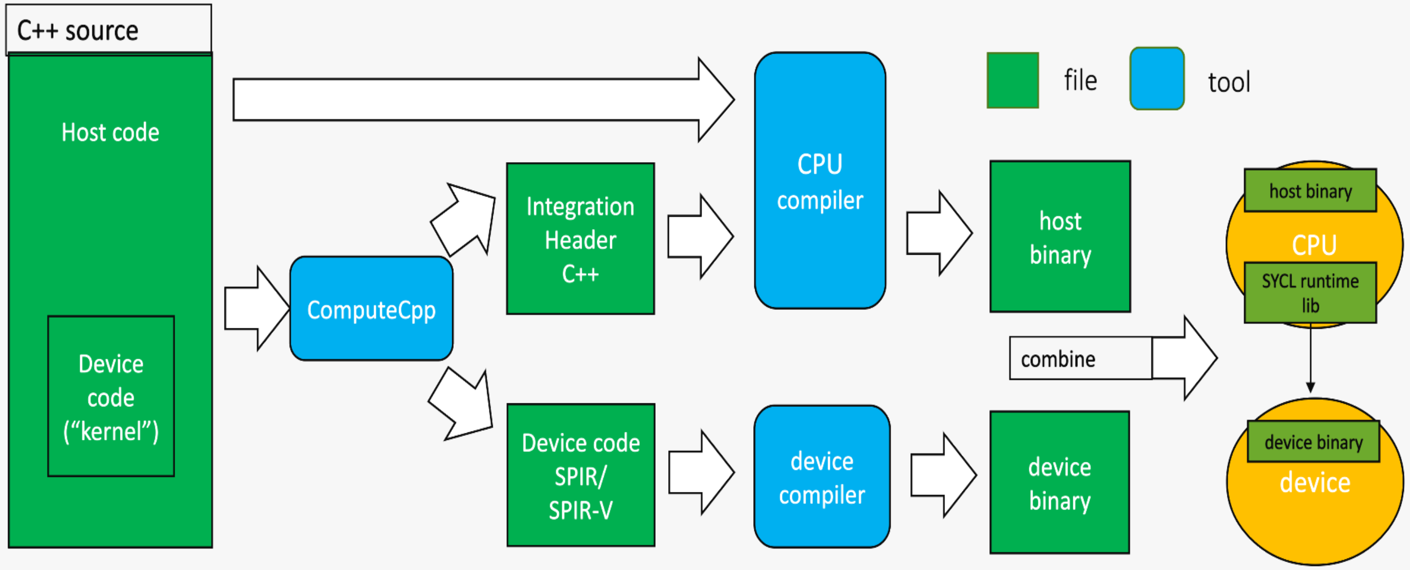
https://sycl.tech/ https://www.khronos.org/sycl/

Targeting SYCL to CPUs

- SYCL allows us to **target CPUs as accelerators**, depending on the backend, e.g. OpenCL can **JIT compile** SPIR/SPIR-V into x86 code.
- Benefits from **offline** compilation.
- SYCL **host device** allows us to compile any SYCL program with any C++-compliant compiler (no need for device compiler).
  - Usually used as a fallback mechanism when other backends are not available – performances not guaranteed.

General SYCL compilation flow

- Same C++ code is **compiled twice**.
- Device compiler extracts kernel code, lowers it to an **intermediate representation** (e.g. SPIR-V), bundles it into the integration header.
- **Integration headers** are included when host compiler re-compiles the source code.
- Final executable contains IR bundle, runtime backend **JIT compiles** it.
- Support for **offline compilation** (pre-compile the IR bundle, no JIT compilation required).
- One option to ComputeCPP does all these steps internally.



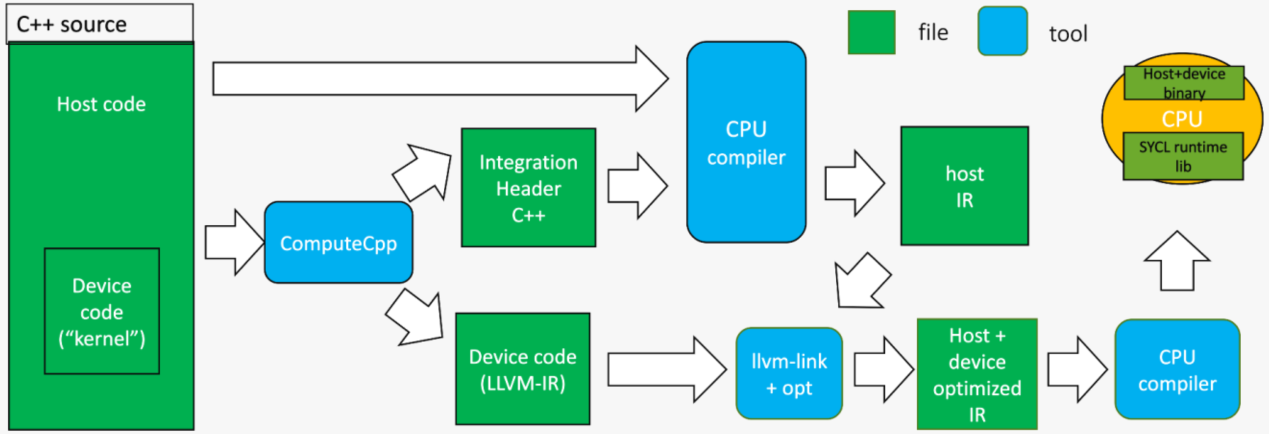
ComputeCPP: compute++ helloworld.cpp -fsycl

New SYCL host compilation

- CPU-specific SYCL backend.
- Offline target, **direct substitute** of OpenCL/other backends.
- Allows to **efficiently execute** SYCL applications on CPUs, without any other dependency than ComputeCpp.
- Performs **same set of program transformations and optimizations** as an OpenCL implementation, but inside the SYCL device compiler.

New SYCL host compilation flow

- Same **two stage compilation** as the usual SYCL flow.
- Device compiler doesn't emit SPIR-V, but **emits LLVM-IR** with the same target triple as the host code.
- Using a clang-based host compiler, we can **llvm-link together the host and the device code**, and then optimize the whole translation unit.
- Acts as an **offline target for SYCL**, allows us to handle the whole SYCL application like a standard CPU application.
- Implemented in ComputeCpp's driver so that it's transparent to the user.



ComputeCPP: compute++ helloworld.cpp -sycl-driver –sycl-target=host

Whole Function Vectorization

- Integrated a Whole Function Vectorizer in **ComputeCpp**, in order to **bypass OpenCL** and perform it **offline**.
- Reduces the number of threads required to execute the workload – **packs more work** into one thread.
- Deal with complex kernel code including **barriers**.

Whole Function Vectorization - Example

Original kernel

```
define void @SimpleVadd(i32*, i32*, i32*) {
  %5 = call i64 @_Z13get_global_idj(i32 0)
  %6 = getelementptr inbounds i32, i32* %1, i64 %5
  %7 = load i32, i32* %6, align 4
  %8 = getelementptr inbounds i32, i32* %2, i64 %5
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %9, %7
  %11 = getelementptr inbounds i32, i32* %0, i64 %5
  store i32 %10, i32* %11, align 4
  ret void
}
```

Vectorized kernel

```
define void @SimpleVadd_v16(i32*, i32*, i32*) {
  %5 = call i64 @_Z13get_global_idj(i32 0)
  %6 = getelementptr inbounds i32, i32* %1, i64 %5
  %7 = bitcast i32* %6 to <16 x i32>*
  %8 = load <16 x i32>, <16 x i32>* %7, align 4
  %9 = getelementptr inbounds i32, i32* %2, i64 %5
  %10 = bitcast i32* %9 to <16 x i32>*
  %11 = load <16 x i32>, <16 x i32>* %10, align 4
  %12 = add nsw <16 x i32> %11, %8
  %13 = getelementptr inbounds i32, i32* %0, i64 %5
  %14 = bitcast i32* %13 to <16 x i32>*
  store <16 x i32> %12, <16 x i32>* %14, align 4
  ret void
}
```

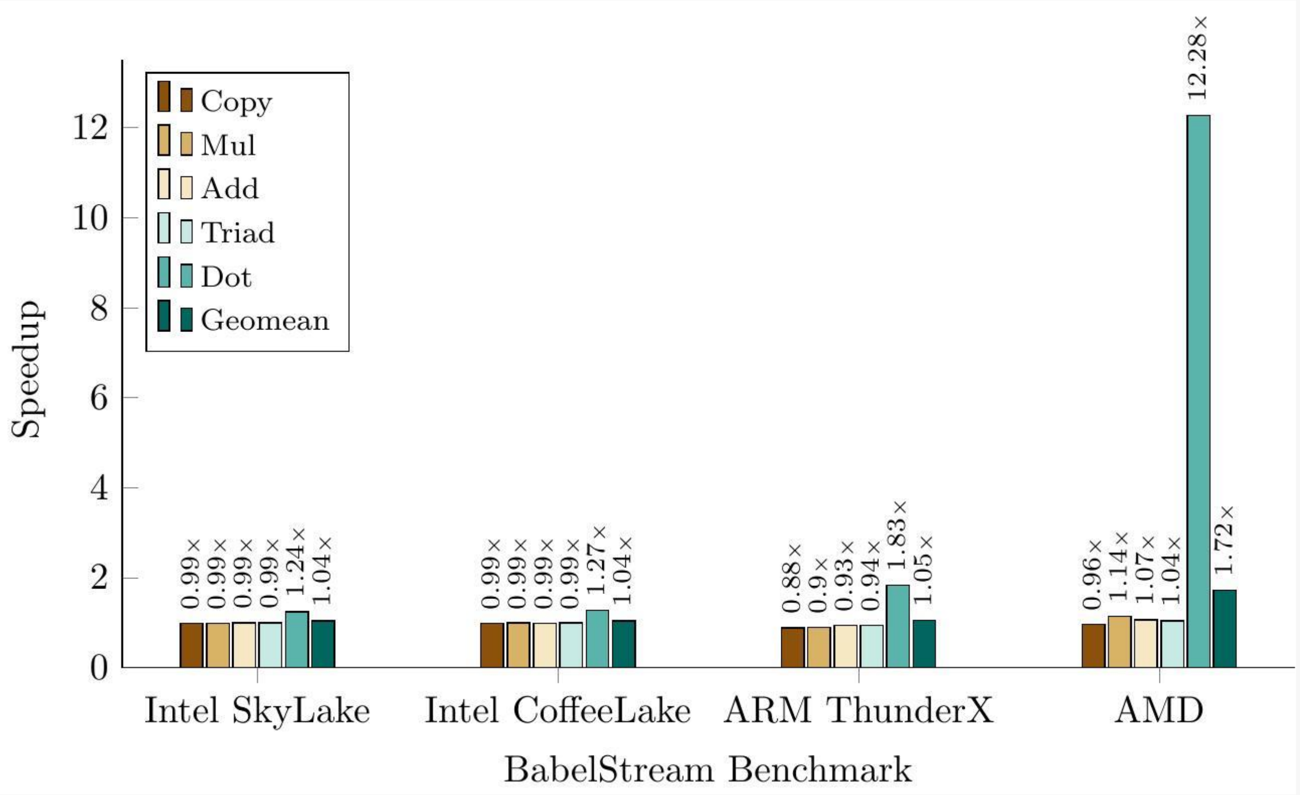
Experiment setup

- Benchmarks: BabelStream + Matrix Multiply from ComputeCpp-SDK.
- Target hardware:

Vendor	Processor	Frequency	No. Cores	Memory	OpenCL driver version
Intel	SkyLake (i7-6700)	4.20 GHz	4	32 GB DDR4	18.1.0.0920
Intel	CoffeeLake (i7-8700)	4.00 GHz	8	32 GB DDR4	2021.12.9.0.24
Cavium	ThunderX 88XX	2.00 GHz	48	32 GB DDR4	ComputeAorta 1.65
AMD	EPYC 7402	2.80 GHz	48	256 GB DDR4	PoCL 1.8

Results - BabelStream
Speedup of Host compilation vs OpenCL

- **ComputeCpp as SYCL implementation.**
- Intel SkyLake – Intel OpenCL.
- Intel CoffeLake – Intel OpenCL.
- ARM ThunderX – ComputeAorta.
- AMD Epyc – PoCL.
- **Performances on par** with OpenCL for memory-bound kernels.
- Speed-up on compute-bound dot kernel.
- Outlier: 12x speed-up on AMD dot: tuning vector width reported that the unvectorized kernel performs better on AMD. Due to different vector instructions emitted by the backend.

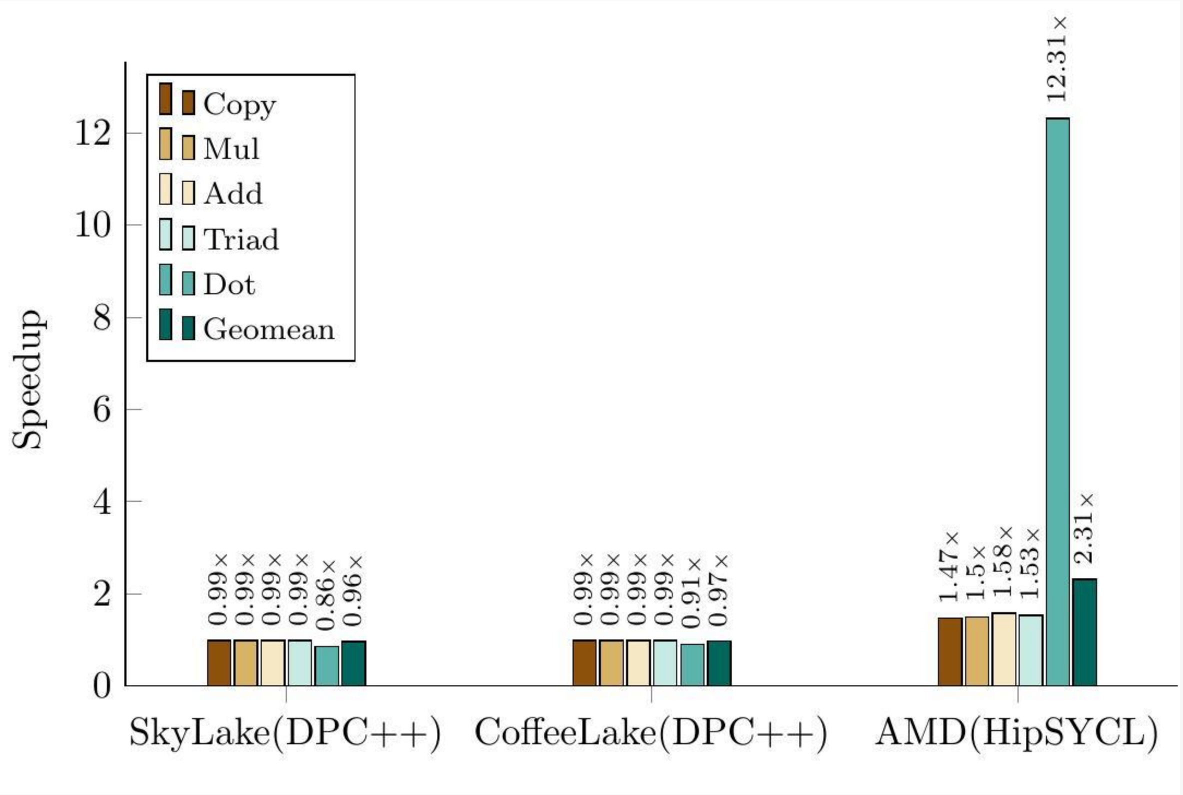


Results – Other SYCL implementations
Speedup of Host compilation vs OpenCL and OpenMP

- **DPC++** using Intel OpenCL.
- **hipSYCL** using OpenMP backend.
  - Same outlier on AMD.

Performance Comparisons

- Performance for host compilation is **comparable** to **DPC++** + Intel OpenCL.
- Performance for host compilation **faster** than **hipSYCL**'s OpenMP backend.



Results – Matrix Multiply

Architecture	Default OpenCL [s]	SYCL Host Compilation [s]	Speedup
Intel SkyLake	0.834	0.742	1.12x
Intel CoffeeLake	0.512	0.434	1.18x
Cavium ThunderX	1.519	1.089	1.39x
AMD Epyc	0.108	0.66	1.63x

Architecture	Compiler	Baseline	SYCL Host Compilation	Speedup
Intel SkyLake	DPC++	0.581	0.599	0.97x
Intel CoffeeLake	DPC++	0.415	0.416	1x
AMD	HipSYCL	0.080	0.066	1.21x

Future Work

- **Auto-tuning** for compile time and runtime parameters (number of threads, work-group size …).
- Allow bundling intermediate representation in integration header, together with CPU-specific offline compiled kernels.
- Improve SYCL runtime, exposing more API implementations to the compiler to improve performances.

Conclusion

- Demonstrates **acceleration of SYCL code on CPUs** without requiring OpenCL backend.
- Added a **configurable vectorization** pass to support different types of CPUs.
- **Comparable performance** to state-of-the-art OpenCL implementation.

We're Hiring!
codeplay.com/careers/

codeplay®

Enabling AI to be Open, Safe & Accessible to All

Thank You

@codeplaysoft

/codeplaysoft

www

codeplay.com