NVIDIA GPU ADAPTATION FOR SYCL-MLIR

(EXTEND AI COMPILER ECOSYSTEM OF SYCL)

Beijing Institute of Open Source Chip (BOSC)
University of Chinese Academy of Sciences (UCAS)
Boyuan Bao

BackGround

SYCL-MLIR:

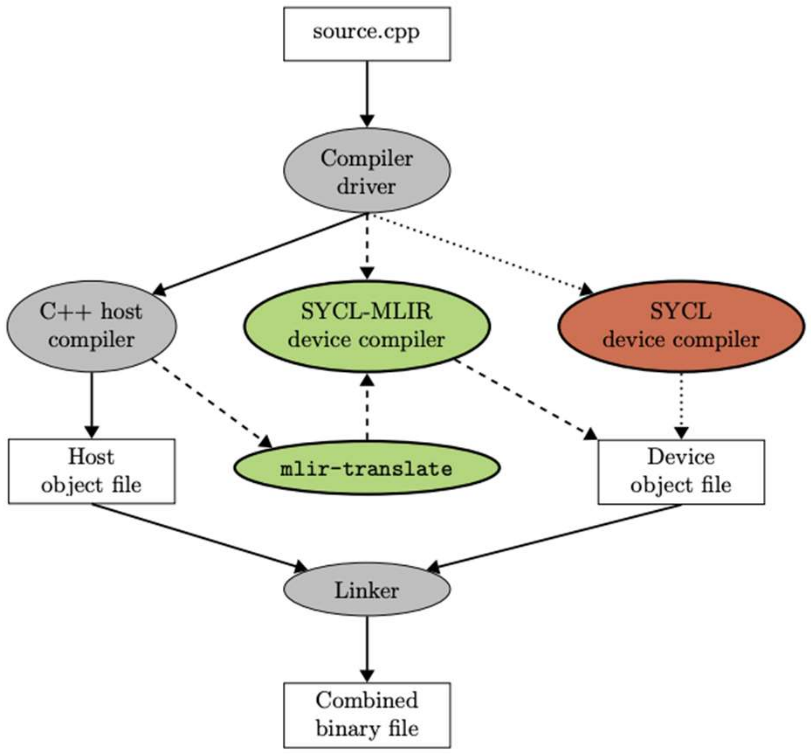
SYCL-MLIR is an MLIR-based compiler infrastructure for the SYCL heterogeneous programming model. Its goal is to provide a **single-source, single-compiler-pass (SSCP)** compilation flow that can analyze and optimize **host and device code together**, rather than separately.

Key feature：

- Host code and device kernels coexist in the **same MLIR module**, which enables cross-boundary analyses and optimizations
- Models core SYCL concepts as **MLIR types / operations / attributes.** Optimizations can use S semantics before lowering to LLVM.

Builds the Unified SYCL IR:

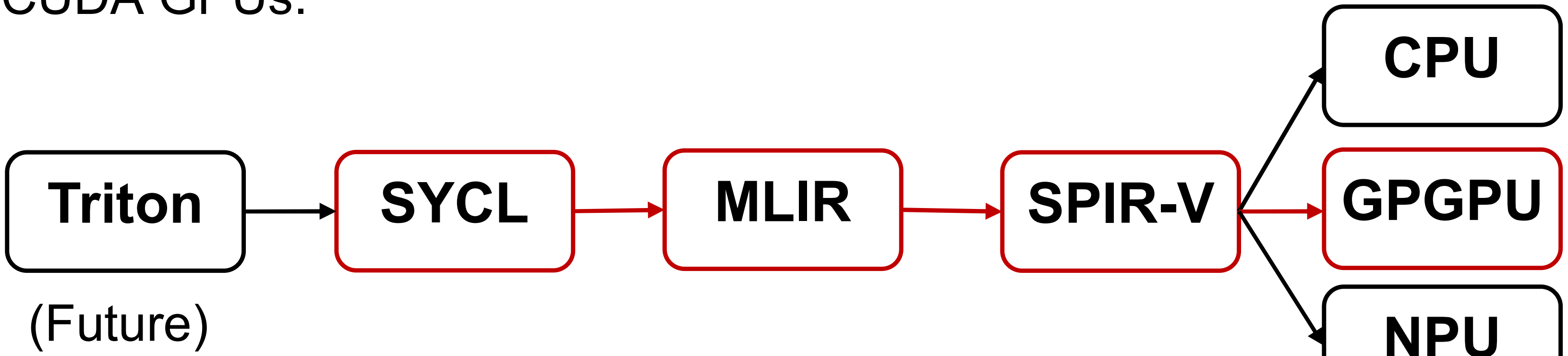
- Device code lowered from C++ → MLIR via **Poly**
- Host code lifted from LLVM IR → MLIR
- Merged into **one nested MLIR module** represen the full SYCL program
- Custom passes apply **SYCL-aware analyses & transformations**.



https://github.com/intel/llvm/tree/sycl-mlir.

Motivation

- SYCL-MLIR aims to introduce MLIR as an intermediate step within the SYCL compilation pipeline, enabling fine-grained optimization and analysis for heterogeneous computing.
- **However, before this contribution, NVIDIA GPUs were not supported**: SYCL-MLIR lacked the dialect-to-NVVM lowering stage. Because of this missing step, the LLVM IR produced by the compiler frontend could not be consumed by the CUDA backend, which made it impossible to complete the full compilation pipeline targeting NVIDIA GPUs.
- This contribution adds the missing adaptation for NVIDIA devices, enabling SYCL-MLIR to successfully compile SYCL programs targeting CUDA GPUs.



Extend AI compiler ecosystem of SYCL

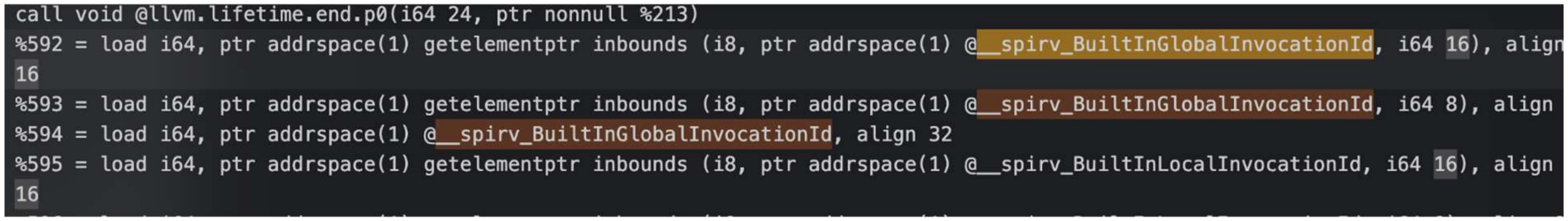
Approach

- This contribution enables SYCL-MLIR’s frontend tool **cgeist** to generate intermediate representation that is compatible with NVIDIA devices. As a result, the compilation pipeline is not fully unified yet.

More concretely, the frontend part of the pipeline is handled by **cgeist** within SYCL-MLIR, while the backend code generation is delegated to the **DPCPP compilation flow**. Our implementation bridges these two stages so that they can work together in practice.

Insight——Built-in variables & Intrinsics

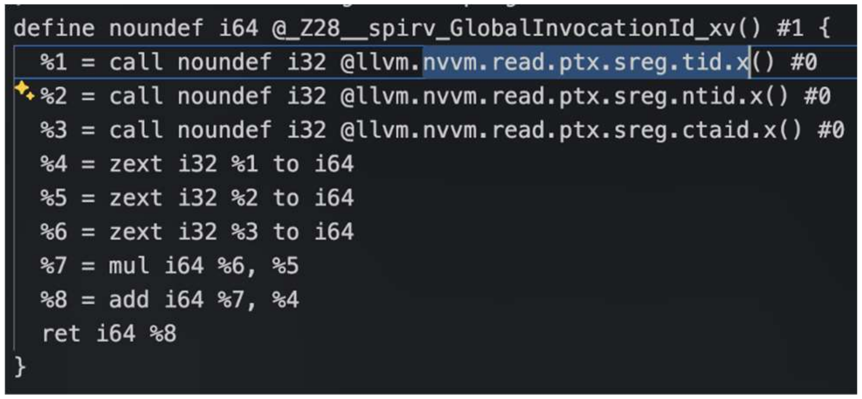
- Currently, the compilation target can only be configured as **SPIR-V**. Through inspecting the LLVM IR emitted by SYCL-MLIR's frontend **cgeist,** the IR generated under the SPIR-V target contains a large number of references to **SPIR-V built-in variables**. Most of these references correspond to operations such as obtaining thread IDs, block dimensions, and other execution parameters.



```
call void @llvm.lifetime.end.p0(i64 24, ptr nonnull %213)
%592 = load i64, ptr addrspace(1) getelementptr inbounds (i8, ptr addrspace(1) @__spirv_BuiltInGlobalInvocationId, i64 16), align
16
%593 = load i64, ptr addrspace(1) getelementptr inbounds (i8, ptr addrspace(1) @__spirv_BuiltInGlobalInvocationId, i64 8), align
%594 = load i64, ptr addrspace(1) @__spirv_BuiltInGlobalInvocationId, align 32
%595 = load i64, ptr addrspace(1) getelementptr inbounds (i8, ptr addrspace(1) @__spirv_BuiltInLocalInvocationId, i64 16), align
16
```

SPIR-V built-in variables

- On NVIDIA hardware, however, these operations are performed using **NVVM intrinsics** rather than built-in variables. This mismatch is the primary reason why the generated program fails to compile for NVIDIA. If these built-in variable references can be replaced with the corresponding NVVM intrinsics, the resulting IR should become executable on NVIDIA GPUs.

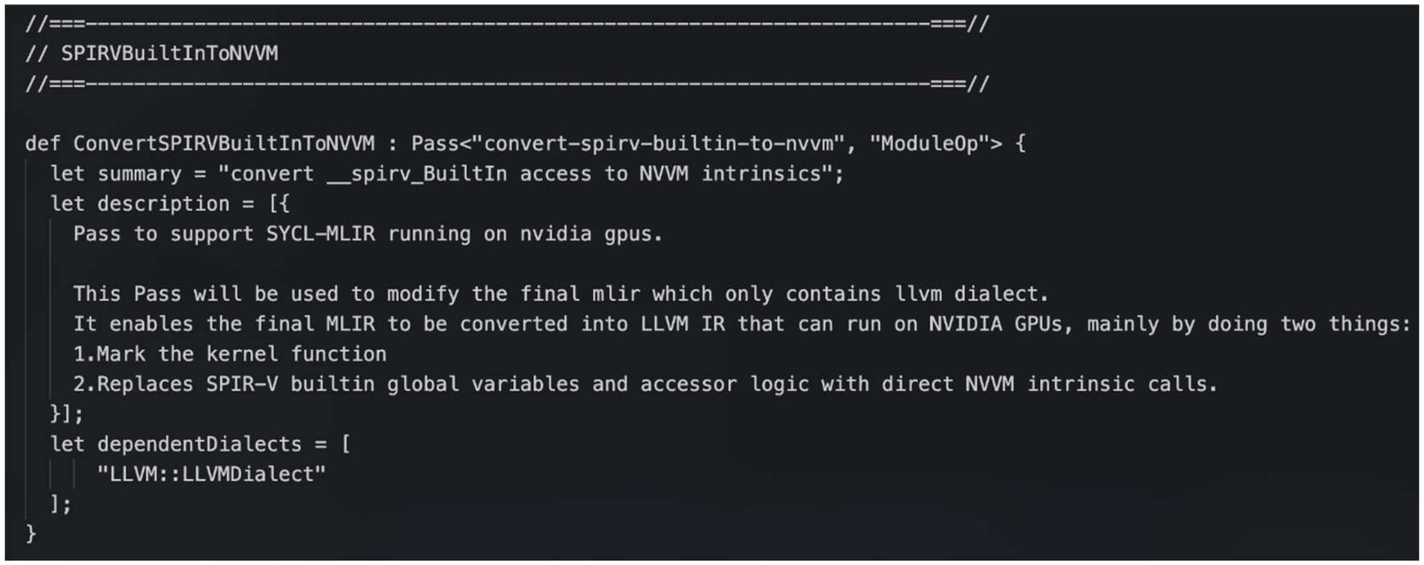


```
define noundef i64 @_Z28__spirv_GlobalInvocationId_xv() #1 {
  %1 = call noundef i32 @llvm.nvvm.read.ptx.sreg.tid.x() #0
  %2 = call noundef i32 @llvm.nvvm.read.ptx.sreg.ntid.x() #0
  %3 = call noundef i32 @llvm.nvvm.read.ptx.sreg.ctaid.x() #0
  %4 = zext i32 %1 to i64
  %5 = zext i32 %2 to i64
  %6 = zext i32 %3 to i64
  %7 = mul i64 %6, %5
  %8 = add i64 %7, %4
  ret i64 %8
}
```

NVVM intrinsics

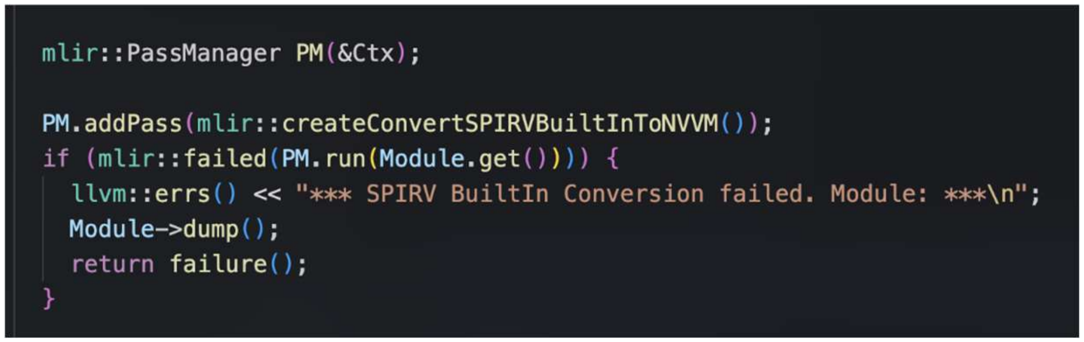
Insight——MLIR pass

- The replacement logic as an **MLIR pass** and registered it in the pipeline. This pass is inserted after **cgeist** finishes lowering to the LLVM dialect, enabling the frontend to emit LLVM IR that is compatible with NVIDIA devices.
- In addition, the kernel function must be annotated with the **nvvm.kernel attribute**; otherwise, the backend compilation pipeline is unable to identify the entry kernel. This attribute insertion is also handled within the same pass.



```
//===----------------------------------------------------------------------===//
// SPIRVBuiltInToNVVM
//===----------------------------------------------------------------------===//

def ConvertSPIRVBuiltInToNVVM : Pass<"convert-spirv-builtin-to-nvvm", "ModuleOp"> {
  let summary = "convert __spirv_BuiltIn access to NVVM intrinsics";
  let description = [{
    Pass to support SYCL-MLIR running on nvidia gpus.

    This Pass will be used to modify the final mlir which only contains llvm dialect.
    It enables the final MLIR to be converted into LLVM IR that can run on NVIDIA GPUs, mainly by doing two things:
    1.Mark the kernel function
    2.Replaces SPIR-V builtin global variables and accessor logic with direct NVVM intrinsic calls.
  }];
  let dependentDialects = [
      "LLVM::LLVMDialect"
  ];
}
```

nvvm.kernel attribute



```
mlir::PassManager PM(&Ctx);

PM.addPass(mlir::createConvertSPIRVBuiltInToNVVM());
if (mlir::failed(PM.run(Module.get()))) {
  llvm::errs() << "*** SPIRV BuiltIn Conversion failed. Module: ***\n";
  Module->dump();
  return failure();
}
```

MLIR Pass

Implementation——Instruction capture

- Instruction capture

DPC++ tool chain

```
clang -### -fsycl -fsycl-targets=nvptx64-nvidia-cuda out.cpp -o out
```

SYCL-MLIR tool chain

```
clang -### -fsycl -fsycl-targets=spir64-unknown-unknown-syclmlir out.cpp -o out
```

1. clang-18 -cc1, compile device code, output .bc
2. llvm-link
3. clang-offload-bundler
4. llvm-link
5. sycl-post-link
6. file-table-tform
7. llvm-foreach
8. file-table-tform
9. clang-offload-wrapper
10. llc
11. append-file
12. clang-18 -cc1, compile, host code
13. ld

1. cgeist, compile device code, output .bc
2. llvm-link
3. clang-offload-bundler
4. llvm-link
5. sycl-post-link
6. file-table-tform
7. llvm-foreach
8. file-table-tform
9. clang-offload-wrapper
10. llc
11. append-file
12. clang-18 -cc1, compile host code
13. ld

Implementation——DPC++ toolchain

We need to capture the instructions for all steps of the DPC++ toolchain, as well as the instruction for the first step of the SYCL-MLIR toolchain (cgeist).

- Then, some modifications are needed for the cgeist instruction:

  cgeist -emit-llvm out.cpp ……. -triple spir64-unknown-unknown-syclmlir …….. -fsycl-instrument-device-code ..……. -o /tmp/out-f57458.bc -x c++ out.cpp

- We need to change ***-triple spir64-unknown-unknown-syclmlir*** to ***-triple nvptx64-nvidia-cuda***, mainly to generate the correct layout information in the intermediate code. We also need to remove ***-fsycl-instrument-device-code***, as this option will conflict with ***-triple nvptx64-nvidia-cuda***.

- Then, execute the modified cgeist instruction to obtain the output .bc file. We may call this file ***cgeist-output.bc***.

Implementation——Instruction execution

- Next, we need to execute the instruction (clang) for the first step of the DPCPP toolchain. This is mainly because:

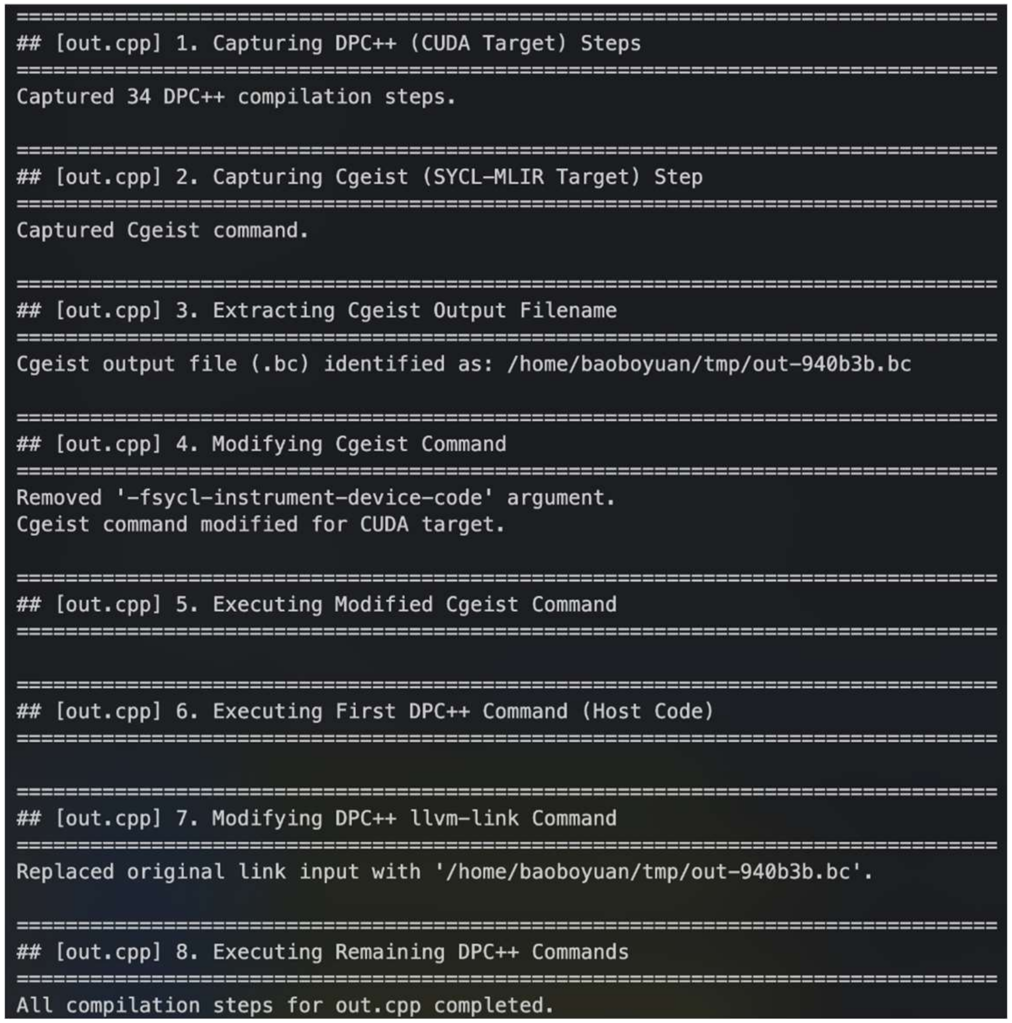
  These files contain information such as **kernel metadata** and **the kernel name**, which helps communication between the host code and the device code. If we don't execute this step, the subsequent DPCPP compilation process will fail due to the lack of these files.

- Next, we'll rename the input file in the second step of the DPCPP toolchain command (llvm-link) to ***cgeist-output.bc***, which is the output of the previous cgeist instruction.

- Then, simply execute the subsequent steps of the DPCPP toolchain to obtain an executable binary file.

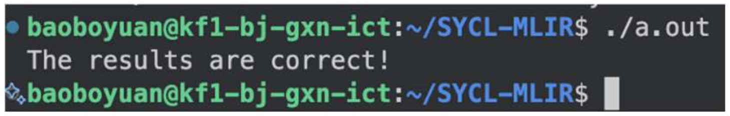
Result

- The overall process:



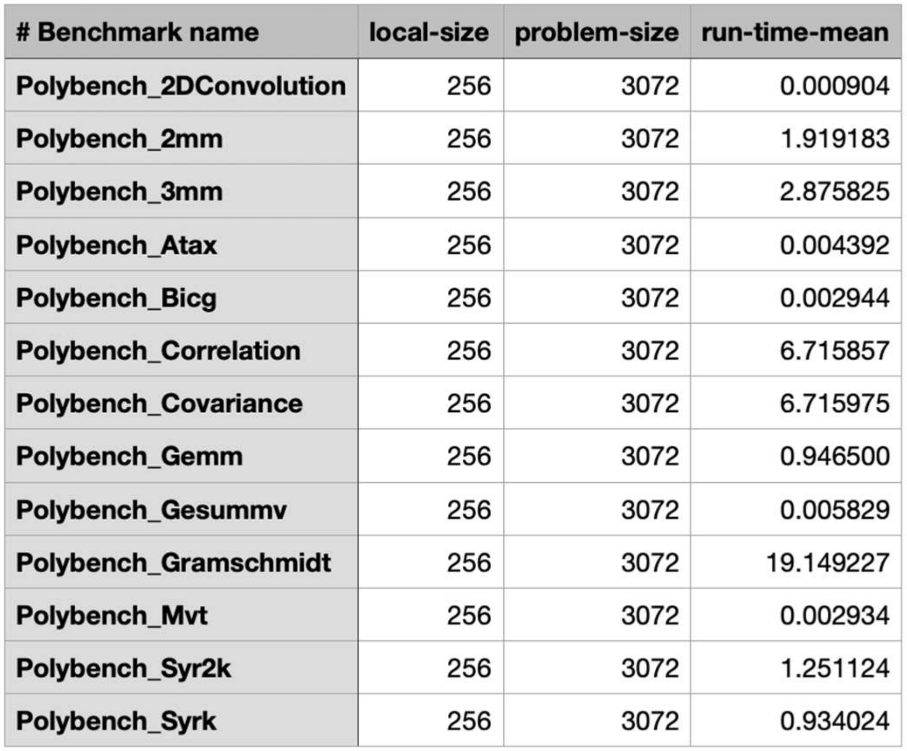
```
================================================================================
## [out.cpp] 1. Capturing DPC++ (CUDA Target) Steps
================================================================================
Captured 34 DPC++ compilation steps.

================================================================================
## [out.cpp] 2. Capturing Cgeist (SYCL-MLIR Target) Step
================================================================================
Captured Cgeist command.

================================================================================
## [out.cpp] 3. Extracting Cgeist Output Filename
================================================================================
Cgeist output file (.bc) identified as: /home/baoboyuan/tmp/out-940b3b.bc

================================================================================
## [out.cpp] 4. Modifying Cgeist Command
================================================================================
Removed '-fsycl-instrument-device-code' argument.
Cgeist command modified for CUDA target.

================================================================================
## [out.cpp] 5. Executing Modified Cgeist Command
================================================================================

================================================================================
## [out.cpp] 6. Executing First DPC++ Command (Host Code)
================================================================================

================================================================================
## [out.cpp] 7. Modifying DPC++ llvm-link Command
================================================================================
Replaced original link input with '/home/baoboyuan/tmp/out-940b3b.bc'.

================================================================================
## [out.cpp] 8. Executing Remaining DPC++ Commands
================================================================================
All compilation steps for out.cpp completed.
```

- execute correctly：



```
baoboyuan@kf1-bj-gxn-ict:~/SYCL-MLIR$ ./a.out
The results are correct!
baoboyuan@kf1-bj-gxn-ict:~/SYCL-MLIR$
```

- sycl-bench/polybench test



# Benchmark name	local-size	problem-size	run-time-mean
Polybench_2DConvolution	256	3072	0.000904
Polybench_2mm	256	3072	1.919183
Polybench_3mm	256	3072	2.875825
Polybench_Atax	256	3072	0.004392
Polybench_Bicg	256	3072	0.002944
Polybench_Correlation	256	3072	6.715857
Polybench_Covariance	256	3072	6.715975
Polybench_Gemm	256	3072	0.946500
Polybench_Gesummv	256	3072	0.005829
Polybench_Gramschmidt	256	3072	19.149227
Polybench_Mvt	256	3072	0.002934
Polybench_Syr2k	256	3072	1.251124
Polybench_Syrk	256	3072	0.934024

THANK YOU