

An Easier, Short Path to Productive Heterogeneous Programming

Source-to-Source CUDA* to SYCL* Code Migration Tool

Intel® DPC++ Compatibility Tool | Coming Soon: Open Source SYCLomatic project



Wang Zhiming, Senior Software Engineer
May 2022



Notices & Disclaimers

All product plans and roadmaps are subject to change without notice.

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Results may vary.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details.
No product or component can be absolutely secure.

Intel technologies may require enabled hardware, software or service activation.

Results have been estimated or simulated.

No product or component can be absolutely secure.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

© Intel Corporation. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

Other names and brands may be claimed as the property of others.

SYCL is a trademark of the Khronos Group Inc.

Agenda

- Easier, Short Path to Productive Heterogeneous Programming
- What is Intel® DPC++ Compatibility Tool
- Open Source project: SYCLomatic (Coming Soon)
- SYCLomatic Usage Flow
- SYCLomatic Architecture
- Migrating VectorAdd Example
- Migration Rule Example
- User Defined Migration Rule
- Summary / Call to Action

Easier, Short Path to Heterogeneous Programming

Diverse architectures have made software development increasingly complex for developers →

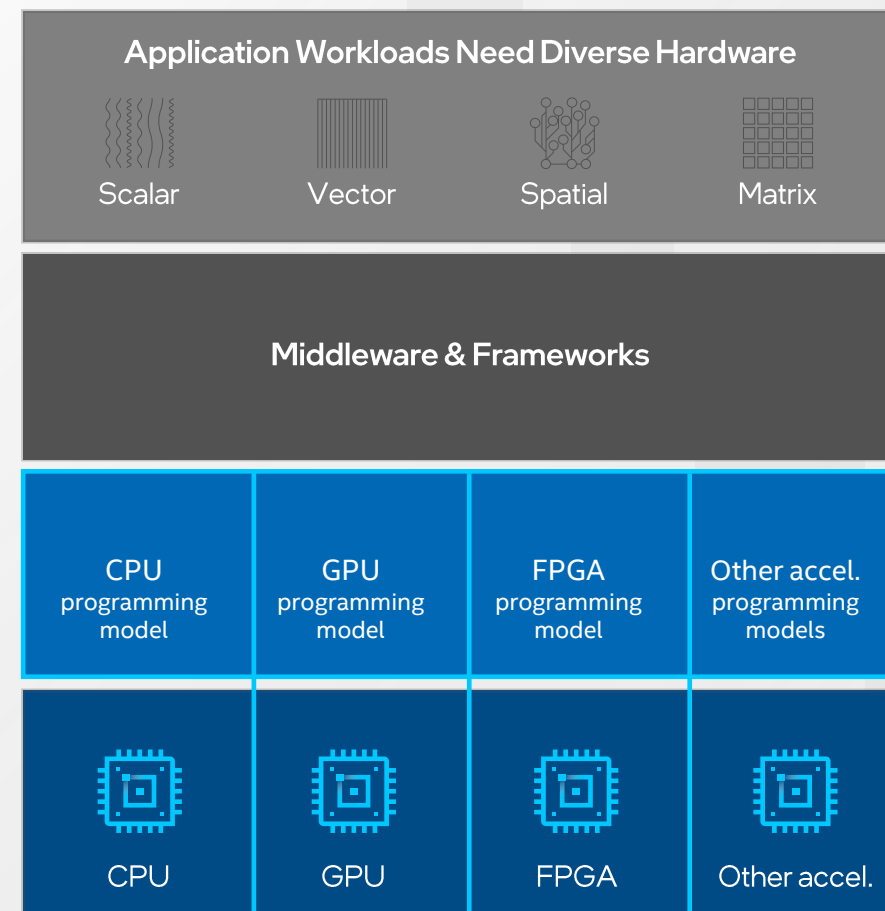
SYCL* with oneAPI open, cross-architecture, standards-based programming

- Allows developers to expand the value of their investments across architectures
- Provides choice & freedom from proprietary, single-vendor lock-in

1
oneAPI

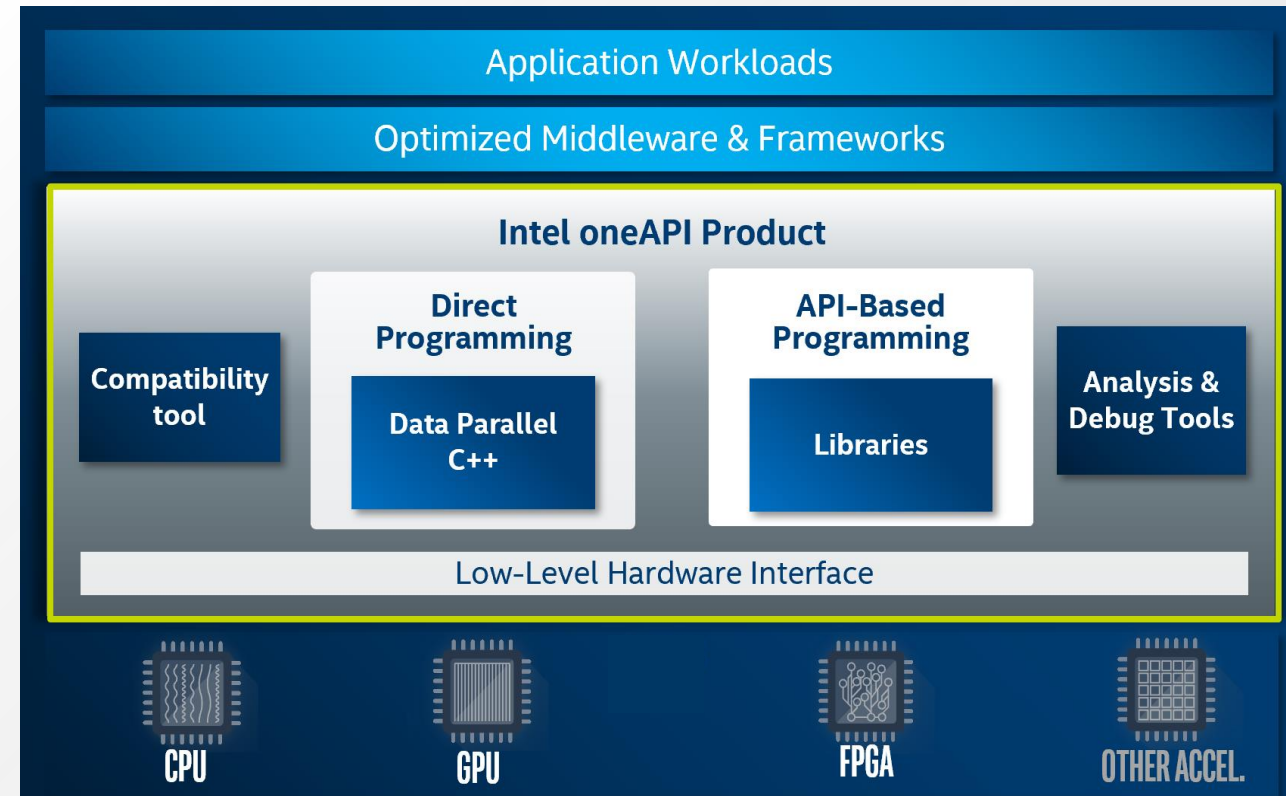
Intel embraces open development to advance ecosystem innovation

- CUDA* to SYCL code migration tool provides developers a productive path to create single-source portable code



What is Intel[®] DPC++ Compatibility Tool

- The tool *assists* in migrating your existing CUDA* code to SYCL* code which can run on any platform that has SYCL compiler support
- The tool ports both CUDA language kernels and library API calls
- Tool was developed by analyzing declaratory code of CUDA and developing migration rules that allow porting of CUDA code to SYCL
- The goal of the tool: make it as easy as possible for developers to migrate their existing CUDA codebase to SYCL
- Intel[®] DPC++ Compatibility Tool is part of the Intel[®] oneAPI Base Toolkit



Intel oneAPI: intel.com/oneAPI

Open Source Project SYCLomatic – Coming Soon

Intel is providing a CUDA* to SYCL* migration tool under an open source license as project SYCLomatic.

- Source to source code migration tool that enables developers to create single-source, portable code for hardware targets regardless of vendor
- Simplifies development while delivering performance and productivity
- Reduces time and costs for code maintenance

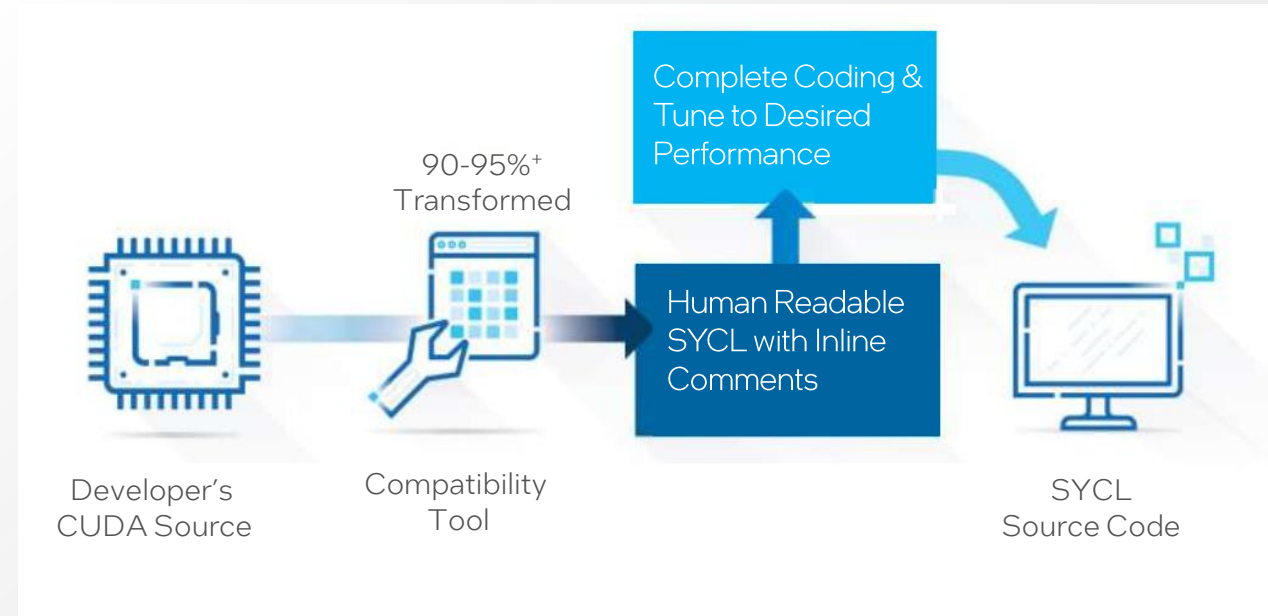
A community to share, collaborate & contribute software technologies

Available on GitHub in the coming weeks

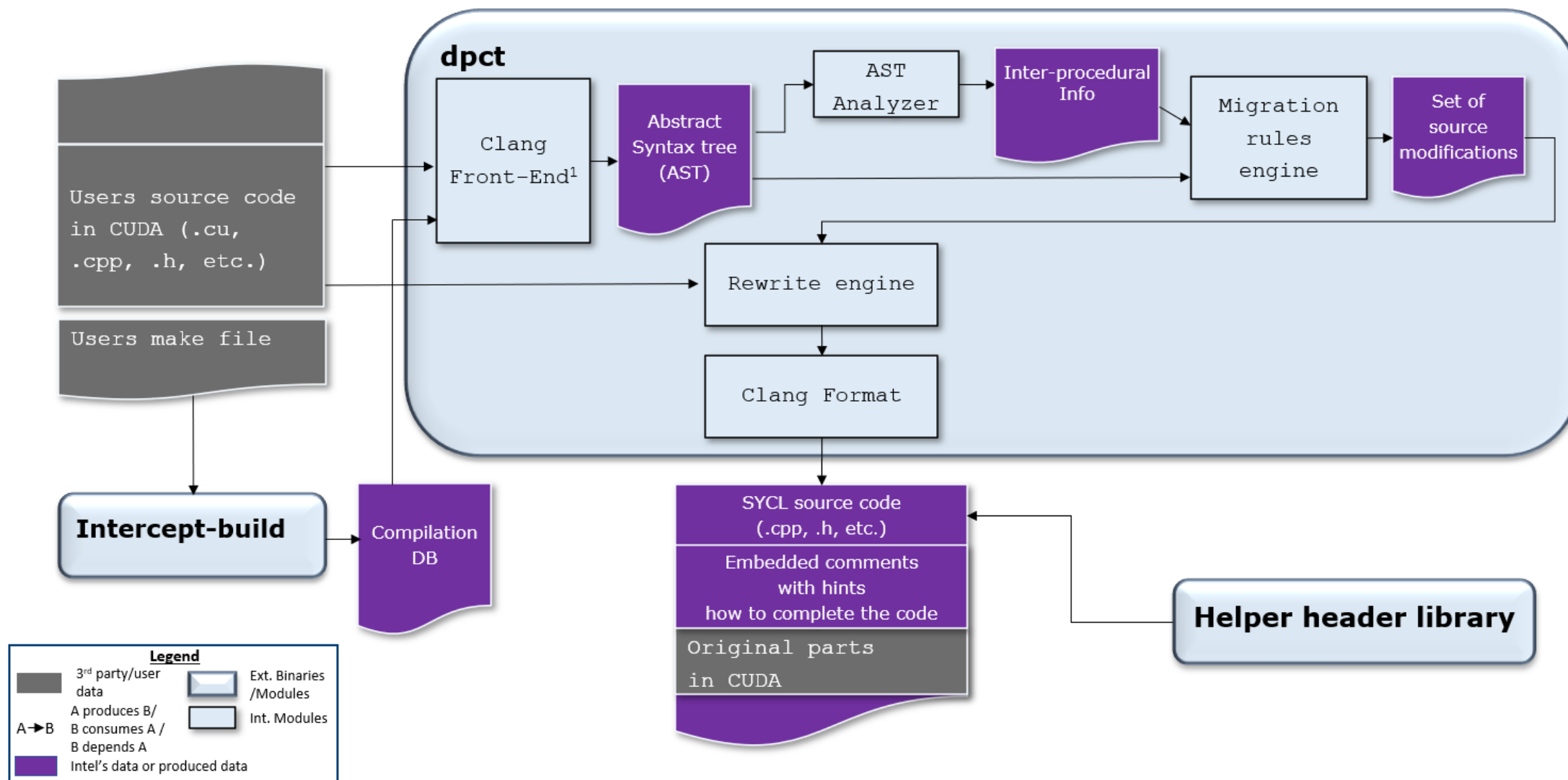
- github.com/oneapi-src/SYCLomatic
- github.com/oneapi-src/SYCLomatic-test
- Use the tool, please provide feedback!

SYCLomatic Usage Workflow

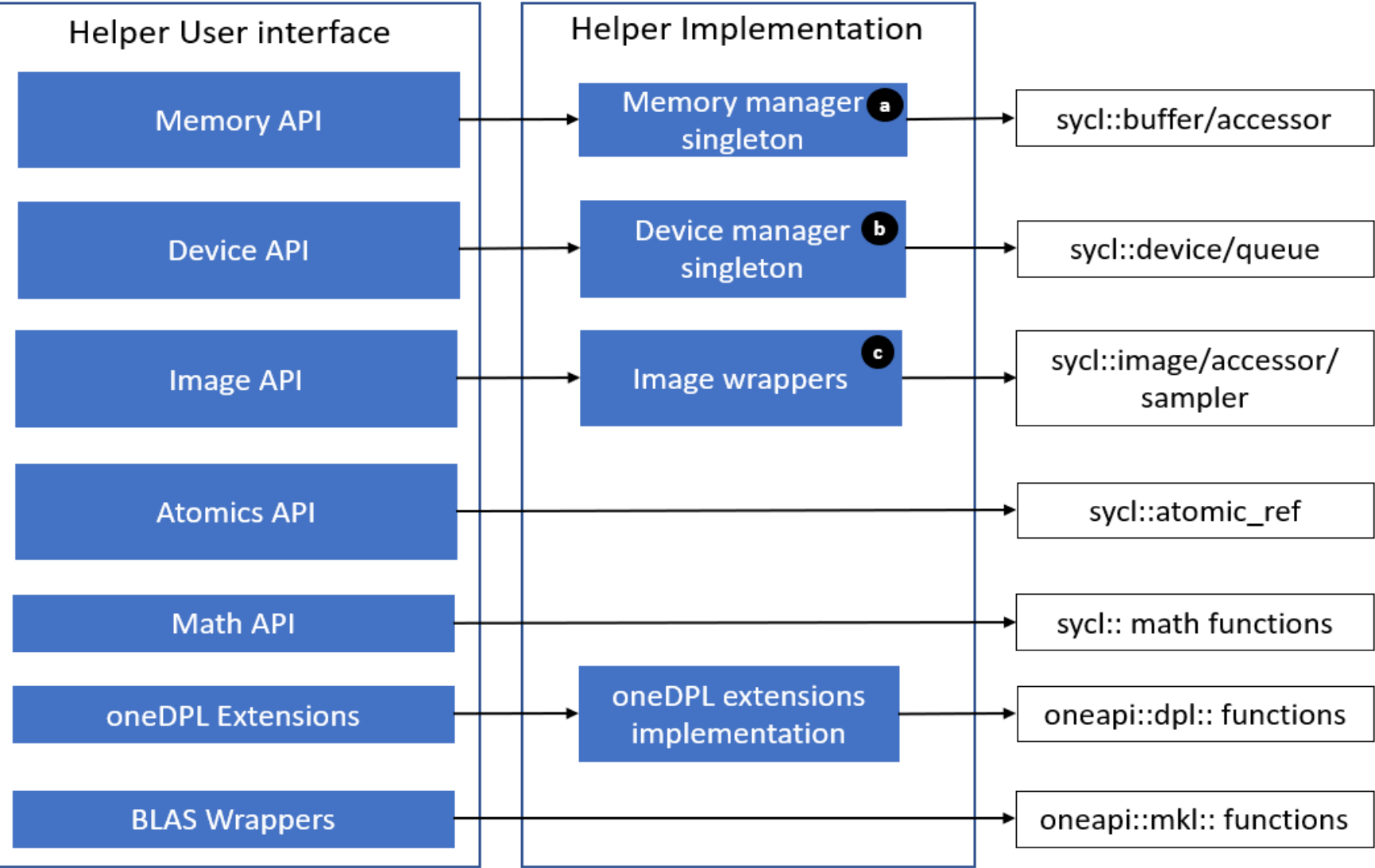
- Collect compilation options of the Developer's CUDA* source from project build scripts, eg. Makefile, vcxproj file
- *Assist* developers migrating code written in CUDA to SYCL by generating SYCL code wherever possible
- Typically, 90%-95%+ of CUDA code automatically migrates to SYCL code
- Inline comments are provided to help developer complete and tune the code



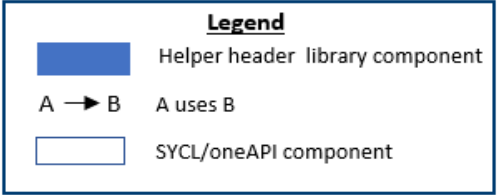
SYCLomatic Architecture (1 of 2 slides)



SYCLomatic Architecture (2 of 2 slides)



Helper header library



- a. Maps host virtual pointers and SYCL buffers
- b. Manages all SYCL capable devices available on the system
- c. Address semantic differences

Migrating VectorAdd Example

CUDA* Code

```
#include <cuda.h>
#include <stdio.h>
#define VECTOR_SIZE 256
```

```
__global__ void VectorAddKernel(float* A, float* B, float* C)
{
    A[threadIdx.x] = threadIdx.x + 1.0f;
    B[threadIdx.x] = threadIdx.x + 1.0f;
    C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];
}
```

```
int main()
{
    float *d_A, *d_B, *d_C;

    cudaMalloc(&d_A, VECTOR_SIZE*sizeof(float));
    cudaMalloc(&d_B, VECTOR_SIZE*sizeof(float));
    cudaMalloc(&d_C, VECTOR_SIZE*sizeof(float));
}
```

SYCL* Code

```
#include <CL/sycl.hpp>
#include <dpct/dpct.hpp>
#define VECTOR_SIZE 256
```

```
void VectorAddKernel(float* A, float* B, float* C, sycl::nd_item<3> item_ct1)
{
    A[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
    B[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
    C[item_ct1.get_local_id(2)] =
        A[item_ct1.get_local_id(2)] + B[item_ct1.get_local_id(2)];
}
```

```
int main()
{
    dpct::device_ext &dev_ct1 = dpct::get_current_device();
    sycl::queue &q_ct1 = dev_ct1.default_queue();
    float *d_A, *d_B, *d_C;

    d_A = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
    d_B = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
    d_C = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
}
```

1

2

3

Migrating VectorAdd Example (2)

CUDA* Code

```
VectorAddKernel<<<1, VECTOR_SIZE>>>(d_A, d_B, d_C);
```

```
float Result[VECTOR_SIZE] = { };  
cudaMemcpy(Result, d_C, VECTOR_SIZE*sizeof(float),  
           cudaMemcpyDeviceToHost);
```

```
cudaFree(d_A);  
cudaFree(d_B);  
cudaFree(d_C);
```

```
for (int i = 0; i < VECTOR_SIZE; i++) {  
    if (i % 16 == 0) {  
        printf("\n");  
    }  
    printf("%f ", Result[i]);  
}
```

```
return 0;
```

SYCL* Code

```
q_ct1.submit([&](sycl::handler &cgh) {  
    cgh.parallel_for(sycl::nd_range<3>(  
        sycl::range<3>(1, 1, VECTOR_SIZE),  
        sycl::range<3>(1, 1, VECTOR_SIZE)),  
        [=](sycl::nd_item<3> item_ct1) {  
            VectorAddKernel(d_A, d_B, d_C, item_ct1);  
        });  
});
```

```
float Result[VECTOR_SIZE] = { };  
q_ct1.memcpy(Result, d_C, VECTOR_SIZE * sizeof(float)).wait();
```

```
sycl::free(d_A, q_ct1);  
sycl::free(d_B, q_ct1);  
sycl::free(d_C, q_ct1);
```

```
for (int i = 0; i < VECTOR_SIZE; i++) {  
    if (i % 16 == 0) {  
        printf("\n");  
    }  
    printf("%f ", Result[i]);  
}
```

```
return 0;
```

4

5

6

7

Migration Rule Example

- Sample code

- Input code: `cudaMalloc(&d_A, vector_size * sizeof(float));`
- Output code: `d_A = sycl::malloc_device<float>(vector_size, dpct::get_default_queue());`

- AST tree

```
| -CallExpr <line:12:3, col:51> 'cudaError_t': 'cudaError'
| | -ImplicitCastExpr <col:3> 'cudaError_t (*) (float **, size_t)' <FunctionToPointerDecay>
| | | -DeclRefExpr <col:3> 'cudaError_t (float **, size_t)' lvalue Function 0x56439167c5a8 'cudaMalloc' 'cudaError_t (float **, size_t)'
| | | -UnaryOperator <col:15, col:16> 'float **' prefix '&' cannot overflow
| | | | -DeclRefExpr <col:16> 'float *' lvalue Var 0x564391659b98 'd_A' 'float *'
| | | -BinaryOperator <col:21, col:49> 'unsigned long' '*'
| | | | -ImplicitCastExpr <col:21> 'unsigned long' <IntegralCast>
| | | | | -ImplicitCastExpr <col:21> 'int' <LValueToRValue>
| | | | | -DeclRefExpr <col:21> 'const int' lvalue Var 0x564391659940 'vector_size' 'const int' non_odr_use_constant
| | | -UnaryExprOrTypeTraitExpr <col:35, col:49> 'unsigned long' 'sizeof' 'float'
```

- AST Matcher

- `callExpr(allOf(callee(functionDecl(hasAnyName("cudaMalloc")) ...)).bind("callExpr"))`

- AST Matcher Action

- Visit the `callExpr` node, analyze the parameters of the `cudaMalloc`, and generate migration result

User Defined Migration Rule

- Provides a way to extent the migration capability by defining migration rule in Yaml file
- Example: Rule “*rule_forceinline*” is used to guide the migration of macro “*__forceinline__*”

<i>- Rule: rule_forceinline</i>	<i># [Required] The unique name of the rule</i>
<i>Kind: Macro</i>	<i># [Required] The kind of the rule [Macro API]</i>
<i>Priority: Takeover</i>	<i># [Required] The priority of the rule [Takeover Default Fallback]</i>
<i>In: __forceinline__</i>	<i># [Required] The target macro name in the input source code</i>
<i>Out: inline</i>	<i># [Required] The migrated name of the macro in output source code</i>
<i>Includes: ["header1.h"]</i>	<i># [Required] A list of header file name which the new macro depends on, can be an empty list</i>

- User defined migration rule target to cover migration of API, Datatype, Class, ENUM type, Macro, Include, Specifier/Qualifier/Attribute
 - Part of feature will be available in next Intel oneAPI release

Summary / Call-to-Action

- Both Intel® DPC++ Compatibility Tool and SYCLomatic assist in migrating your existing CUDA* code to SYCL* code which can run on any platform that has SYCL compiler support
- Intel® DPC++ Compatibility Tool is the Intel product version of SYCLomatic
- SYCLomatic will open source in coming weeks
- Try it
 - Intel® DPC++ Compatibility Tool in Intel® oneAPI Base Toolkit - Free: intel.com/oneAPI-BaseKit
 - github.com/oneapi-src/SYCLomatic coming soon

More Resources

- [SYCLomatic Project](#) on GitHub: [GetStartedGuide.md](#), [Contributing.md](#) guide (coming soon)
- Get started developing
 - [Book](#): Mastering Programming of Heterogeneous Systems using C++ & SYCL
 - [Essentials of SYCL training](#)
 - [The oneAPI samples](#) on Github
- [oneAPI specification](#) and [SYCL](#) specification
- [Intel® DevCloud](#) - A free environment to access Intel® oneAPI Tools and develop and test code across a variety of Intel® architectures (CPU, GPU, FPGA)
- CodeProject: [Using oneAPI to convert CUDA code to SYCL](#)

The Intel logo is centered on a solid blue background. It features the word "intel" in a white, lowercase, sans-serif font. A small, light blue square is positioned above the letter "i". To the right of the word "intel" is a white registered trademark symbol (®).

intel®