Enabling the Use of C++20 Unseq Execution Policy for OpenCL

Po-Yao Chang
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
pychang@pllab.cs.nthu.edu.tw

Tai-Liang Chen
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
tlchen@pllab.cs.nthu.edu.tw

Jenq-Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
jklee@cs.nthu.edu.tw

ABSTRACT

This work facilitates the usage of unsequenced execution policy as seen in ***C++20*** standard library with the newly introduced OpenCL kernel language, ***C++ for OpenCL***. By passing ***unseq***, a global object of type ***unsequenced_policy***, as an argument to selected C++ parallel algorithms, the function would then be vectorized with the help of clang and LLVM. This work complements the introduction of C++ for OpenCL, which brings the core language part of ***C++17*** to OpenCL while leaving out the standard library part. In the best case, we see a whopping 6.9 time speedup.

CCS CONCEPTS

• **Software and its engineering → Specialized application languages**; **Application specific development environments**.

KEYWORDS

OpenCL, C++, SPIR-V, Vectorization, Clang, LLVM

ACM Reference Format:
Po-Yao Chang, Tai-Liang Chen, and Jenq-Kuen Lee. 2021. Enabling the Use of C++20 Unseq Execution Policy for OpenCL. In *International Workshop on OpenCL (IWOCL'21), April 27–29, 2021, Munich, Germany.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3456669.3456674

1 INTRODUCTION

C++ has been gaining traction in the realm of OpenCL. For example, Chen et al. [2] incorporated ***OpenCL C++*** into ViennaCL, while Chang et al. [1] harnessed the power of C++ templates. To our excitement, here comes a new OpenCL kernel language, C++ for OpenCL [6], which brings the core language part of C++17 [4] on top of ***OpenCL C*** without breaking backward compatibility with it. However, the C++ standard library, which takes up a substantial part of the standard, is not supported as of this writing [3], and certainly not parallel algorithms introduced in C++17 and its successor, C++20. This is quite discouraging for C++ practitioners and enthusiasts who are just getting started with OpenCL.

This work makes some functions in C++ standard library work with execution::unseq, as defined in C++20 [5] standard [execpol.objects], which is a global object of type execution::unsequenced_policy, as stated in C++20 [execpol.unseq]. Functions taking execution::unseq as the first parameter are expected to be executed as if it was vectorized; whereas functions without any execution policy parameter or taking execution::seq should be executed as normal (sequentially). For instance, the for_each call as shown in Listing 1 may be vectorized.

```
std::for_each(std::execution::unseq, base_ptr, base_ptr +
      1024 * 1024, [&](auto& v) {
  v = a[idx] + b[idx];
});
```

Listing 1: This for_each call may be vectorized

Inspired by the OpenCL vector types, this work selects functions in C++ standard library, primarily for_each, transform, inner_product, and transform_reduce, and made a version of them that can take execution policy objects as their first parameter. Inside those functions, clang compiler directives are added to vectorize the loops where applicable. The resulting LLVM IR would look as if the OpenCL vectors were used. Listing 2 shows the difference between scalar addition and int4 addition and how int4 is mapped to LLVM IR. Users wanting to use those functions with execution::unseq can simply include the headers we made.

```
%29 = add nsw i32 %28, %26          ; scalar addition
%29 = add nsw <4 x i32> %28, %26    ; OpenCL vector (int4)
     addition
```

Listing 2: OpenCL vector are mapped to LLVM vector in LLVM IR layer.

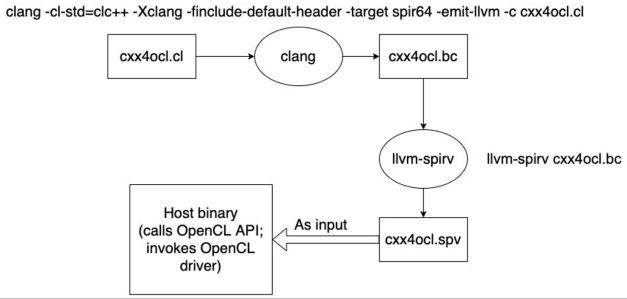

Figure 1: C++ for OpenCL compilation steps(from text to SPIR-V binary) and how it's consumed

Then finally comes the compilation steps, which looks like this: (1) clang fed with C++ for OpenCL code outputs vectorized LLVM IR,

IWOCL'21, April 27–29, 2021, Munich, Germany

ACM ISBN 978-1-4503-9033-0/21/04…$15.00
https://doi.org/10.1145/3456669.3456674

(2) llvm-spirv converts LLVM IR to SPIR-V, (3) host code leverages host API to compile SPIR-V to target assembly and execute it on an OpenCL platform. Figure 1 portrays this.

In the end, this work also benchmarks applications using the functions this work implements with and without execution::unseq on both CPU and iGPU. In the best case, this work results in a speedup up to over 6.9 times compared to functions without unseq (non-vectorized).

2 METHOD

First and foremost, we explain what unseq really is. According to the C++20 standard, execution policies are unspecified classes, and std::execution::unseq is just a global object of type unsequenced_policy [5]. Hence, we define the types as follows in a header file, and define a global object unseq of type unsequenced_policy accordingly as indicated in Listing 3.

```
struct unsequenced_policy {};
struct sequenced_policy {};
constexpr unsequenced_policy unseq{};
```

Listing 3: Definition of unseq and its type

Listing 4 shows a complete C++ for OpenCL kernel which serves as an example of user code, as opposed to Listing 3 and 5 which dwell in header files. A type alias declaration for DataTy preceding this function is expected. Note that unseq is the first argument.

```
__kernel void vadd(__global DataTy const* a, __global
    DataTy const* b, __global DataTy* c) {
  auto idx = get_global_id(0);
  auto base_ptr = c + idx;
  std::for_each(std::execution::unseq, base_ptr, base_ptr
      + 1024 * 1024, [&](auto& v) {
    v = a[idx] + b[idx];
  });
}
```

Listing 4: A complete C++ for OpenCL kernel doing vector addition (typical user code)

Our goal is to make the call in Listing 4 meet the semantics of unseq. We achieved this by overloading functions with execution policy types, and hence, the for_each call in Listing 4 resolves to the call in Listing 5, in which a vectorization compiler directive is added.

```
template <typename ForwardIterator, typename Function>
Function for_each(execution::unsequenced_policy exec,
                  ForwardIterator first,
                  ForwardIterator last, Function f) {
#pragma clang loop vectorize(enable) vectorize_width(
    VEC_WIDTH)
  for (; first != last; ++first)
    f(*first);
  return f;
}
```

Listing 5: The vectorized version of for_each

In the case of Listing 5, during the first step of the compilation process mentioned in the second last paragraph of section 1, Clang would then inline the function object call operator as in ***f(*first)*** and vectorizes the loop with clang directive by leveraging LLVM library. The resulting LLVM bitcode would contain LLVM vector types which is what OpenCL vector types are also mapped to, as illustrated in Listing 2 and the third paragraph of section 1.

3 EXPERIMENT

Our experiments are conducted on an Intel Core i7-7700 CPU 3.60GHz with a built-in HD Graphics 630 graphics card (Intel Gen 9 HD Graphics), which means this experiment is run on both CPU and iGPU. Clang 10.0.1 is used to compile kernel code to LLVM IR, and llvm-spirv, which is built against LLVM 10.0.1, is used to translate LLVM IR to SPIR-V binary.

As illustrated in Figure 2, we benchmark against 2 kernels (1D convolution, and Sum of Absolute Differences). Unoptimized kernels (not vectorized at all) are used as the baseline which is represented as blue bars. Vectorization with vector width 4 and 16 are represented as orange and grey bars respectively. Incidentally, SAD on GPU far outshines other cases, with a speedup of vector widths 4 and 16 being 3.4X and 6.9X respectively.

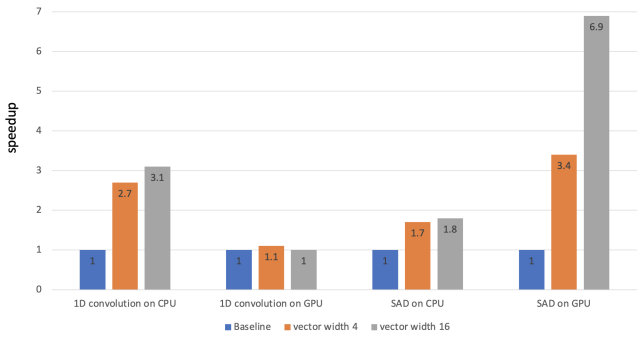

Figure 2: Speedup with various vector widths

4 CONCLUSION

This work brings a part of the C++20 parallel algorithm to C++ for OpenCL. Not to our surprise, results vary from kernel to kernel, since the vendor's graphics compiler, which turns SPIR-V into actual machine code, is beyond our control. In the best-case scenario, a speedup of 6.9 is observed.

REFERENCES

[1] Chia-Hsuan Chang, Chun-Chieh Yang, Jenq-Kuen Lee, and Yung-Chia Lin. 2019. Case Study: Support OpenCL Complex Class for Baseband Computing. In *Proceedings of the International Workshop on OpenCL (IWOCL'19)*. ACM, Article 23, 2 pages. https://doi.org/10.1145/3318170.3318175

[2] Tai-Liang Chen, Shih-Huan Chien, and Jenq-Kuen Lee. 2018. ViennaCL++: Enable TensorFlow/Eigen via ViennaCL with OpenCL C++ Flow. In *Proceedings of the International Workshop on OpenCL (IWOCL '18)*. ACM, Article 28, 2 pages. https://doi.org/10.1145/3204919.3207894

[3] Khronos Group. 2020. *The C++ for OpenCL 1.0 Programming Language Documentation.* https://github.com/KhronosGroup/OpenCL-Docs/releases/download/v3.0.6/CXX_for_OpenCL.pdf

[4] ISO. 2017. *ISO/IEC 14882:2017: Programming languages — C++.*

[5] ISO. 2020. *ISO/IEC 14882:2020: Programming languages — C++.*

[6] Anastasia Stulova, Neil Hickey, Sven van Haastregt, Marco Antognini, and Kevin Petit. 2020. The C++ for OpenCL Programming Language. In *Proceedings of the International Workshop on OpenCL (IWOCL '20)*. ACM, Article 3, 2 pages. https://doi.org/10.1145/3388333.3388647