arXiv:2304.01841v1 [hep-ex] 4 Apr 2023

异构计算环境下液氩时间投影室模拟的可移植编程模型探索：OpenMP 与 SYCL 的比较

Meifeng Lin[1],**Zhihua Dong**[1],**Tianle Wang**[1],**Mohammad Atif**[1],**Meghna Battacharya**[2],**Kyle Knoepfel**[2],**Charles Leggett**[3],**Brett Viren**[1],**Haiwang Yu**[1]

[1] 布鲁克海文国家实验室，厄普顿，纽约州 11973，美国[2] 费米国家加速器实验室，巴达维亚，伊利诺伊州 60510，美国[3] 劳伦斯伯克利国家实验室，伯克利，加利福尼亚州 94720，美国

摘要。 计算领域的发展催生了多样化硬件架构的广泛应用，不同类型的图形处理器和其他计算加速器也日益普及。为在异构计算环境中高效利用这些架构，人们开发了多种编程模型，以实现不同计算系统之间的可移植性和性能，例如 Kokkos、SYCL、OpenMP 等。作为高能物理计算卓越中心（HEP-CCE）项目的一部分，我们通过若干具有代表性的用例，研究这些不同的编程模型是否适用于实验高能物理工作流，以及如何应用。其中一个用例是液氩时间投影室（LArTPC）模拟，这对 LArTPC 探测器设计、验证和数据分析至关重要。在此前研究如何使用 Kokkos 将 Wire-Cell 工具包（WCT）中的 LArTPC 模拟移植到图形处理器的基础上，我们进一步探索了将 OpenMP 和 SYCL 作为 WCT 的潜在可移植编程模型，旨在让 LArTPC 模拟能够使用多样化的计算资源。本文介绍了我们如何在 WCT 的 LArTPC 模拟模块中运用 OpenMP 和 SYCL 的相关特性。我们还展示了 OpenMP 和 SYCL 两种实现方案在多核 CPU、NVIDIA 和 AMD GPU 上的性能基准测试结果，并在适当之处比较不同编译器的表现。

1. 引言

计算在实验高能物理（HEP）的许多方面都发挥着核心作用，包括探测器模拟、事例重建和终端用户数据分析。传统上，实验高能物理软件主要面向同构多核 CPU 编写。随着下一代高能粒子实验的数据量和数据率预计将增长数个数量级，仅依赖 CPU 计算资源可能无法满足处理需求。其中一个实验是计划中的长基线中微子设施（LBNF）/深地中微子实验（DUNE），这是一项旨在研究中微子基本性质的国际项目。DUNE 采用液氩时间投影室（LArTPC）技术，预计仅远端探测器（FD）的 LArTPC 模块每年就会产生 30 PB 数据 [1]。这至少比当前及以往中微子实验产生的数据高一个数量级。单次

预计四个远探测器模块之一的单次读出将以约 0.1 Hz 的频率产生约 5 GB 的原始打包数据。此外，由于观测到与超新星中微子暴发相符的活动，预计远端探测器平均每月还会产生约 500 TB 数据。尽管绝大多数此类触发事件都源于本底涨落，但仍必须对每个事件进行全面分析。及时分析 DUNE 远探测器数据，可能需要访问并高效利用现代高性能计算（HPC）系统；这类系统通常采用异构配置，包含主机端多核 CPU 和通用图形处理器（GPU）等计算加速器。为了高效利用这些 HPC 资源，现有软件需要重构，以便将工作负载中某些计算密集型部分转移到 GPU 上运行。然而，高能物理软件必须经过严格的验证与确认流程，因此我们既不希望，也不应当每当出现新的硬件架构时就重写软件。如今这一点尤为重要，因为至少有三家不同的 GPU 厂商：AMD、英特尔和 NVIDIA，它们的 GPU 使用各自不同的原生应用程序编程接口（API）。

因此，我们希望探索可移植编程模型，使我们能够通过单一应用程序接口编写软件，同时在多种 CPU 和 GPU 架构上实现良好的性能。这是高能物理计算卓越中心（HEP-CCE）项目所采用策略的一部分；该项目旨在研究如何让高能物理工作流在各种高性能计算系统上高效运行。DUNE 尤其关注的一个测试用例是 LArTPC 信号模拟，它模拟高能带电粒子穿过探测器时产生的电离电子在液氩时间投影室导线上感应出的电流。在先前的研究中， [2, 3], 我们已表明, Kokkos [4], 这一支持多种架构的 C++抽象层，能够在保持熟悉的 C++语法和易用性的同时，提供良好的性能与可移植性。本文介绍我们将 LArTPC 信号模拟移植到另外两种可移植编程模型 SYCL [5] 和 OpenMP [6]的实践经验。第 2 节简要介绍 Wire-Cell 工具包（WCT）中的 LArTPC 信号模拟。第 3 节概述 Kokkos、SYCL 和 OpenMP。第 4 节介绍 SYCL 和 OpenMP 的实现细节。第 5 节记录 Kokkos、SYCL 和 OpenMP 的性能基准测试结果。最后，我们在第 6 节总结实践经验。

2. Wire-Cell 工具包中的 LArTPC 信号模拟

Wire-Cell 工具包（WCT） [7] 目前提供了先进的 LArTPC 信号模拟实现。LArTPC 信号模拟包括三个主要步骤，详见参考文献 [2]:

(i) **光栅化**：将各个电离电子群分箱到大小不一的“斑块”中，其尺寸通常为 $\mathcal{O}(20 \times 20)$ 个元素。大多数浮点运算都发生在这一步，因此可通过并行化显著提升效率，因为不同斑块上的操作可以并行执行。(ii) **散射加法**：将这些斑块叠加并求和，形成一个更大的网格，其范围覆盖所有斑块并集的纵向时间和横向空间。该网格通常是一个包含 $\mathcal{O}(10,000 \times 10{,}000)$ 个元素的数组。这一步需要使用原子加法操作。(iii) **卷积**：随后，将该大网格与尺寸为$\mathcal{O}(100 \times 100)$ 的探测器响应进行卷积，以获得模拟信号。这里的主要计算是快速傅里叶变换（FFT）。

为简化对不同可移植编程模型的测试，我们基于 WCT 开发了一个用于液氩时间投影室信号模拟的独立代码 wire-cell-gen，并分别实现了 Kokkos、SYCL 和 OpenMP 版本 [8, 9, 10]。在每种实现中，上述三个步骤都包含在 DepoTransform 函数中。

3. Kokkos、SYCL 和 OpenMP

Kokkos [4] 是一种 C++ 抽象层，提供与架构无关的应用程序接口，以实现可移植性。自问世以来，它凭借对多种硬件架构的支持而日益普及，包括多核/众核 CPU、NVIDIA GPU、AMD GPU 和 Intel GPU。

SYCL [11] 是一种由 Khronos Group 开发的开放标准编程模型，支持编写可在异构系统上运行的 C++应用程序，例如 CPU、GPU 和 FPGA 等。开发者可以编写单一源代码，并使用受支持的编译器进行构建，以针对不同的硬件平台使用不同的后端。如果编译器支持，也可以在运行时选择目标平台。

OpenMP [6] 是一种应用程序接口，包含用于在 CPU 上实现多线程，以及在异构架构上进行“目标卸载”的编译器指令和运行时库例程。使用合适的编译器，OpenMP 代码有可能在 CPU、图形处理器和 FPGA 上运行。

4. 实现细节

4.1. 面向厂商优化的快速傅里叶变换和随机数生成器的封装器

由于 wire-cell-gen 使用了计算开销较大的快速傅里叶变换和随机数生成器（RNG），因此我们希望能够使用硬件厂商提供的优化库。目前，Kokkos、SYCL 和 OpenMP 均未提供一种可移植的机制来使用这些优化库。因此，与我们的 Kokkos 实现 [3], 类似，我们编写了自己的封装器，以便使用厂商针对快速傅里叶变换和随机数生成器优化的库。对于快速傅里叶变换，我们在 NVIDIA GPU 上使用 cuFFT，在 AMD GPU 上使用 rocFFT。对于主机 CPU 代码，我们使用非并行的 fftw，因为原始 WCT 代码也采用了该库。对于随机数生成器，我们在 NVIDIA GPU 上使用 cuRAND，在 AMD GPU 上使用 rocRAND，而在 CPU 上使用 random123 [12, 13] 。

4.2. SYCL

在参考文献 [3] 中，我们介绍了将 wire-cell-gen 代码移植到 Kokkos 的流程及其性能。由于我们已在 Kokkos 实现中将数据流和并行执行模式设计为适用于图形处理器，因此 SYCL 实现直接以 Kokkos 版本为基础。在许多情况下，SYCL 语法与 Kokkos 非常相似，因此从 Kokkos 移植到 SYCL 相对简单。

我们创建了 Array1D 和 Array2D 类，用来替代 Kokkos 实现中的 KokkosArray 类（该类是 KokkosView 的封装）。这两个类只包含简单的指针和大小信息，以及少量方法。借助它们，许多行 Kokkos 代码只需进行如下所示的简单转换，即可轻松改写为 SYCL 代码。

```
Kokkos::deep_copy(sps_f, spf_h); ->   sp_fs.copy_from(sps_h); auto sp_ts  =  -
>  auto sp_ts   =KokkosArray::idft_cr(sp_fs,1) ; SyclArray::idft_cr(sp_fs,1) ;
```

4.3. OpenMP

对于许多基于 CPU 的项目，使用 OpenMP 移植时，我们可以从添加 #pragma omp 来实现数据移动和循环并行化入手，而无需大幅修改 CPU 代码。然而，原始 WCT 代码的结构并不适合这种并行化方式，因此我们从 Kokkos 版本开始着手。我们使用一维数组表示所有数据。此外，我们还通过 #pragma omp target data map 手动执行数据移动，以消除不必要的数据移动并降低峰值内存使用量。

Kokkos 版本与 OpenMP 版本的一个主要区别在于：Kokkos 支持 GPU 前缀和运算（扫描），而 OpenMP 目前尚不支持。因此，我们修改了其中一个内核的算法，以移除对 GPU 扫描运算的使用。这项改动只会带来可忽略的性能损失，并增加少量额外内存开销。

5. 性能评估

5.1. 硬件平台与软件环境

为确保性能基准测试的一致性，我们在布鲁克海文国家实验室使用了同一台工作站（lambda1）。该工作站配备 NVIDIA V100 GPU、AMD Raedon Pro VII GPU，以及拥有 48 个超线程的 AMD 24 核 Ryzen Threadripper 3960X CPU。这与用于评估 Kokkos 实现性能的平台相同 [3]。

SYCL 和 OpenMP 目标卸载的编译器支持仍在持续完善中。我们尝试了几种 SYCL 编译器，包括 Intel LLVM 编译器、OpenSYCL（前身为 hipSYCL）和 Intel oneAPI 编译器（dpcpp）。它们都无法成功编译我们的代码并支持所有后端。不过，对于特定目标后端，至少有一种编译器能够正常工作。OpenMP 的情况更好：LLVM Clang 编译器在我们测试的所有后端上都运行良好。我们还尝试了 GCC 和 NVIDIA HPC SDK（nvc++）进行 OpenMP 目标卸载。NVIDIA HPC SDK 仅适用于 NVIDIA GPU。至于 GCC，它能够通过 OpenMP 目标卸载成功编译并运行各种单元测试。不过，尝试使用 GCC 编译并运行 wire-cell-gen-openmp 时，我们遇到了一些运行时错误。能够成功编译我们代码的编译器汇总见表 1。

编程模型	编译器	版本	目标架构
SYCL	英特尔/LLVM (clang++)	nightly-20220425	NVIDIA GPU
			AMD GPU
	OpenSYCL	v0.93	CPU
	Intel oneAPI (dpcpp)	2022.02	CPU
OpenMP	LLVM/Clang	15	NVIDIA GPU
			AMD GPU
			CPU
	NVIDIA HPC SDK（nvc++）	21.9	NVIDIA GPU

表 1. 本研究使用的 SYCL 和 OpenMP 编译器。

5.2. 性能比较

我们通过与 Kokkos 版本对比，验证了 SYCL 和 OpenMP 实现的正确性。为比较各实现针对不同架构的性能，我们测量了关键计算内核——光栅化、散射加法和快速傅里叶变换——以及 DepoTransform 的总耗时。每种实现／后端的计时结果均取 20 次运行的平均值。图 1 至图 4 展示了各后端的计时结果，其中 SYCL CPU 后端的结果来自 oneAPI 编译器，而 OpenMP NVIDIA GPU 后端的结果来自 LLVM/Clang 编译器。我们还列出了 Kokkos 实现的计时结果 [3] 以供比较。

图 1 显示，在同一平台上，不同编程模型的总运行时间相近，差异小于 20%。不过，从分解结果可以看出，三种编程模型在基于原子操作的散射加法内核（图 3）上的性能差异很大。Kokkos 实现的性能最佳，而 OpenMP 实现的速度慢约 5–10 倍。此外，在 AMD GPU 上，OpenMP 实现在首次调用随机数生成器时开销较大，导致栅格化内核的计时差异较大（图 2）。对于快速傅里叶变换（图 4），由于不同实现都使用相同的优化快速傅里叶变换库，性能差异很小。

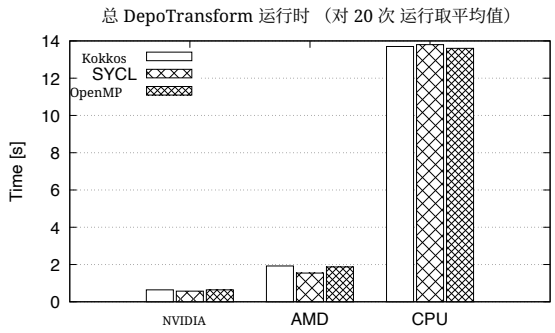

图 1. DepoTrans 总运行时间对比。

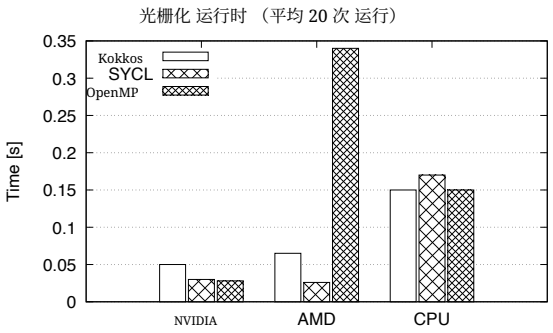

图 2. 栅格化内核耗时对比。

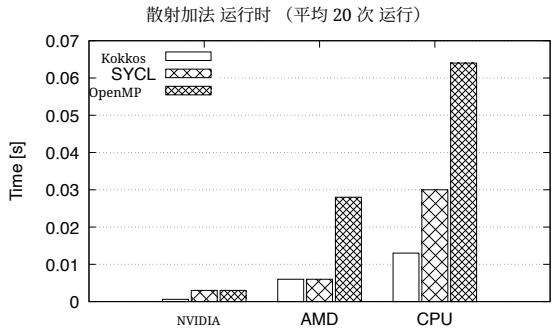

图 3. 散射加法内核耗时比较。

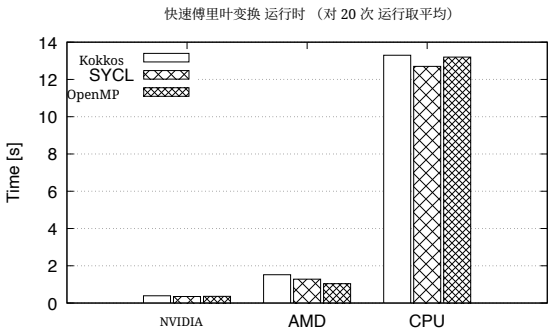

图 4. 快速傅里叶变换耗时比较。

5.3. 性能调优

为获得上述性能结果，我们对 SYCL 和 OpenMP 实现都进行了一些性能调优。例如，在 OpenMP 实现中，num_threads 和 num_teams 的默认取值有时并非性能最佳。另一个例子是，在栅格化中，有几个内核包含两层嵌套循环，外层循环较大，内层循环较小。我们发现，对于某些内核，两个循环都并行化效果最好；而对于另一些情况，只并行化外层循环效果最佳。对于 SYCL CPU 后端，使用 dpcpp 编译程序有两种方式：即时编译（JIT）或预先编译（AOT）。如果未通过 -fsycl -fsycl-targets=spir64 x86 64 -Xs 标志显式指定 CPU 目标平台，默认采用即时编译（JIT）。在 WCT 中，对于 CPU 后端，与预先编译（AOT）相比，即时编译（JIT）使栅格化的计时增加了 1 秒。图 1–4 仅展示 SYCL AOT 的结果。

5.4. 编译器比较

由于我们能够使用 OpenSYCL 和 Intel oneAPI 两种编译器为 CPU 后端编译 SYCL 代码，因此我们在图 5 中展示了栅格化内核耗时随 CPU 线程数变化的情况。OpenSYCL 在最多 16 个线程时表现出更好的扩展性，之后则是 oneAPI 优于 OpenSYCL。这可能是因为这些编译器使用了不同的运行时库：OpenSYCL 使用 OpenMP 后端，而 oneAPI 使用 Intel TBB。

对于 OpenMP 实现，我们成功使用 LLVM/Clang 编译器和 NVIDIA HPC SDK 将代码编译为可在 NVIDIA V100 GPU 上运行。正如图6所示，两种编译器的整体性能相近。

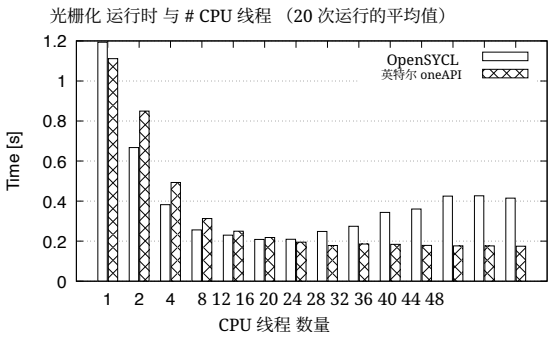

图 5. 使用两种不同编译器的 SYCL 实现中 CPU 上光栅化时间的比较。

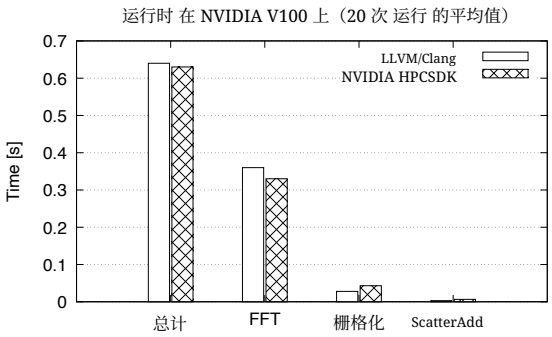

图6. OpenMP 实现中 LLVM/Clang 编译器与 NVIDIA HPC SDK 的比较。

6. 总结与展望

我们成功地在 Wire-Cell 工具包中采用三种可移植编程模型实现了 LArTPC 信号模拟: Kokkos、SYCL 和 OpenMP。我们展示了，在所测试的模拟参数下，这三种编程模型都能达到相近的计算性能。它们都支持我们编写一份代码，以适配不同的计算架构。目前，我们正在研究将 stdpar 模型用于 WCT 的方案 [14] ，并将在未来的论文中详细比较这些不同的编程模型。

致谢

本研究得到美国能源部科学办公室高能物理办公室高能物理计算卓越中心（HEP-CCE）的支持, 项目编号为 B&R KA2401045。本研究使用了国家能源研究科学计算中心（NERSC）的计算资源。该中心是美国能源部科学办公室用户设施，位于劳伦斯伯克利国家实验室，由合同编号 DE-AC02-05CH11231 项下运营。我们衷心感谢布鲁克海文国家实验室物理系电子探测器组的 Wire-Cell 团队以及科学数据与计算中心提供的支持；该实验室由美国能源部通过合同编号 DE-SC0012704 提供支持。

参考文献

[1] Abed Abud A *et al.* (DUNE) 2022年 (*Preprint* 2210.15665)[2] Yu H、Dong Z、Knoepfel K、Lin M、Viren B 和 Yu K 2021 *EPJ Web of Conferences* 第 251 卷（EDP Sciences）第 03032 页[3] Dong Z、Knoepfel K、Lin M、Viren B 和 Yu H 2023 *Journal of Physics: Conference Series* 第 2438 卷（IOP Publishing）第 012036 页[4] Edwards H C、Trott C R 和 Sunderland D 2014 *Journal of parallel and distributed computing* **74** 3202–3216[5] Reyes R 和 Lom¨uller V 2016 *Parallel Computing: On the Road to Exascale*（IOS Press）第 673–682 页[6] OpenMP openmp.org[7] Wire-Cell 工具包 https://github.com/WireCell/wire-cell-toolkit[8] Wire-Cell 信号模拟 Kokkos 实现 https://github.com/WireCell/wire-cell-gen-kokkos[9] Wire-Cell 信号模拟 SYCL 实现 https://github.com/WireCell/wire-cell-gen-sycl[10] Wire-Cell 信号模拟 OpenMP 实现 https://github.com/WireCell/wire-cell-gen-openmp[11] SYCL https://www.khronos.org/sycl/[12] Random123 https://github.com/DEShawResearch/random123[13] Salmon J K、Moraes M A、Dror R O 和 Shaw D E 2011 *Proceedings of 2011 international conference forhigh performance computing, networking, storage and analysis* 第 1–12 页[14] stdpar https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/