MinTallySYCL: SYCL Conversion of OpenMP-Based Monte Carlo Tallying

Vincenzo Raia
Department of Computer Science
University of Salerno
Italy

***Abstract*—MinTallySYCL is a conversion project that translates the minTally application from OpenMP to SYCL, aimed at leveraging modern hardware features for enhanced performance. Originally, minTally abstracts key components from a Monte Carlo neutron transport application, particularly focusing on tally scoring regions. This SYCL conversion, based on Amanda Lund's minTally project, seeks to optimize the performance and adaptability of tallying systems in realistic Monte Carlo reactor core problems, supporting extensive tally data structures without the complexities of full application tracking physics**

I. Introduction

Motivation. The primary motivation behind the minTallySYCL project is to enhance the performance and adaptability of Monte Carlo neutron transport simulations by converting the original minTally application from OpenMP to SYCL. This project addresses these limitations through several key improvements:

- **Performance Optimization:** SYCL allows for better parallelization and efficient utilization of both CPU and GPU architectures, significantly reducing execution times.
- **Improved Data Management:** The elimination of the tensor3d structure leads to more efficient data handling, reducing computational overhead and enhancing overall performance.
- **Hardware Adaptability:** SYCL provides a unified programming model capable of targeting various hardware platforms, making the application more versatile and future-proof.
- **Enhanced Computational Efficiency:** By refining tally scoring mechanisms and optimizing data structures, minTallySYCL aims to deliver more accurate and efficient simulations.
- **Unified Shared Memory (USM) Utilization:** The use of USM in SYCL improves memory management and access patterns, further enhancing computational efficiency and simplifying code complexity.

This project seeks to create a robust, high-performance solution for Monte Carlo neutron transport simulations, addressing the shortcomings of the original OpenMP-based implementation and paving the way for future advancements.

Related work. The minTallySYCL project builds on prior implementations of minTally using OpenMP, a widely used API for shared-memory parallelism. Studies such as "The OpenMP Cluster Programming Model" explore extending OpenMP to distributed clusters, improving scalability and performance by enabling efficient task offloading to GPUs and other accelerators [1]. Another relevant work, "Advising OpenMP Parallelization via a Graph-Based Approach with Transformers," introduces OMPify, a tool that automates the insertion of OpenMP pragmas into serial code, achieving high accuracy on standard benchmarks like NAS and PolyBench [2]. The original minTally project on GitHub also provides a solid foundation for understanding the implementation of Monte Carlo neutron transport simulations with OpenMP [3]. These foundations highlight the importance of efficient parallelism and data management, informing the transition to SYCL in minTallySYCL for enhanced performance and adaptability.

II. Background

In this section, we provide essential background information to make the paper self-contained. We outline the problem of Monte Carlo neutron transport simulations and discuss the original minTally project implemented with OpenMP.

Monte Carlo Neutron Transport Simulations. Monte Carlo neutron transport simulations are critical in nuclear engineering for modeling neutron behavior in reactors. These simulations involve tracking the paths of neutrons and their interactions with materials, such as absorption, scattering, and fission events. The results are used to optimize reactor designs, ensure safety, and improve efficiency.

The minTally Project. The original minTally project utilizes OpenMP for parallel processing in tally scoring within Monte Carlo simulations. Tally scoring is the process of recording specific neutron interactions during the simulation, which is essential for analyzing reactor behavior and performance. The project focuses on optimizing the performance of tally scoring regions by leveraging the parallel capabilities of OpenMP.

OpenMP. OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It provides a simple and flexible interface for developing parallel applications on various architectures, including multi-core processors and GPUs. OpenMP uses a set of compiler directives, library routines, and environment variables to specify parallel regions and control the execution of parallel code.

SYCL. (pronounced ”sickle”) is a higher-level programming model for heterogeneous computing built on OpenCL. It enables developers to write code that can run on different types of processors, such as CPUs, GPUs, and FPGAs, using a single source. SYCL simplifies the development of parallel applications by providing a unified programming model that enhances portability and efficiency.

Unified Shared Memory (USM). USM is a feature of SYCL that enhances memory management by allowing pointers to be shared across host and device code. This simplifies data movement and management, leading to more efficient memory access patterns and reduced code complexity. USM is particularly beneficial for applications with complex memory interactions, such as Monte Carlo simulations.

Reduction. Reduction is an essential operation in parallel computing, where a collection of values is combined to produce a single result using operations such as sum, maximum, minimum, or logical AND/OR. Reduction operations are ubiquitous in scientific computing, data analysis, and machine learning due to their efficiency in summarizing large data sets.

SYCL, on the other hand, provides more granular control over reduction operations, allowing developers to leverage the capabilities of heterogeneous computing platforms. SYCL supports reductions using built-in functions that can operate across different types of devices, optimizing the reduction process for the target architecture. This includes leveraging specialized hardware features available on GPUs and FPGAs to accelerate the reduction process.

Effective use of reduction operations can significantly enhance the performance and scalability of parallel applications. By minimizing synchronization overhead and efficiently combining partial results from multiple threads or compute units, reduction operations ensure that applications can scale across multiple processors and diverse architectures, maintaining high performance and efficient resource utilization.

III. Method of conversione from OpenMP to SYCL

This section outlines the approach for converting the original minTally project from OpenMP to SYCL, enhancing performance and simulation capabilities. The key steps include code conversion from C to C++, adopting SYCL for simulation, and optimizing data types for GPU compatibility.

1) **Analyzing the OpenMP Implementation.** The initial step involves analyzing the original minTally project, implemented in C with OpenMP, to understand its structure, parallel regions, and data handling mechanisms. This analysis identifies areas for optimization and necessary changes for conversion to SYCL.
2) **Converting Code from C to C++.** The project code is converted from C to C++ to leverage the object-oriented features and compatibility of C++ with SYCL. This conversion includes refactoring the code to use C++ constructs and preparing it for integration with SYCL.
3) **Utilizing SYCL for Simulation.** The core of the method involves translating OpenMP parallel regions into SYCL kernels, enabling the code to run efficiently on various hardware platforms. This step includes:
   - Implementing SYCL buffers and accessors to manage data transfer between host and device.
   - Designing kernels that execute the tally scoring mechanisms in parallel on different device types.
4) **Optimizing Data Types for GPU Compatibility.** To ensure optimal performance on GPUs, data types are converted from `double` to `float`. This change is critical since many GPUs do not support double-precision operation
5) **Integrating Unified Shared Memory (USM).** USM is used to simplify memory management by allowing pointers to be shared between host and device code. This reduces the complexity of data movement and improves access patterns, further enhancing performance.
6) **Optimizing Data Structures.** Replacement of 3D tensors with arrays to emulate their behavior for improved data management. This approach significantly enhances data locality and memory access patterns, which are critical for efficient GPU execution. By using arrays instead of 3D tensors, we can reduce the overhead associated with complex indexing and improve cache utilization. This leads to better performance and scalability on GPU architectures, enabling more efficient parallel computation and faster data processing.

```
typedef struct {
  int n_scores;
  int n_filter_bins;
  int *filter_bins;
  double ***results; //3D Tensor
} Tally;
```

Fig. 1. Implementation of tallies structure in OpenMP

```
typedef struct {
  int n_scores;
  int n_filter_bins;
  int *filter_bins;
  float *results;
} Tally;
```

Fig. 2. Implementation of tallies structure in SYCL

This method leverages SYCL's capabilities to create a robust, high-performance solution for Monte Carlo neutron transport simulations, addressing the limitations of the original OpenMP implementation.

IV. Experimental Results

Experimental setup.

- **Processor**: Intel Core i7-1165G7
  - **Base Frequency**: 2.8 GHz

  - **Max Turbo Frequency**: 4.7 GHz
  - **Cache**: 12 MB L3
- **Graphics**: Intel Iris Xe Graphics
  - **Frequency**: Up to 1.3 GHz
  - **Unified Memory**: Can use up to half of the total system memory, e.g., up to 8 GB if the system has 16 GB of RAM.
- **Memory**: 16 GB DDR4-3200 MHz RAM
- **Operating System**: Windows 11 Pro

The source codes are compiled with the version 2024.2.0 of the Intel oneAPI DPC++/C++ compiler, using the `-O3` optimization flag to obtain better performance.

To verify that the code is working correctly, a checksum has been used that does not change when the number of work items varies. In this way it is guaranteed that the code is working correctly even when it scales in terms of work items.

All tests were performed ten times and only the median of these values was taken into consideration

Performance Comparison. In these experiments, our aim is to evaluate the performance impact of our SYCL implementation by comparing it against the sequential and OpenMP implementations. Each implementation is tested using identical parameters to ensure a fair comparison. Specifically, we utilize the maximum number of threads for the OpenMP implementation[3] and the maximum number of work items for the SYCL implementation.

Both implementations are initialized with the following parameters:

- **Seed**: 12345
- **Number of Particles**: 100000
- **Number of Tallies**: 180
- **Number of Filter Bins**: 50952
- **Number of Scores**: 6
- **Number of Nuclides**: 6
- **Number of Coordinates**: 10
- **Number of Events**: 5.4
- **Number of Work Items / Threads**: The maximum number possible. In this context 512 work items for the SYCL version and 8 threads for the OpenMP version.

Fig. 3. Comparison of the execution time of the Mintally program between sequential,OpenMP and Sycl version.

- `Sequential implementation`: 5.11645 seconds
- `OpenMp implementation`: 2.90872 seconds
- `Sycl implementation`: 3.87568 seconds

In Figure 3, it is possible to see the comparison of execution times between the Sequential, OpenMp, and Sycl implementations. The results show that the Sequential implementation takes 5.11645 seconds, the OpenMp implementation takes 2.90872 seconds, and the Sycl implementation takes 3.87568 seconds.

The figure clearly highlights that the OpenMp implementation is the fastest, followed by the Sycl implementation, with the Sequential implementation being the slowest. The fact that the Sycl implementation is not the fastest can be attributed to several factors. A key factor is the use of an integrated GPU, which shares memory resources with the CPU. This can cause contention for memory access, slowing down overall performance. Additionally, cache efficiency may be lower on an integrated GPU compared to a dedicated GPU, negatively impacting execution times.

Performance Scalability with Increasing Work Items (Strong scalability). To evaluate performance scalability, tests were conducted starting with a single work item and incrementally increasing the number of work items up to 3000, in steps of 100. This approach allows for a detailed analysis of how performance changes as the workload grows, providing insights into the efficiency and scalability of the implementation under varying conditions.

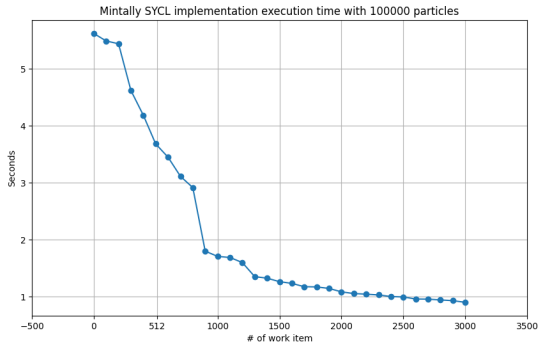

Fig. 4. Execution time of the Mintally SYCL implementation with 100,000 particles as a function of the number of work items on an integrated GPU.

The graph in Figure 4 demonstrates good strong scalability. As the number of work items (X) increases, the simulation times decrease significantly, indicating effective use of additional resources. For the first 300 work items, the simulation times decrease slightly, remaining around 5 seconds, likely due to the initial overhead of starting the GPU. From 400 to 900 work items, the simulation times drop drastically, showing more efficient GPU utilization. From 900 to 3000 work items, the reduction in simulation times continues but slows down, suggesting a point of saturation for the integrated GPU's resources.

Although the integrated GPU supports a maximum of 512 work items, the simulation times improve with more work items. This could be due to efficient memory management and parallelism by the system. The use of Unified Shared Memory

(USM) can help reduce data transfer times between the CPU and GPU, further enhancing performance.

Performance Scalability with Increasing number of particles (Weak scalability). In this section, we analyze the performance scalability with an increasing number of particles (weak scalability). Specifically, we evaluate how the system handles a growing workload while maintaining a constant number of work items. Our analysis ranges from a minimum of 1 particle to a maximum of 150,000 particles, in steps of 1000, using the same number of work items for each execution. The number of work items remains constant at 512. This approach allows us to understand the system's efficiency and resource utilization under more demanding conditions, highlighting implications for performance optimization and hardware adaptability in realistic simulation scenarios.

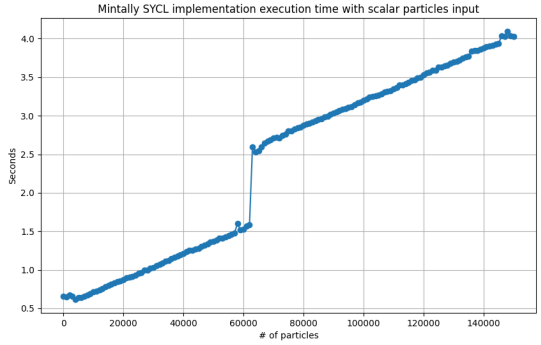

Fig. 5. Execution time of the Mintally SYCL implementation with increasing numbers of scalar particles input.

The graph in Figure 5 shows the execution time of the Mintally SYCL implementation with an increasing number of particles, highlighting weak scalability. As the number of particles increases from 1 to 150,000, the execution time rises, which is expected with a larger workload.

Up to approximately 60,000 particles, the execution time increases linearly and gradually. However, around this point, there is a sudden jump in execution time, which then continues to rise linearly after this spike.

This jump can be attributed to several factors. One possible cause is the contention for memory resources between the CPU and the integrated GPU, as they share the same memory. Additionally, cache efficiency may be negatively impacted when the number of particles exceeds a certain threshold, requiring more complex memory management.

Performance comparison with different compiler optimization.

In this section, we compared the performance of the Mintally SYCL implementation using different compiler optimizations. The objective was to evaluate how various levels of optimization affected the execution time and overall efficiency of the program. Compiler optimizations can significantly impact performance by improving code efficiency.

The implementation is initialized with the following parameters:

- **Seed**: 12345
- **Number of Particles**: 1000
- **Number of Tallies**: 180
- **Number of Filter Bins**: 50952
- **Number of Scores**: 6
- **Number of Nuclides**: 6
- **Number of Coordinates**: 10
- **Number of Events**: 5.4
- **Number of Work Items**:512 work items

The C++ compiler optimization levels considered were:

- `-O0`
- `-O1`
- `-O2`
- `-O3`
- `-Ofast`
- `-Og`

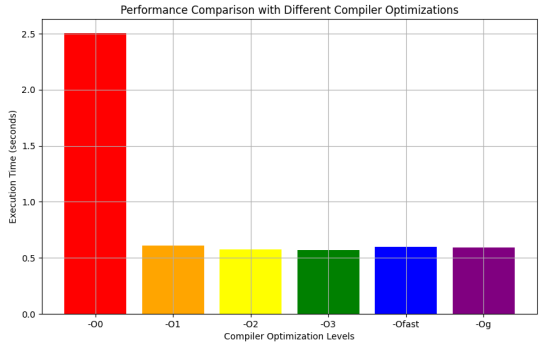

Fig. 6. Performance comparison of the Mintally SYCL implementation using different compiler optimization levels.

The results (showed in figure 6) from the comparison of the Mintally SYCL implementation using various compiler optimization levels are as follows:

- `-O0`: 2.50688 seconds
- `-O1`: 0.610229 seconds
- `-O2`: 0.576811 seconds
- `-O3`: 0.568585 seconds
- `-Ofast`: 0.599856 seconds
- `-Og`: 0.59029 seconds

Without optimization (`-O0`), the execution time is significantly higher at 2.50688 seconds. This is because the code is compiled as quickly as possible without any transformations to improve efficiency. With light optimization (`-O1`), the execution time drops drastically to 0.610229 seconds, demonstrating that even simple optimizations can have a significant impact on performance.

Standard optimization (`-O2`) further reduces the execution time to 0.576811 seconds by applying a comprehensive set of optimizations that improve code efficiency. With aggressive optimization (`-O3`), the execution time is even lower at 0.568585 seconds. This level includes all optimizations from `-O2` plus additional transformations that further enhance performance.

Interestingly, with `-Ofast`, which should include the most aggressive optimizations, the execution time is slightly higher than `-O3`, at 0.599856 seconds. This may be due to trade-offs between precision and speed that do not always lead to performance improvements in every situation.

Finally, the `-Og` level, which optimizes for debugging, shows an execution time of 0.59029 seconds. This level balances performance with the need to maintain good debugging information.

In summary, the results indicate that compiler optimizations can drastically reduce execution times.

V. Conclusion

The minTallySYCL project demonstrates the potential and challenges of converting an OpenMP-based application to SYCL to leverage modern hardware capabilities and improve the performance of Monte Carlo neutron transport simulations. The following key observations and results emerged from our experiments:

Overall Performance. Although the SYCL conversion showed no performance improvements over the OpenMP implementation, albeit not uniformly. Performance was significantly affected by the hardware used, with the integrated GPU showing limitations compared to dedicated GPUs. The execution time of the SYCL implementation was 3.87568 seconds compared to 2.90872 seconds of the OpenMP implementation (As shown in Figure 3). This highlights the importance of adequate hardware to fully exploit the benefits of heterogeneous programming with SYCL.

Strong and Weak Scalability. Scalability tests yielded promising results. Strong scalability demonstrated that increasing the number of work items significantly reduced simulation times up to a saturation point, beyond which further increases had a lesser effect. Weak scalability, on the other hand, showed an increase in execution time with an increasing number of particles, with significant spikes between 60,000 and 80,000 particles.

Compiler Optimization. Analyzing the effect of compiler optimizations revealed that using more aggressive optimization levels can drastically reduce execution times. Optimizations `-O3` and `-O2` produced the best results, reducing execution time to around 0.57 seconds. It is notable that `-Ofast` optimization, while designed to maximize performance, did not always provide significant improvements over `-O3`, likely due to trade-offs between precision and speed.

VI. Further comments

The minTallySYCL project highlighted the advantages of SYCL in improving memory management and computational efficiency through the use of Unified Shared Memory (USM) and data type conversion for GPU compatibility. However, the integration and efficient use of hardware resources remain crucial challenges, particularly when using integrated GPUs.

Future optimizations could focus on:

- **Resource Management Improvement**: Optimizing resource management to reduce execution time spikes.
- **Dedicated Hardware Utilization**: Evaluating performance on dedicated GPUs to fully leverage SYCL's advantages.
- **Memory Model Optimization**: Refining memory management to minimize transfer times and improve data access efficiency.
- **Expanded Hardware Support**: Extending support to diverse hardware platforms to increase application versatility and portability.

In general, it would be ideal to perform further tests on dedicated hardware or on a discrete GPU rather than using an integrated one

References

[1] Huber, N., & Mayer, P. (2022). The OpenMP Cluster Programming Model. arXiv preprint arXiv:2207.05677.

[2] Kadosh, T., Schneider, N., Hasabnis, N., Mattson, T., & Pinter, Y. (2023). Advising OpenMP Parallelization via a Graph-Based Approach with Transformers. arXiv preprint arXiv:2305.07899.

[3] Lund, A. (2020). minTally. GitHub repository. Available at: https://github.com/amandalund/minTally