



# Parallelism Made Easy: HIPSTDPAR

HLRS Training

May 7th 2025

Presenter: Alessandro Fanfarillo

# Agenda

1. Introduction to stdpar in C++17/20, how to compile HIPSTDPAR code, and restrictions
2. How to reason, example of porting application from serial, to CPU parallel, to GPU parallel
3. Performance results
4. Mix and Match
5. Surprise!

# What is C++ Standard Parallelism?

With the C++ 17 standard, support for parallelism was introduced. The application developer specifies parallelism as the first parameter to a C++ algorithm:

- `std::execution::seq` – Sequential execution

All operations on the thread that invoked the algorithm

- `std::execution::unseq` – Vectorized execution (C++20)

Indicate that a parallel algorithm's execution may be vectorized (e.g., executed on a single thread using instructions that operate on multiple data items)

- `std::execution::par` – Parallel multithreaded execution

Parallel execution allowed. Operations are indeterminately sequenced within a thread

- `std::execution::par_unseq` – Parallel multithreaded and vectorized execution

The various operations can be interleaved with each other on the same thread. Any given operation may start on a thread and end on a different thread

This means that user code should not do any memory allocation / deallocation, only relies on lock-free specializations of `std::atomic`, and does not rely on synchronization primitives such as `std::mutex`

# Bringing C++ Standard Parallelism to AMD GPUs

- With the release of ROCm™ 6.1, C++ standard parallelism is available for AMD GPUs
- Blog post on this topic: <https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-hipstdpar-readme/>
- To enable, use the `--hipstdpar` compile flag, and `--hipstdpar-path=/rocm/include/thrust/system/hip/hipstdpar`
- This release only supports the `par_unseq` execution policy
- Offloading C++ Standard Parallel algorithm execution to GPU relies on the interaction between the LLVM compiler, HIPSTDPAR, and rocThrust
- By default, HIPSTDPAR assumes that the underlying system is HMM-enabled (HMM Mode, export `HSA_XNACK=1` required)
- On systems without HMM, HIPSTDPAR requires an extra compilation flag: `--hipstdpar-interpose-alloc`
- This flag instructs the compiler to replace all dynamic memory allocations with compatible `hipManagedMemory` allocations (Interposition mode)

# C++ Standard Algorithms to Parallel GPU Execution

The following serial code:

```
transform(x.begin(), x.end(), x.begin(), [](double elem_x) {  
    return 5.0*elem_x;  
} );
```

Runs in parallel on CPU and requires Thread Building Blocks (TBB) library:

```
transform(std::execution::par,  
    x.begin(), x.end(), x.begin(), [](double elem_x) {  
    return 5.0*elem_x;  
} );
```

Runs in parallel on GPU when --hipstdpar is passed at compile time and converted to:

```
transform(std::execution::par_unseq,  
    x.begin(), x.end(), x.begin(), [](double elem_x) {  
    return 5.0*elem_x;  
} );
```

# Restrictions

- Pointers to function, and all associated features (e.g., dynamic polymorphism), cannot be used (directly or transitively) by the user provided callable function
- Global / namespace scope / static / thread storage duration variables cannot be used (directly or transitively) by the user provided callable function
- Only algorithms that are invoked with iterator arguments that model `random_access_iterator` are candidates for offload
- Exceptions cannot be used by the user provided callable function
- Dynamic memory allocation (e.g., `operator new`) cannot be used by the user provided callable function
- Selective offload is not possible i.e., it is not possible to indicate that only some algorithms invoked with the `parallel_unsequenced_policy` are to be executed on the accelerator

# Restrictions for Interposition Mode

All previous restrictions apply to Interposition Mode. In addition, the following also apply:

1. All code that is expected to interoperate must be recompiled with the `--hipstdpar-interpose-alloc` flag (i.e., it is not safe to compose libraries that have been independently compiled)
2. Automatic storage duration (i.e., stack allocated) variables cannot be used (directly or transitively) by the user provided callable

```
bool never(const vector& v, int n) {  
    return any_of(execution::par_unseq, cbegin(v), cend(v),  
        [p = &n](auto&& x) { return x == *p; });  
}
```

# How to reason about hipstdpar parallelism

- Parallelism can provide substantial speedup to serial apps. Important to choose the right kind of parallelism, policy, and device
- Prioritize Data Parallelism over Task Parallelism
- Use Standard Algorithms whenever possible: the beauty of stdpar is that it works with existing C++ Standard Library algorithms, making parallelization effortless
- Instead of writing explicit loops, use `std::for_each`, `std::transform`, `std::reduce`, etc. to allow the compiler to optimize execution automatically
- When using `par` or `par_unseq`, operations must not have dependencies between elements:
  - Do not modify shared variables inside parallelized loops
  - Use reductions instead of accumulating results manually

The following is a bad idea:

```
double sum = 0;
std::for_each(std::execution::par,
              data.begin(), data.end(), [&](double x) {
    sum += x; // Race condition! Multiple threads modifying 'sum' at the same time
});
```

# Minimax problem

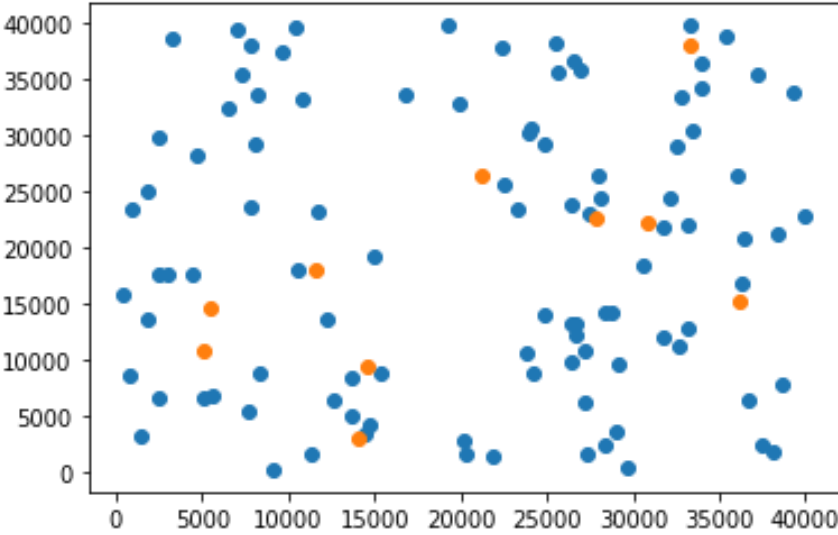
Space-filling point selection in a 2D space (e.g., sensor placement): minimize the maximum distance from any point in the space to the nearest placed point. Greedy algorithm good option

S: subset of points (chosen samples, like 10)

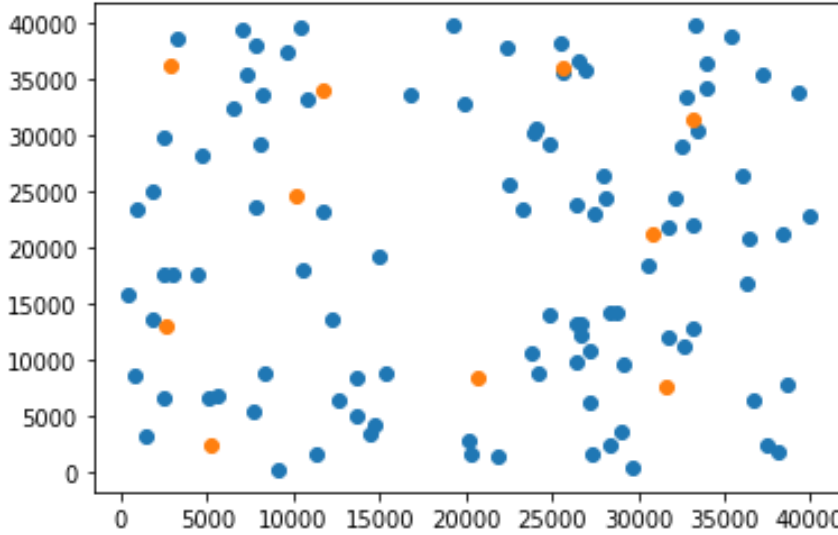
X: entire set of points ([1-40000])

Goal: minimize  $d_{\max}(X, S)$  where  $d_{\max}$  is the maximum distance between a point in X and the closest point in S

Random



Space-filling



# Performance results for Minimax

Node equipped with 4 APUs (192 threads). Single APU execution. ROCm-6.1.3

Numactl to select number of threads,24 in this case. (e.g., numactl -C 0-23 ./cpu\_minimax)

Code version	Time 4000 elements	Time 10000 elements	Time 40000 elements
Original	1 minute	Too long	Too long
Seq	42 seconds	4 m 22 s	Too long
Unseq	42 seconds	4 m 22 s	Too long
Par (192 threads)	2 m 5 s	Too long	Too long
Par (24 threads)	56 seconds	57 seconds	6 m 51 s
Par_unseq	22 seconds	1 minute	4 m 48 s
Par_unseq + affinity	20 seconds	55 seconds	4 m 4 s

# Performance results for TSP

- Travelling salesman problem (TSP): “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?”
- Solved via brute-force by testing all possible permutations of cities
- NP-Hard problem with exponential complexity. Extra city corresponds to exponential increase in the search space
- CPU version using all cores available: 48 logical on MI300A

```
afanfari@sh5-1e707-rd04-03:~/tsp/stdpar$ ./tsp_clang_stdpar_cpu 14
Trav Salesman Prob N=14, best route cost is: 2650, average time is 72.700000 seconds
Solution route is Permutation #41165779714 || 11 8 7 6 0 12 13 10 9 1 3 2 5 4
afanfari@sh5-1e707-rd04-03:~/tsp/stdpar$ ./tsp_clang_stdpar_gpu 14
Trav Salesman Prob N=14, best route cost is: 2650, average time is 4.592000 seconds
Solution route is Permutation #41165779714 || 11 8 7 6 0 12 13 10 9 1 3 2 5 4
```

# Mix and Match: OpenMP and STDPAR on GPU

It is not possible to mix OpenMP® and STDPAR sections in the same file

It is possible to have two separate files, one using OpenMP and the other using stdpar, and invoke them from the same function

In Computer Science, you can always add a layer of abstraction to make things happen...

```
{
    std::vector<double> copy = data;
    std::unique_ptr<ParallelExecutor> executor = std::make_unique<OpenMPExecutor>();
    executor->compute(copy);
}
{
    std::vector<double> copy = data;
    std::unique_ptr<ParallelExecutor> executor = std::make_unique<StdParExecutor>();
    executor->compute(copy);
}
```

OpenMP CPU: `hipcc -fopenmp openmp_executor.cpp -c`

StdPar GPU: `hipcc -std=c++20 --hipstdpar stdpar_executor.cpp --offload-arch=gfx942 -c`

Final: `hipcc -std=c++20 -fopenmp stdpar_executor.o openmp_executor.o main.cpp -o final`

Number of CPU threads controlled via `OMP_NUM_THREADS`

# Mix and Match: STDPAR on CPU and GPU

It is not possible to mix `par` and `par_unseq` sections in the same file

It is possible to have two separate files, with a specific implementation in each of them... Same as before

```
{
    std::vector<double> copy = data;
    std::unique_ptr<ParallelExecutor> executor = std::make_unique<StdParCPUExecutor>();
    executor->compute(copy);
}
{
    std::vector<double> copy = data;
    std::unique_ptr<ParallelExecutor> executor = std::make_unique<StdParGPUExecutor>();
    executor->compute(copy);
}
```

Std\_cpu: `hipcc -O3 -std=c++20 stdpar_cpu_executor.cpp -c`

Std\_gpu: `hipcc -std=c++20 --hipstdpar stdpar_gpu_executor.cpp --offload-arch=gfx942 -c`

Final: `hipcc -O3 -std=c++20 stdpar_gpu_executor.o stdpar_cpu_executor.o main.cpp -o final -ltbb`

Number of CPU threads controlled via `numactl -C`

# Mix and Match: STDPAR on GPU and HIP + Optimizations

- Mixing stdpar with par\_unseq policy and HIP code within the lambda function is not supported, it may or may not work
- Using built-in functions within a stdpar section is supported but not encouraged
- Having separate HIP and stdpar code in the same file works fine
- Under the hood, stdpar is invoking rocThrust/rocPRIM
- It is possible to inspect ISA and register usage stats in the usual ways (e.g., --save-temps) even for stdpar sections

How to “optimize” stdpar code?

- Rely on how rocThrust/rocPRIM selects launch parameters for each architecture: `export ROCPRIM_TARGET_ARCH=942`
- Main advantages are register allocation and loop unrolling

# HIP Data Movement and HIPSTDPAR

The main goal of STDPAR is to mask the complexity of GPU programming, including CPU/GPU data movement

On discrete GPUs, host/device transfers can consume most of the execution time

It is allowed, but not encouraged, to use GPU buffers allocated via hipMalloc inside stdpar sections when run with par\_unseq and manage the data movement explicitly

```
float* d_data = nullptr; hipError_t err = hipMalloc(&d_data, N * sizeof(float));  
err = hipMemcpy(d_data, hostData.data(), N * sizeof(float), hipMemcpyHostToDevice);  
std::for_each(std::execution::par_unseq, d_data, d_data + N, [] __device__ (float &x) {x = x * x;});  
err = hipMemcpy(hostData.data(), d_data, N * sizeof(float), hipMemcpyDeviceToHost);
```

# Atomics on AMD GPUs via HIPSTDPAR

Atomic operations are allowed within a stdpar section only if implemented in a lock-free way (e.g., no mutex)

```
void StdParExecutor::compute(std::vector<double>& data) {
    std::atomic<double> sum{0.0};
    // Note: std::atomic<double> may not be lock-free on all platforms.

    std::for_each(std::execution::par_unseq, data.begin(), data.end(),
        [&](double x) {
            for (int j = 0; j < 100; ++j) {
                x = std::sin(x) + std::cos(x) + std::sqrt(std::fabs(x) + 1.0);
            }
            sum.fetch_add(x, std::memory_order_relaxed);
            //__atomic_fetch_add(&sum, x, __ATOMIC_SEQ_CST);
        });
    std::cout << "StdPar Accumulated Sum: " << sum.load() << std::endl;
}
```

# Surprise!

The code modifications for the minimax code and all codes samples have been generated by a reasoning LLM (DeepSeek-R1)!

*“Given the following code, parallelize the loop that provides the highest performance boost using stdpar as defined in the standard C++20. To find the changes that provide the highest boost make sure to consider hardware aspects like caching and memory access patterns. Make sure the code works:...”*

Advantages of standard parallelism for AI generated code:

- **Standardization and Portability:** stdpar is part of the C++17 standard, ensuring that code adheres to a consistent specification across different platforms and compilers. This standardization means AI-generated code using stdpar is more likely to be portable and compatible with various systems without requiring modifications.
- **Simplified Syntax:** stdpar integrates seamlessly into standard C++ code, allowing parallelism to be expressed using familiar syntax without the need for additional directives or annotations. This simplicity can make it easier for AI models to generate correct and efficient parallel code, as they can leverage the existing structure of standard library algorithms.

# Conclusions

- HIPSTDPAR represents a great alternative to OpenMP® or HIP for porting CPU applications to GPUs
- Perfect fit for data parallelism, not great for task or other parallel paradigms
- HIPSTDPAR assumes unified memory support. Perfect fit for MI300A!
- Object Oriented Programming is good option for mix-and-match stdpar with other parallel approaches or technologies (e.g., OpenMP)
- Calling functions inside stdpar section implemented in a separate compilation unit is allowed but a bug does not currently allow that
- Implementing the function in the header file is a valid alternative
- Host/Device data movement could be problematic, possible to manually transfer data via HIP routines
- Simplified and standard syntax excellent for AI-driven porting

# Disclaimers and Trademark

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like.

Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information.

However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS.' AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY

IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE.

IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD Arrow logo, ROCm, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies

The OpenMP® name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board

LLVM™ is a trademark of LLVM Foundation

© 2025 Advanced Micro Devices, Inc. All rights reserved.

**AMD** 