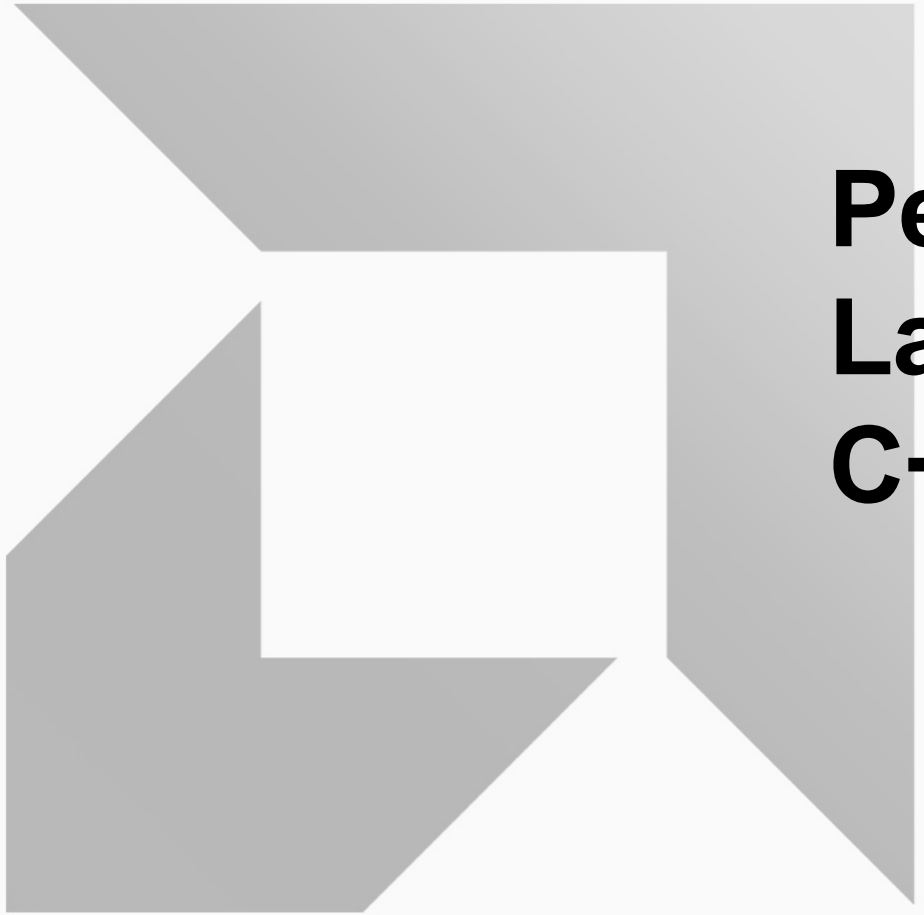


Performance Portable Languages (Kokkos, Raja, C++ Standard Parallelism)

Bob Robey
AMD @ Tsukuba University
Oct 21-23, 2025

AMD
together we advance_

Performance Portability Languages

- Performance, Portability and Productivity (PPP) is critical for today's HPC application developer
  - Custom implementations for each new computer hardware vendor/type is not sustainable
  - Single-source application code is a necessity
- Department of Energy (DOE) has sponsored PPP conferences and workshops on this topic
- Two popular PPP languages have emerged in the DOE
  - Kokkos – SNL C++ performance portable programming model
    - Comprehensive approach to performance portability
    - Parts being integrated into C++ standard
  - RAJA – LLNL C++ performance portability layer
    - Modular in structure with separation of compute and data management
    - Adaptable for how each application team implements in their code

Both RAJA and Kokkos are great choices for C++ HPC applications. They share more similarities than differences. Both are well-supported for AMD GPUs. AMD staff support both RAJA and Kokkos. There are other similar portability frameworks that are also good.

Why Kokkos?

- For C++ applications, Kokkos is an attractive PPP development language

- Kokkos provides a single source capability for C++ codes to run on a variety of parallel CPU and GPU architectures
- Kokkos is well-supported and relatively mature
  - Been around for 10 years
  - Used in many critical applications at Sandia National Laboratories
  - Gaining use in lots of other applications worldwide
  - Selected for Exascale Computing Project funding
- Kokkos generated code performs nearly as well or better than lower-level languages

HIP backend

- Kokkos has long had a HIP backend for selected AMD Processors
  - Both CPUs and GPUs
- The Kokkos team has aggressively developed their implementation for new AMD systems coming online
- In the Fall of 2022, the HIP backend was promoted to production status

- Kokkos handles many of the unique attributes of the AMD GPUs for you

What is Kokkos and How does it work?

- Kokkos (κόκκος) is greek for “grains”, “seed” or “kernels” as in grains of sand or kernels in an ear of corn
- Library based on C++ templates
  - Libraries are quicker to implement and distribute
  - Eventually these techniques can migrate to compilers
    - but this is one-by-one for each compiler
    - additions to language standards takes even longer
    - The concept of multi-dimensional arrays from Kokkos will be implemented as “mdspan” in the C++ 23 standard
- Developed by a team of computer scientists at Sandia National Laboratory
  - Original purpose was to provide an abstraction layer for mathematical solvers
- Supports many backends including OpenMP® threading, CUDA, HIP, and others

Kokkos abstractions for GPUs (and parallelism on CPUs)

- Two basic requirements for a GPU programming language
  - Actually, for any fine-grained parallel language that runs on either GPUs or CPUs

- **Execution capability** – this handles how to generate the execution code within a program to run on the target architecture. Generally, this is for loops, but may also include single lines of computation.
- **Memory handling** – the control of the allocation and movement of memory between the CPU and GPU or other memory locations.

- Kokkos, as a portability layer for various fine-grained programming languages, must have an abstract representation of these two requirements.

Execution and Memory abstractions in Kokkos

Execution Spaces -- compute hardware where computations are done

- Execution Patterns
  - Simple loops -- parallel_for
  - Reductions -- parallel_reduce
  - Scans -- parallel_scan
- Execution Policies
  - Range policies -- basically index sets that need to be operated on
  - Team policies – grouping threads into teams as a subset of the execution space for hierarchical parallelism.

Memory Spaces – memory hardware where the data is stored

- Memory Layout
  - LayoutRight vs LayoutLeft or automatic conversion between the two for different execution spaces
- Memory Traits
  - atomic access, random access (shader memory), streaming stores

Kokkos has two main build options for cmake

External build

- Modify CMakeLists.txt
  - **find_package**(Kokkos)
  - **target_link_libraries**(<my_application> Kokkos::kokkos)
- export Kokkos_DIR=<kokkos_path>/lib/cmake/Kokkos

In-line build

- Retrieve a copy of Kokkos
  - git clone https://github.com/kokkos/kokkos Kokkos, or
  - download a zip or tar file of kokkos from https://github.com/kokkos/kokkos
- or create a submodule
  - git submodule add https://github.com/kokkos/kokkos Kokkos
- Modify CMakeLists.txt
  - **add_subdirectory**(Kokkos)
  - **target_link_libraries**(<my_application> Kokkos::kokkos)

Kokkos Examples with HIP backend

We'll demonstrate how Kokkos works with some examples

- Stream Triad
- Shallow Water

It is recommended that you try these out on your own to learn the most effectively.

Stream Triad Example

Stream Triad application - the steps

We'll work through these one at a time

1. First do an external Kokkos build with OpenMP® backend and a HIP backend
2. Modify CMakeList.txt to add Kokkos headers and library
3. Add Kokkos views for memory allocation of arrays
4. Add Kokkos execution pattern – parallel_fors
5. Add Kokkos timers
6. Run and measure performance for OpenMP
7. Rebuild for AMD Instinct GPUs
8. Run and measure performance for AMD Instinct™ GPU

There is a kokkos module on the system that can be used instead of building an external Kokkos package

Step 1: Build a separate Kokkos package

```
module load amdclang rocm
git clone https://github.com/kokkos/kokkos Kokkos_build
cd Kokkos_build
```

- Build Kokkos with OpenMP® backend

```
mkdir build_kokkos && cd build_kokkos
cmake -DCMAKE_INSTALL_PREFIX=${HOME}/Kokkos_HIP \
    -DKokkos_ENABLE_SERIAL=ON -DKokkos_ENABLE_OPENMP=On -DKokkos_ENABLE_HIP=ON \
    -DKokkos_ARCH_ZEN=ON -DKokkos_ARCH_VEGA90A=ON -DCMAKE_CXX_COMPILER=hipcc \
    ..
make –j 8; make install
cd ..
```

- Set Kokkos_DIR to point to external Kokkos package to use

```
export Kokkos_DIR=${HOME}/Kokkos_HIP
```

Step 2: Modify build

```
git clone –recursive https://github.com/EssentialsOfParallelComputing/Chapter13
Chapter13
cd Chapter13/Kokkos/StreamTriad
cd Orig
```

- Test serial version with

```
mkdir build && cd build; cmake ..; make; ./StreamTriad
```

- If run fails, try reducing the size of the arrays
- Add to `CMakeLists.txt`

```
find_package(Kokkos REQUIRED)
target_link_libraries(StreamTriad Kokkos::kokkos)
```

- Retest with

```
cmake ..; make; ./StreamTriad
```

- Check Ver1 for solution. These modifications have already been made in this version.

Step 3: Add Kokkos views for memory allocation of arrays

Add include file

```
#include <Kokkos_Core.hpp>
```

Add initialize and finalize

```
 Kokkos::initialize(argc, argv);  {
 } Kokkos::finalize();
```

Replace static array declarations with Kokkos views

```
int nsize=80000000;
Kokkos::View<double *> a( "a", nsize);
Kokkos::View<double *> b( "b", nsize);
Kokkos::View<double *> c( "c", nsize);
```

Rebuild and run

Kokkos Syntax: Initialization of Kokkos

- The first requirement for using Kokkos is to include a header file

```
#include <Kokkos_Core.hpp>
```

- The next requirement is to initialize and finalize the Kokkos environment

```
Kokkos::initialize(argc, argv);
Kokkos::finalize();
```

- The initialize call should follow the MPI_Init call, if present, and should be near the start of the program
- You should add scope guards to these calls so that the memory that Kokkos allocates gets deallocated before the finalize call

```
Kokkos::initialize(argc, argv);
{
…
}
Kokkos::finalize();
```

Kokkos Syntax: Kokkos memory (views)

```
Kokkos::View<double *> x("data label", N0);
```

Data can be accessed with either x[i] or x(i)

Kokkos handles deallocation automatically

By default, Kokkos views are initialized. This can be overridden by adding an optional parameter.

```
Kokkos::View<double *> x (Kokkos::ViewAllocateWithoutInitializing (label), N0);
```

You can also create an unmanaged view of a raw pointer, x_raw

```
Kokkos::View<double*, Kokkos::HostSpace,
   Kokkos::MemoryTraits<Kokkos::Unmanaged> > x_view (x_raw, N0);
```

Step 4: Add Kokkos execution pattern – parallel_for

Change for loops to Kokkos parallel fors.

- At start of loop

```
Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
```

- At end of loop, replace closing brace with

```
});
```

Rebuild and run. How much speedup do you observe?

Step 5: Add Kokkos timers

Add Kokkos calls

```
Kokkos::Timer timer;
timer.reset(); // for timer start
time_sum += timer.seconds();
```

Remove

```
#include <timer.h>
struct timespec tstart;
cpu_timer_start(&tstart);
time_sum += cpu_timer_stop(tstart);
```

Completed version of Kokkos StreamTriad

```
#include <Kokkos_Core.hpp>

int main(int argc, char *argv[]){
   Kokkos::Timer timer;
   int nsize=80000000; int ntimes=16;
   double scalar = 3.0, time_sum = 0.0;

   Kokkos::initialize(argc, argv); {

   // initializing arrays
   Kokkos::View<double *> a( "a", nsize);
   Kokkos::View<double *> b( "b", nsize);
   Kokkos::View<double *> c( "c", nsize);

   Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
      a[i] = 1.0;
      b[i] = 2.0;
   });

   for (int k=0; k<ntimes; k++){
      timer.reset();
      // stream triad loop
      Kokkos::parallel_for(nsize, KOKKOS_LAMBDA (int i) {
         c[i] = a[i] + scalar*b[i];
      });
      time_sum += timer.seconds();
   }

   printf("Average runtime is %lf msecs\n", time_sum/ntimes*1000.0);

   }  Kokkos::finalize();
}
```

Kokkos: performance profiling

Build kokkos tools

```
git clone https://github.com/kokkos/kokkos-tools kokkos-tools
cd kokkos-tools/src/tools/simple-kernel-timer
make
```

Run application with tool

```
./StreamTriad --kokkos-tools-library=<path to kokkos tools>/
    src/tools/simple-kernel-timer/kp_kernel_timer.so
```

or

```
KOKKOS_PROFILE_LIBRARY=<path to kokkos tools>/
   –   src/tools/simple-kernel-timer/kp_kernel_timer.so ./StreamTriad
```

Print out results of tool

```
<path_to_tool_directory>/kp_reader
```

Review

We covered:

- How to use an external Kokkos build (pre-built)
- How to add the Kokkos dependency to a cmake build
- How to initialize and finalize Kokkos in your application
- How to convert arrays to Kokkos views
- How to express simple loops in Kokkos parallel_for syntax

Parallelism Made Easy: HIPSTDPAR

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!

C++ Standard Algorithms

- C++ STD Library contains a massive amount of utility subroutines (algorithms)
- The Algorithms header contains various methods, such as: sort, copy_backwards, for_each, transform…
- Lambda expressions are used to define a method that can be applied to each element of the container

```
vector<double> x(1024, 1);

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

What is C++ Standard Parallelism?

- The C++ 17 standard introduced support for parallelism with parallel policies. 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

For the par_unseq policy, this means that user code does not do any memory allocation / deallocation. It 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 instruct the compiler to replace all dynamic memory allocations with compatible hipManagedMemory allocations (Interposition mode)

* HMM is Heterogeneous Memory Management also called Memory Management

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 (requires TBB library – Threading Building Blocks):

```
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;
} );
```

How to reason

- 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. This allows the compiler to optimize execution automatically
- When using par or par_unseq, operations must not have dependencies between elements:
  - Avoid modifying 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
});
```

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 when the parallel policy is added as the first argument:

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

This code will be offloaded to AMD GPUs when --hipstdpar is passed at compile time

Restrictions

1. Pointers to functions, and all associated features, e.g. dynamic polymorphism, cannot be used (directly or transitively) by the user provided callable
2. Global / namespace scope / static / thread storage duration variables cannot be used (directly or transitively) by the user provided callable
3. Only algorithms that are invoked with iterator arguments that model random_access_iterator are candidates for offload
4. Exceptions cannot be used by the user provided callable
5. Dynamic memory allocation (e.g. operator new) cannot be used by the user provided callable
6. 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 has to 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<int>& v, int n) {
  return any_of(execution::par_unseq, cbegin(v), cend(v),
                [p = &n](auto&& x) { return x == *p; });
}
```

Full list of supported C++ algorithms

adjacent_difference	find	min_element	replace_if	uninitialized_copy
adjacent_find	find_if	minmax_element	reverse	uninitialized_copy_n
all_of	find_if_not	mismatch	reverse_copy	uninitialized_default_construct
any_of	for_each	move	set_difference	uninitialized_default_construct_n
copy	for_each_n	none_of	set_intersection	uninitialized_fill
copy_if	generate	partition	set_symmetric_difference	uninitialized_fill_n
copy_n	generate_n	partition_copy	set_union	uninitialized_move
count	includes	reduce	sort	uninitialized_move_n
count_if	inclusive_scan	remove	stable_partition	uninitialized_value_construct
destroy	is_partitioned	remove_copy	stable_sort	uninitialized_value_construct_n
destroy_n	is_sorted	remove_copy_if	swap_ranges	unique
equal	is_sorted_until	remove_if	transform	unique_copy
exclusive_scan	lexicographical_compare	replace	transform_exclusive_scan	
fill	max_element	replace_copy	transform_inclusive_scan	
fill_n	merge	replace_copy_if	transform_reduce	

Examples

- The first examples are from https://github.com/AMD/HPCTrainingExamples in the HIPStdPar/CXX directory
- Checkout examples

```
git clone https://github.com/AMD/HPCTrainingExamples
cd HPCTrainingExamples/HIPStdPar/CXX
```

- Run each of the examples

```
cd saxpy_foreach
make
./saxpy

cd ../saxpy_transform
make
./saxpy

cd ../saxpy_transform_reduce
make
./saxpy
```

Example with for_each algorithm

```
#include <vector>
#include <algorithm>
#include <execution>

using namespace std;

int main(int argc, char *argv[])
{
   vector<double> x(1024, 1);

   for_each(
      execution::par_unseq, x.begin(), x.end(), [](double& x) {
         x *= 5.0;
      }
   );

   printf("Finished Run\n");
}
```

Makefile

```
EXEC = saxpy
default: ${EXEC}
all: ${EXEC}

ROCM_GPU ?= $(strip $(shell rocminfo |grep -m 1 -E gfx[^0]{1} | sed -e 's/ *Name: *//'))

CXX1=$(notdir $(CXX))

# hipstdpar-path is not needed for ROCm 6.1 and later
ifeq ($(findstring amdclang++,$(CXX1)), amdclang++)
  STDPAR_FLAGS = --hipstdpar --offload-arch=$(ROCM_GPU) --hipstdpar-path=${STDPAR_PATH}
else ifeq ($(findstring clang++,$(CXX1)), clang++)
  STDPAR_FLAGS = --hipstdpar --offload-arch=$(ROCM_GPU) --hipstdpar-path=${STDPAR_PATH}
endif
# Add --hipstdpar-interpose-alloc if HSA_XNACK is not set
ifeq ($(findstring gfx1030,$(ROCM_GPU)),gfx1030)
  STDPAR_FLAGS += --hipstdpar-interpose-alloc
endif

CXXFLAGS = -g -O3 -fstrict-aliasing ${STDPAR_FLAGS}
LDFLAGS = -fno-lto -lm

${EXEC}: ${EXEC}.o
        ${CXX} ${STDPAR_FLAGS} $(LDFLAGS) $^ -o $@

# Cleanup
clean:
        rm -f *.o ${EXEC}
```

--hipstdpar-path won't be needed with ROCm 6.1

--hipstdpar-interpose-alloc replaces allocations with hipMallocManaged for systems without Managed Memory. The AMD clang++ compiler does this behind the scenes for you.

Example with transform algorithm

```
#include <vector>
#include <algorithm>
#include <execution>

using namespace std;

int main(int argc, char *argv[])
{
   vector<double> x(1024, 1);

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

   printf("Finished Run\n");
}
```

Example with transform_reduce algorithm

```
#include <vector>
#include <algorithm>
#include <execution>

using namespace std;

int main(int argc, char *argv[])
{
   vector<double> x(1024, 1);

   double result = transform_reduce(
      execution::par_unseq, x.begin(), x.end(), 0.0, plus<>(), [](double x_elem) {
         return 5.0*x_elem;
      }
   );

   printf("Finished Run: Result %lf\n",result);
}
```

Initial value and operator are added to the transform call

Traveling Salesman Problem

- 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?”
- NP-Hard problem with exponential complexity. Extra cities cause exponential increase in the search space.
- Solved via brute-force by testing all possible permutations of cities in parallel
- Only change needed in the code is the policy: from **execution_par to execution_par_unseq**

```
std::transform_reduce(std::execution::par,
                      counting_iterator(0),
                      counting_iterator(factorial(N)),
                      route_cost(),
                      [](route_cost x, route_cost y)
                      { return x.cost < y.cost ? x : y; },
                      [=](int64_t i)
```

```
std::transform_reduce(std::execution::par_unseq,
                      counting_iterator(0),
                      counting_iterator(factorial(N)),
                      route_cost(),
                      [](route_cost x, route_cost y)
                      { return x.cost < y.cost ? x : y; },
                      [=](int64_t i)
```

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
```

Minimax problem

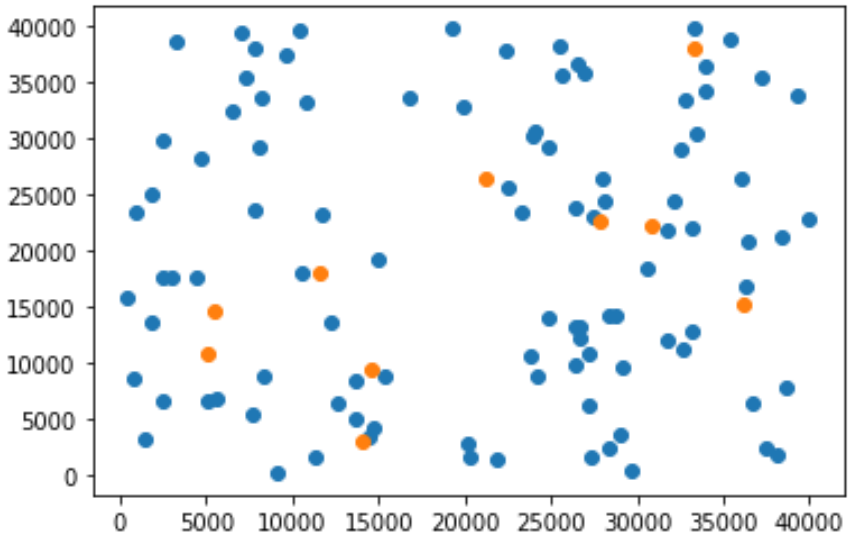
Space-filling point selection in a 2D space (e.g., sensor placement): minimize the maximum distance from any point in the space to the nearest placed point. Greedy algorithm good option

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

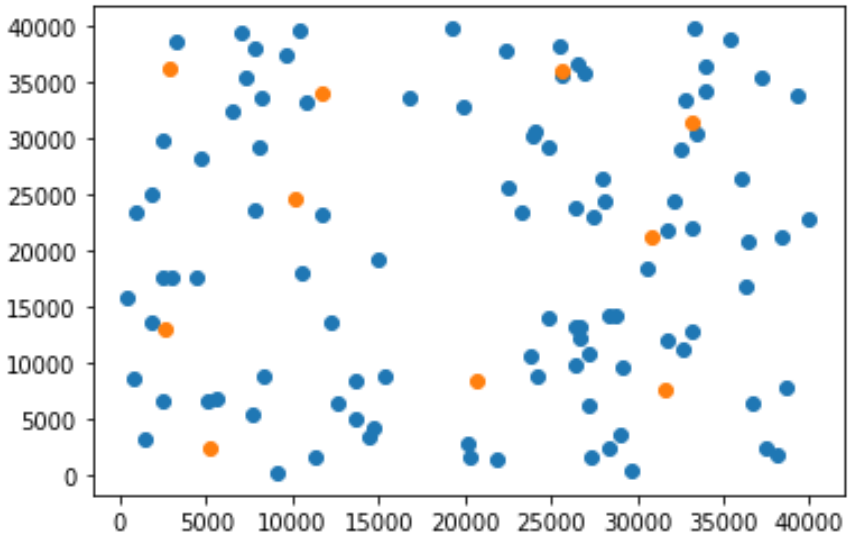
X: entire set of points ([1-40000])

Goal: $\underset{S}{\text{minimize}}\, d\_\max(X, S)$ 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

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
- Works better for walking through a list of particles or cells than 2D grid indexing
- HIPSTDPAR assumes unified memory support. Perfect fit for MI300A!
- 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 routine

Disclaimer

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.

Third-party content is licensed to you directly by the third party that owns the content and is not licensed to you by AMD. ALL LINKED THIRD-PARTY CONTENT IS PROVIDED “AS IS” WITHOUT A WARRANTY OF ANY KIND. USE OF SUCH THIRD-PARTY CONTENT IS DONE AT YOUR SOLE DISCRETION AND UNDER NO CIRCUMSTANCES WILL AMD BE LIABLE TO YOU FOR ANY THIRD-PARTY CONTENT. YOU ASSUME ALL RISK AND ARE SOLELY RESPONSIBLE FOR ANY DAMAGES THAT MAY ARISE FROM YOUR USE OF THIRD-PARTY CONTENT.

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

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries

LLVM is a trademark of LLVM Foundation


AMD