Codes in C++20 Included

Benchmarking Performance of `C++ Algorithms` in

STL, **oneAPI** (±**SYCL**) & **CUDA**

Platforms: CPU, Intel GPU & Nvidia GPU

By: Pouya Ebadollahyvahed

1- What Questions are Going to be Answered Here?

- To sort a very large array or vector which way is the **fastest** way & how much?

  1- Running sorting code on **CPU**

  a. Using C++ standard library (STL)? `std::sort()`

  b. Using Intel's oneDPL library? `oneapi::dpl::sort()`

  2- Running sorting code on **GPU**

  a. Using CUDA + Thrust (on Nvidia GPU)? `thrust::sort()`

  b. Using oneDPL + SYCL (on Intel GPU)? `oneapi::dpl::sort()`

- When size of a vector is growing, how much the sorting time increases? (for each of above cases)

- How much is the performance difference between execution policies?

  - `std::execution::seq`, `std::execution::par` & `std::execution::unseq`

To find out answers, we use 3 C++ compilers: Intel's **DPC++** (icpx), Nvidia's **nvcc** & **MSVC** (Visual C++)

2- C++ Algorithms, Libraries Implemented Them & Platforms They Run On

C++ programmers are usually well familiar with C++ standard algorithms (like for sorting & searching) and commonly use them on daily basis. C++ standard library (STL) contains handful of algorithms which are designed extremely flexible and optimized obsessively. Some categories of STL algorithms are:

- Sorting algorithms (like: `sort`, `partial_sort`, `nth_element`)
- Searching algorithms (like: `binary_search`, `lower_bound`, `equal_range`)
- Merging algorithms (like: `merge`, `inplace_merge`)
- Finding minimum & maximum algorithms (like: `max`, `max_element`, `minmax`)
- Algorithms for partitioning a range of elements into two groups (like: `partition`, `partition_copy`)

Before C++17 such algorithms (functions) were running sequentially or single-threaded which simply means they were always running on 1 CPU core only, so they were not utilizing the entire processing power of multi-core CPUs. Almost 7 years ago and by introduction of C++17 a new optional parameter (argument) was added to algorithm functions which is called **execution policy** which can be one of:

- `std::execution::seq` This is default & classic policy. Algorithm code runs sequentially (single threaded)
- `std::execution::par` Runs code of algorithm function in parallel, utilizing all CPU cores (data parallelism)
- `std::execution::par_unseq` Runs code in parallel (like above) or uses vectorization whichever is faster
- `std::execution::unseq` (Since C++20) Uses vectorization

On pages 3 - 5 of "Mini Handbook of Parallel Computing" (1st reference, the last page), concepts of "*Data Parallelism*", "*Functional Parallelism*" & "*Vectorization*" & their differences are discussed a bit in detail.

Execution policies support data parallelism but still have a limitation: run only on CPUs. This limitation was lifted by introduction of libraries and toolkits like **Nvidia's CUDA** (+Thrust) and **Intel oneAPI** which implemented **versions of algorithms to run on other types of processors like GPUs (in parallel)**.

Here comes a question: which platform runs C++ algorithms faster? CPU or GPU? And which implementation (library) of C++ algorithms is faster? For example, to run a C++ algorithm on CPU we may use standard STL implementation or Intel's oneDPL implementation, so which of them is faster?

In this technical report we want to compare both

1. **Performance** (from one hand) &
2. **Scalability** (from the other hand)

Of different implementations of C++ algorithms.

Since there are 100+ algorithms in STL, we just chose one of them to make the comparison. The one algorithm chosen for the benchmarking is `sort()` function. There are two reasons for this choice:

- `sort()` is one of the most commonly used algorithms
- `sort()` is process-intensive function. Especially when the dataset is large, it does lots of operations

For the benchmarking, following four different scenarios are considered:

Platform		Library / Toolkit	Compiler	Algorithm Function	Exec Policy
CPU	Intel	STL	MSVC, icpx	`std::sort()`	`seq` & `par`
		oneDPL	DPC++ (icpx)	`oneapi::dpl::sort()`	
GPU	Nvidia	CUDA + Thrust	NVCC	`thrust::sort()`	`par`
	Intel	oneDPL + SYCL	DPC++ (icpx)	`oneapi::dpl::sort()`	

Note 1: Section 9 encompasses hardware specification of test platform and software versions of test tools

Note 2: Section 8 encompasses benchmarking source codes in C++, compiling options & methods

Note 3: Since above mentioned implementations of `sort()` function run on different hardware processors (with radically different architecture, frequencies & number of cores), it is **not valid to use absolute measured values (time) to compare performance of libraries, toolkits and compilers with each other** while still it is sane to compare scalability of solutions (= processor + library + compiler) with each other

3- Pre-Requirements

1- **Performance and ABC's of CUDA:**

Check pages 6 - 10 of "Heterogeneous Programming" article (2nd reference, the last page)

2- **oneAPI & oneDPL; what are these Intel Toolkits & Libraries:**

Check pages 41 - 45 of "Mini Handbook of Parallel Computing" (1st reference, the last page)

Check pages 18 - 20 of "Heterogeneous Programming" article (2nd reference, the last page)

3- **Understanding SYCL:**

Check pages 15 - 17 of "Heterogeneous Programming" article (2nd reference, the last page)

4- **Yet Installing CUDA on Windows Machines and Write Codes for Nvidia GPUs:**

Check page 6 of "Heterogeneous Programming" article (2nd reference, the last page)

5- **Application and How to Install oneAPI on Windows Machines and Write SYCL & oneDPL codes:**

Check page 42 of "Mini Handbook of Parallel Computing" (1st reference, the last page)

Note: Because `std::execution::unseq` is being supported since C++20, **all codes are compiled in C++20**

4- Benchmarking Performance of Execution Policies, Data Types & Libraries

Test Case:

- Sorting a `vector<T>` with **50 million elements**
  - Trying different data types for `T`
  - Trying different execution policies (`std::execution`)
  - Trying MSVC and icpx **C++ compilers** for the same code (STL based)
  - Trying `sort()` implementation in **STL** & **oneDPL** libraries

Objectives:

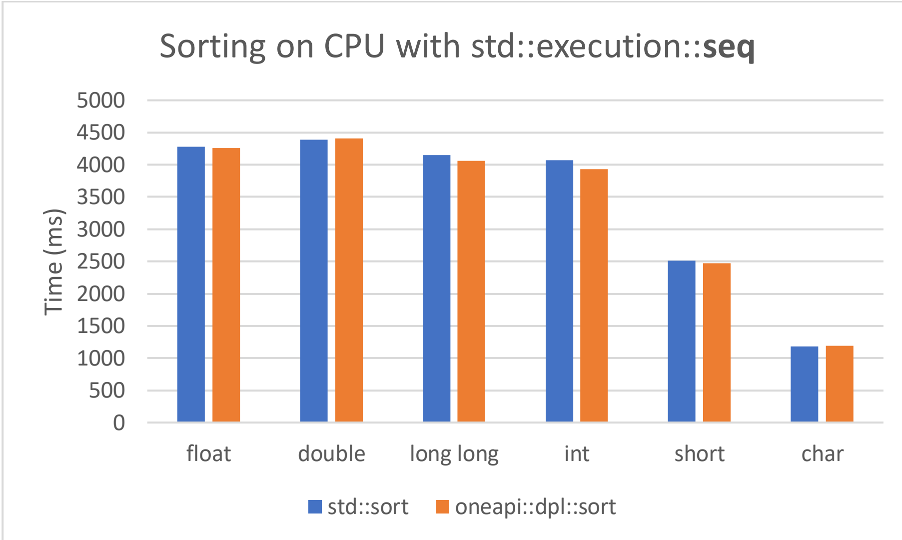
1- Comparing sorting performance of vectors with elements of different **data types**

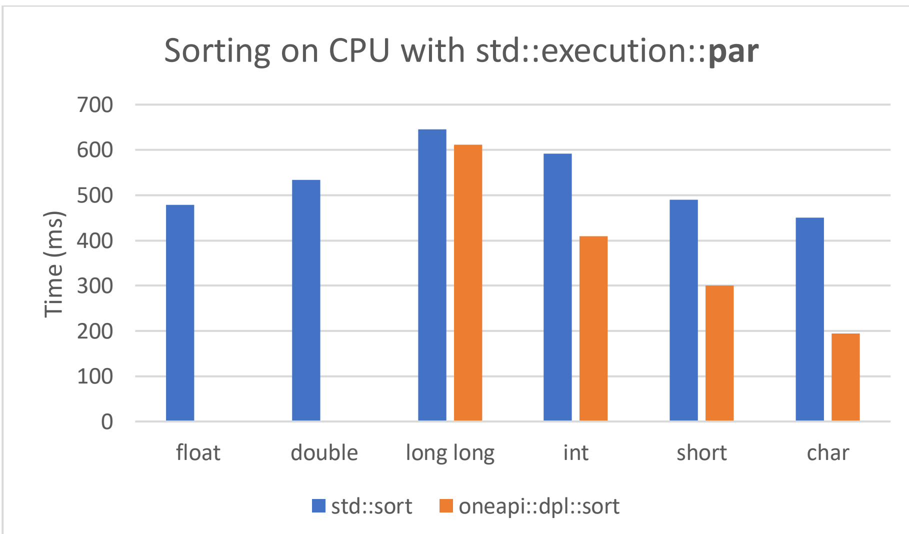
2- Comparing sorting performance of a vector with different **execution policies**

3- Comparing performance of sorting functions in different libraries: **STL** & **oneDPL** (running on **CPU**)

4- Comparing compiling optimization between MSVC and DPC++ (Intel Data Parallel Compiler, icpx)

Data Type		Policy	Measured Time (ms)		
			std::sort()		oneapi::dpl::sort()
			CPU		
			MSVC	icpx	icpx
vector<float>	4B	seq	4284	4013	4255
		unseq	4287	4046	4241
		par	478	477	Failed
vector<double>	8B	seq	4387	4206	4409
		unseq	4341	4160	4457
		par	533	525	Failed
vector<long long>	8B	seq	4151	3973	4056
		unseq	4094	4030	4043
		par	645	616	611
vector<int>	4B	seq	4073	3829	3936
		unseq	4043	3830	3971
		par	592	577	409
vector<short>	2B	seq	2512	2432	2474
		unseq	2516	2451	2503
		par	490	457	301
vector<char>	1B	seq	1178	1171	1191
		unseq	1183	1165	1189
		par	450	461	194





Findings & Outcomes of This Test Case:

1- No significant performance difference is measured between `seq` and `unseq` policies

2- Performance of `par` policy is **510% - 860%** higher than `seq` (running on CPU with 20 cores)

   a. **Only** in `STL`, while data type gets shorter performance difference drops down to 150%

3- With `par` policy, performance of `oneDPL` is 5% - 130% higher than `STL`. For real types oneDPL fails

   a. With `seq` policy `oneDPL` outperforms `STL` by 0.5% - 6%

4- Max performance difference between all 4B & 8B integer & real types is 12% (`seq`) & 50% (`par`)

5- Intel's DPC++ (icpx) compiler shows slightly better (average **3%**) compiling optimization over MSVC

5- Benchmarking Data Parallel Performance on Different Processors

Test Case:

- Sorting a `vector<T>` with **50 million elements**
  - Trying different data types (integer 1B, 2B & 4B length) for `T`
  - Trying different processor types (hence different libraries and compilers)

Objectives:

1- Benchmarking sorting performance on **different processors**: CPU, Nvidia GPU & Intel GPU

  a. Comparing sorting performance of different **data types** on each platform

Data Type		**Policy**	**Measured Time (ms)**			
			`std::sort()`	`oneapi::dpl::sort()`	`CUDA Thrust`	`SYCL + DPL`
			`CPU`		`Nvidia GPU`	`Intel GPU`
			MSVC	**icpx**	**nvcc**	**icpx**
`vector<int>`	4B	`par`	592	409	16 + 129	657
`vector<short>`	2B		490	301	15 + 185	512
`vector<char>`	1B		450	194	14 + 171	508

Note: Format of CUDA time is: "*Sorting Time*" + "*Time to Copy Data, Host to Device + Device to Host*"

Findings & Outcomes of This Test Case:

1- Maximum performance difference between sorting vectors with 1B, 2B & 4B integer data type elements is **30%** (STL & oneDPL on GPU), **38%** (CUDA + Thrust on GPU) & **110%** (oneDPL on CPU)

2- Performance of `oneDPL` is 45% - 130% higher than `STL` (both on CPU – 1974 08 05)

3- By checking CUDA (+Thrust) numbers it reveals that less than 10% of time is spent for sorting & the rest is spent for copying between RAM and G-RAM, copying shorter data types takes longer

6- Re-doing Performance Tests with Different Data Sets

Test Case:

- Sorting a `vector<int>` with **50 million elements**
  - Trying different data sets as input

Objective:

1- Measuring impact of changing **data sets** on performance of `sort()` function (for each platform)

Data Set	Data Type		Policy	Measured Time (ms)			
				`std::sort()`	`oneapi::dpl::sort()`	`CUDA Thrust`	`SYCL + DPL`
				CPU		Nvidia GPU	Intel GPU
				MSVC	**icpx**	**nvcc**	**icpx**
1	`vector<int>`	4B	`par`	**592**	409	16 + 129	657
2				550	391	15 + 119	658
3				582	386	15 + 116	660
4				567	393	15 + 126	654
5				574	383	15 + 128	678
Average ($\overline{x}$)				**573**	**392**	**139**	**661**
Standard Deviation (σ)				3%	3%	4%	1.5%

Note: Format of CUDA time is: *"Sorting Time"* + *"Time to Copy Data, Host to Device + Device to Host"*

Findings & Outcomes of This Test Case:

- Standard deviation for each set of tests can be considered as low, so changing data sets doesn't have much impact on sorting performance

The first test in above table (in red bold) repeated 100 times and as result: $\overline{x}$ = 600.3, σ= 13.86

EXPANDED UNCERTAINTY (U) = 2.757 (K = 2)

7- Scalability Test & Time Complexity of Sort () function

Test Case:

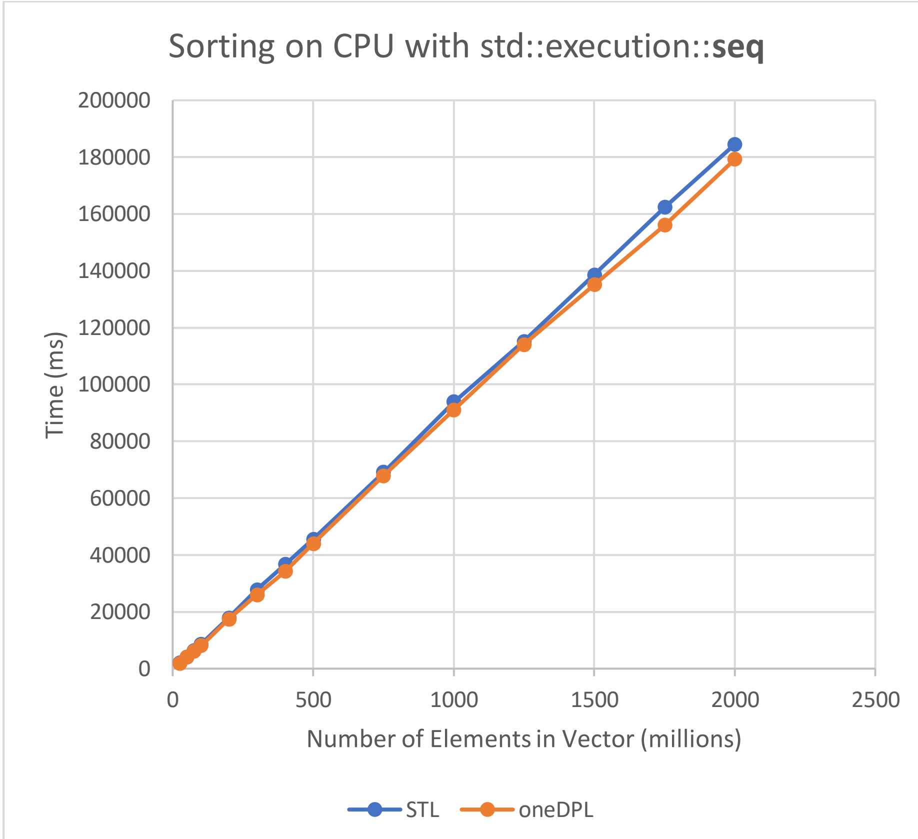
- Sorting a `vector<int>` with **different number of elements** in vector

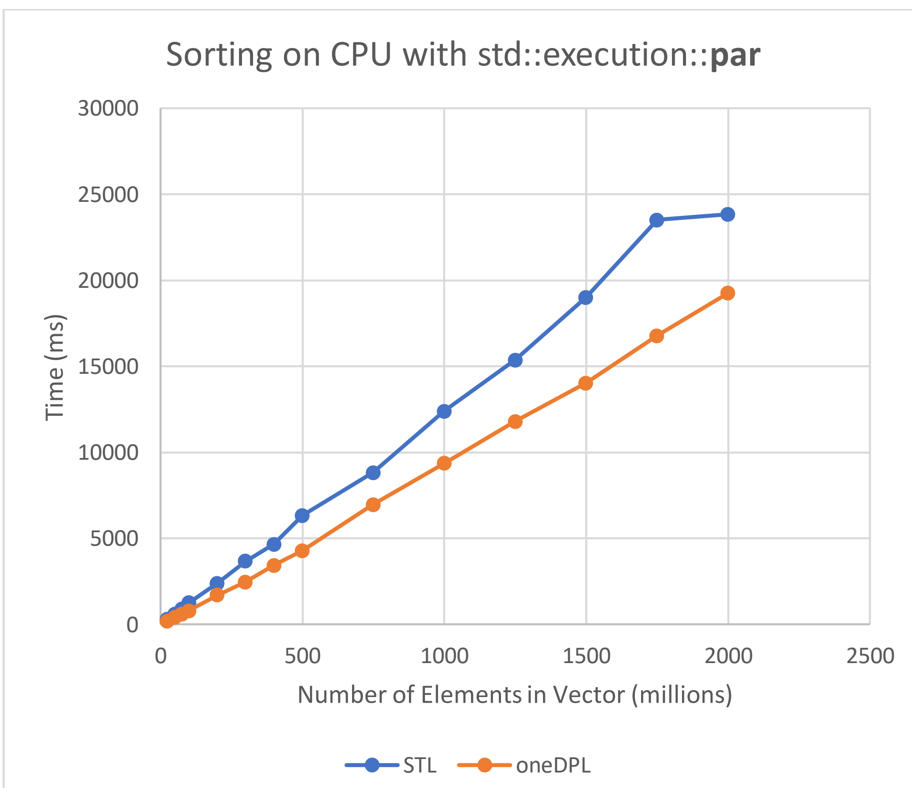
Objective:

1- Measuring performance scalability of `sort()` function in different libraries & comparing them

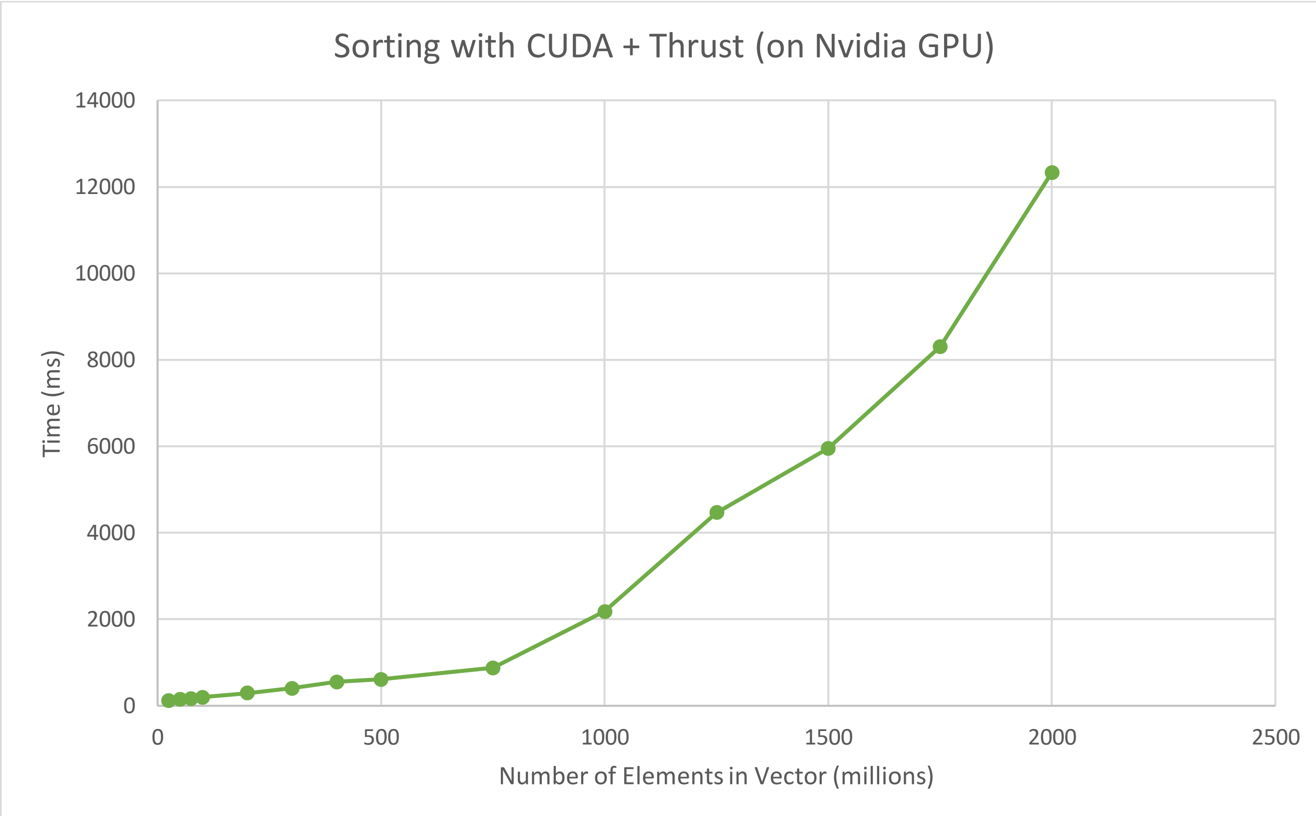
Number of Elements In `vector<int>`	Policy	Measured Time (ms)			
		`std::sort()`	`oneapi::dpl::sort()`	`CUDA Thrust`	`SYCL + DPL`
		`CPU`		`Nvidia GPU`	`Intel GPU`
		MSVC	**icpx**	**nvcc**	**icpx**
25 M	`seq`	1969	1834	-	
	`par`	286	182	9 + 102	558
50 M	`seq`	4073	3936	-	
	`par`	592	409	16 + 129	657
75 M	`seq`	6303	6009	-	
	`par`	902	588	22 + 140	820
100 M	`seq`	8592	8071	-	
	`par`	1261	792	29 + 167	1456
200 M	`seq`	17819	17383	-	
	`par`	2385	1697	58 + 235	1988
500 M	`seq`	45572	44010	-	
	`par`	6334	4282	135 + 472	3693
1000 M	`seq`	93807	91039	-	
	`par`	12383	9359	1219 + 959	6498
1500 M	`seq`	138511	135024	-	
	`par`	18999	14034	4679 + 1275	Failed
2000 M	`seq`	184487	179185	-	
	`par`	23836	19251	9980 + 2347	Failed

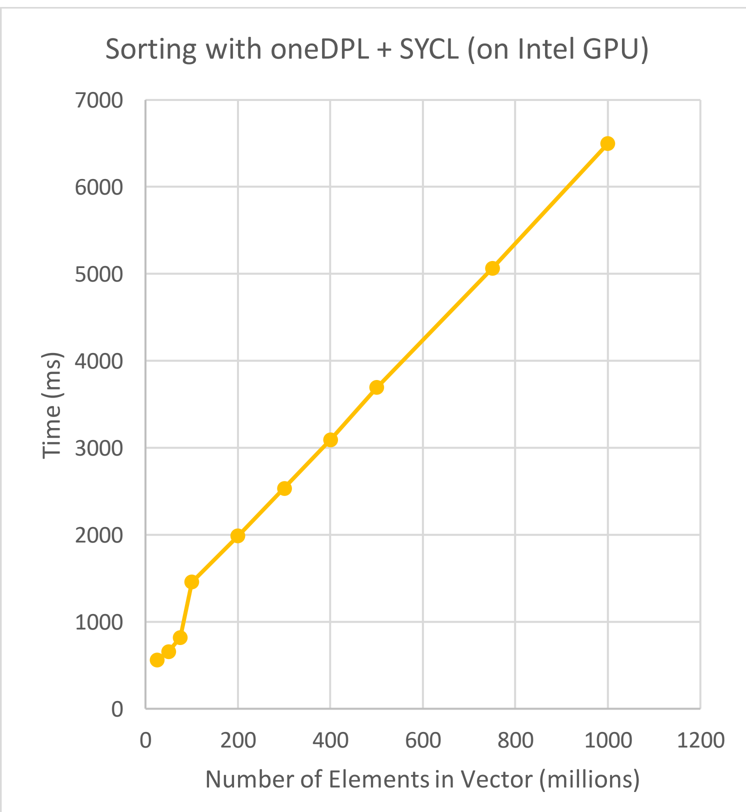
Note: Format of CUDA time is: "*Sorting Time*" + "*Time to Copy Data, Host to Device + Device to Host*"





In Above Diagrams Lower is Better





Findings & Outcomes are listed in section 10 at the end of this repot together with findings of other tests

8- Source Codes

1- **To compile C++ codes which only use standard C++ functions (STL):**

A. **Option 1:** Create "C++ Console App" project in Visual Studio, copy the code into the project, set language to C++20, set optimized for speed & built the project in 64-bit release mode

✓preferred

B. **Option 2:** open **"x64 Native Tools Developer Command Prompt for VS 2022"** command prompt (can be found in Visual Studio Tools) & compile the code with the following command

```
C:\> Cl /O2 /std:c++20 filename.cpp
```

2- **To compile C++ CUDA codes:**

A. **Option 1:** Create "CUDA 12.8 Runtime" project in Visual Studio, copy the code into the project & set GPU architecture to "compute_89, sm_89" or anything else matching your Nvidia GPU, set language to C++20, set optimized for speed, then built the project in 64-bit release mode

✓preferred

B. **Option 2:** open **"x64 Native Tools Developer Command Prompt for VS 2022"** command prompt (can be found in Visual Studio Tools) & compile the code with the following command:

```
C:\> nvcc --optimize 2 --gpu-architecture=compute_89 --gpu-code=sm_89 kernel.cu
```

3- **To compile C++ codes using oneDPL library of oneAPI toolkit (+ SYCL):**

A. **Option 1:** Create "DPC++ Console Application" project in Visual Studio, copy the code into the project, set "VC++ Directors" as described in section 3 & set language to C++20, set optimized for speed, then built the project in 64-bit release mode.

✓preferred

B. **Option 2:** open **"Intel oneAPI command line for Visual Studio 2022"** command prompt, then:

```
C:\> icpx -std=c++20 -fsycl -march=alderlake -O3 filename.cpp
```

Test Code for C++ (STL on CPU)

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <fstream>
#include <chrono>
#include <execution>

#define TYP int

int main()
{
	const int bufSize{ 50'000'000 };

	std::ifstream fileInputStream;
	fileInputStream.open("1.bin", std::ios::in | std::ios::binary);

	if (!fileInputStream.is_open())
		return -1;

	TYP* pBuffer;

	try{
		pBuffer = new TYP[bufSize];
	}
	catch (...){
		return -2;
	}

	fileInputStream.seekg(9'000'000, std::ios::beg); //avoiding zero-area of file
	fileInputStream.read((char *)pBuffer, bufSize * sizeof(TYP));
	fileInputStream.close();

	if (fileInputStream.bad())
		return -4;

	std::vector<TYP> v(pBuffer, pBuffer + bufSize);

	delete [] pBuffer;


	auto tStart = std::chrono::high_resolution_clock::now();
	std::sort(std::execution::par, v.begin(), v.end());      //The main Operation
	auto tEnd = std::chrono::high_resolution_clock::now();


	const std::chrono::duration<double, std::milli> passed = tEnd - tStart;
	size_t timeElapsedProcessingTotal_ms{ (size_t)passed.count() };


	std::cout << "Size of Vector = " << v.size() << ", Time elapsed = ";
	std::cout  << timeElapsedProcessingTotal_ms << " ms" << std::endl;
}
```

Test Code for Intel oneAPI (oneDPL on CPU)

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <fstream>
#include <chrono>
#include <execution>

#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>

#define TYP int

int main()
{
	const int bufSize{ 50'000'000 };

	std::ifstream fileInputStream;
	fileInputStream.open("1.bin", std::ios::in | std::ios::binary);

	if (!fileInputStream.is_open())
		return -1;

	TYP* pBuffer;

	try {
		pBuffer = new TYP[bufSize];
	}
	catch (...){
		return -2;
	}


	fileInputStream.seekg(9'000'000, std::ios::beg); //avoiding zero-area of file
	fileInputStream.read((char*)pBuffer, bufSize * sizeof(TYP));
	fileInputStream.close();

	if (fileInputStream.bad())
		return -4;

	std::vector<TYP> v(pBuffer, pBuffer + bufSize);

	delete[] pBuffer;

	auto tStart = std::chrono::high_resolution_clock::now();
	oneapi::dpl::sort(oneapi::dpl::execution::par, v.begin(), v.end());//The main
	auto tEnd = std::chrono::high_resolution_clock::now();

	const std::chrono::duration<double, std::milli> passed = tEnd - tStart;
	size_t timeElapsedProcessingTotal_ms{ (size_t)passed.count() };

	std::cout << "Size of Vector = " << v.size() << ", Time elapsed = ";
	std::cout  << timeElapsedProcessingTotal_ms << " ms" << std::endl;
}
```

Test Code for Intel oneAPI (SYCL + oneDPL on Intel GPU)

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <sycl/sycl.hpp>
#include <iostream>
#include <vector>
#include <fstream>
#include <chrono>

#define TYP int

int main()
{
	auto gpus = sycl::platform(sycl::gpu_selector_v).get_devices();
	sycl::queue queue_GPU(gpus[0]);

	const int bufSize{ 50'000'000 };

	std::ifstream fileInputStream;
	fileInputStream.open("1.bin", std::ios::in | std::ios::binary);

	if (!fileInputStream.is_open())
		return -1;

	TYP* pBuffer;

	try {
		pBuffer = new TYP[bufSize];
	}
	catch (...) {
		return -2;
	}

	fileInputStream.seekg(9'000'000, std::ios::beg); //avoiding zero-area of file
	fileInputStream.read((char*)pBuffer, bufSize * sizeof(TYP));
	fileInputStream.close();

	if (fileInputStream.bad())
		return -4;

	std::vector<TYP> v(pBuffer, pBuffer + bufSize);

	delete[] pBuffer;

	auto tStart = std::chrono::high_resolution_clock::now();
	oneapi::dpl::sort(oneapi::dpl::execution::make_device_policy(queue_GPU),
	                  v.begin(), v.end());  //The main Operation
	auto tEnd = std::chrono::high_resolution_clock::now();

	const std::chrono::duration<double, std::milli> passed = tEnd - tStart;
	size_t timeElapsedProcessingTotal_ms{ (size_t)passed.count() };

	std::cout << "Size of Vector = " << v.size() << ", Time elapsed = ";
	std::cout  << timeElapsedProcessingTotal_ms << " ms" << std::endl;
}
```

Test Code for CUDA (on Nvidia GPU)

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <iostream>
#include <fstream>
#include <chrono>

#define TYP int

int main()
{
	const int bufSize{ 50'000'000 };

	std::ifstream fileInputStream;
	fileInputStream.open("1.bin", std::ios::in | std::ios::binary);

	if (!fileInputStream.is_open())
		return -1;

	TYP* pBuffer;

	try {
		pBuffer = new TYP[bufSize];
	}
	catch (...) {
		return -2;
	}
	fileInputStream.seekg(9'000'000, std::ios::beg); //avoiding zero-area of file
	fileInputStream.read((char*)pBuffer, bufSize * sizeof(TYP));
	fileInputStream.close();

	if (fileInputStream.bad())
		return -4;

	thrust::host_vector<TYP> h_vec(pBuffer, pBuffer + bufSize);

	delete[] pBuffer;

	auto t1 = std::chrono::high_resolution_clock::now();
	thrust::device_vector<int> d_vec = h_vec;

	auto t2 = std::chrono::high_resolution_clock::now();
	thrust::sort(d_vec.begin(), d_vec.end()); //The main Operation

	auto t3 = std::chrono::high_resolution_clock::now();
	thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

	auto t4 = std::chrono::high_resolution_clock::now();
	const std::chrono::duration<double, std::milli> passed = t3 - t2;
	size_t timeElapsedProcessingTotal_ms{ (size_t)passed.count() };

	std::cout << "Size of Vector = " << h_vec.size() << ", Time elapsed = ";
	std::cout  << timeElapsedProcessingTotal_ms << " ms" << std::endl;
}
```

9- Test platform (Hardware & Software Specification)

Machine	Asus TUF Gaming F15
CPU	Intel Core i7 12700 (6 P-Core + 8 E-Core, in total **20 cores**) @ 4 GHz
RAM	32 GB @ 3.2 GHZ
GPU	Nvidia RTX 4070 (36 SMs, each has 128 cores, in total **4608 CUDA cores**)
	Intel Iris Xe (**96 cores**)
G-RAM (Nvidia)	8 GB @ 8 GHZ (128-bit bus)
OS	Windows 11 Enterprise 64-bit (24H2)
Compiler	Visual Studio 2022 V17.12 (MSVC) nvcc 12.8 for CUDA Intel icpx (dpc++) 2025.0
Compile Mode	64-bit Release Mode Optimized for speed; optimized for hardware architecture (if supported)
Language	ISO C++20
Test Data Sets	Data read from beginning of a movie file The main (1^{st}) data set: **the first 9MB is skipped** to reach data-rich area of the file with less zero & less identical bytes The other (N^{th}, N>1) data sets: Beginning of the file till end of the previous dataset (N-1) is skipped

Note: At the time of running all tests, there were no other [user] processes running on the test machine

10- Conclusion

In this technical report, performance of sorting algorithm in following scenarios is measured & being compared with each other:

Platform		Library / Toolkit	Compiler	Algorithm Function	Exec Policy
CPU	Intel	STL	MSVC, icpx	`std::sort()`	`seq` & `par`
		oneDPL	DPC++ (icpx)	`oneapi::dpl::sort()`	
GPU	Nvidia	CUDA + Thrust	NVCC	`thrust::sort()`	`par`
	Intel	oneDPL + SYCL	DPC++ (icpx)	`oneapi::dpl::sort()`	

Findings & Outcomes (of all Tests of This Report):

1 **No Significant** “**Performance Difference**” is measured between (applies to all libraries & platforms)

a. Sorting different data sets

b. Sorting with `seq` & `unseq` policies

c. Sorting with `par` & `par_unseq` policies (to keep the report short, output test data is not included)

d. Sorting `signed` & `unsigned` integer types (to keep the report short, output test data is not included)

e. Codes compiled for `C++17` & `C++20` (to keep the report short, output test data is not included here)

2 **A Small** “**Performance Difference**” is measured for sorting

a. With `seq` policy on CPU (all data types); `oneDPL` outperforms `STL` by 0.5% – 7%

b. Performance diff between `4B` & `8B integer` & `real` types (`seq` policy on CPU, all libs) < 12%

c. Performance diff between `4B` & `8B integer` & `real` types (`par` policy on CPU, all libs) < 50%

d. Performance diff between `1B`, `2B` & `4B integer` types (`par` policy on CPU, oneDPL) < 110%

e. Performance diff between `1B`, `2B` & `4B integer` types (`par` policy on GPU[1] & STL on CPU) < 40%

f. For STL, compiling optimization by Intel’s DPC++ is a bit (in average **3%**) better than MSVC

[1] Both Intel GPU (oneDPL + SYCL) & Nvidia GPU (CUDA + Thrust)

3 **A Great** “**Performance Difference**” is measured for sorting

   a. With `par` policy on CPU, `oneDPL` outperforms `STL` by 5% - 130% (for shorter types it is higher)

      i. For `1B`, `2B` & `4B integer` types, `oneDPL` always outperforms `STL` by more than 45%

   b. With `par` policy on CPU (all libs), sorting performance is **510% - 860%** higher than `seq` policy

      i. **Only** for `STL`, when data types get shorter performance diff drops down to 150%

4 Failed Cases

   a. oneDPL + SYCL (on Intel GPU) fails to sort vectors with more than 1000 million elements (int)

   b. oneDPL fails to sort vectors with real type (float & double) elements

5 When sorting smaller arrays (several millions of elements) with CUDA (+Thrust) on Nvidia GPU, less than 10% of time is spent for sorting & the rest is spent for copying data between RAM and G-RAM, once vector is growing in size this ratio decreases. For an array of 2000 million elements (int type), 80% of time is spent on sorting and only 20% of time is spent on copying

6 When sorting with `par` policy, CPU load reaches almost 100% while with `seq`, it is less than 10%

7 **Best absolute performance** is gained by “CUDA + Thrust + Nvidia GPU”. One of the reasons can be high quantity of cores and high performance of Nvidia GPU on the test machine.

8 **Time Complexity & Scalability**

Library / Toolkit	Platform	Scalability	Line Slope ≈		
STL	CPU	Linear	45^{o}	✓	-
oneDPL					Has offset to STL ✓
oneDPL + SYCL	Intel GPU	Linear (jumps in 100M elements)	30^{o}	✓✓	-
CUDA + Thrust	Nvidia GPU	Linear (2 lines)	30^{o}	✓✓	< 750M elements
			70^{o}	-	> 1000M elements

11- Table of Acronyms and Abbreviations

Acronym / Abbreviation	Stands For
App	Application
B	Byte
char	Character
cpp	C Plus Plus
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
Diff	Difference
DPC++	Data Parallel Compiler (for C++)
DPL	Data Parallel Library
E-Core	Efficient Core (CPU)
G	Giga (almost one billion)
GPU	Graphics Processing Unit
G-RAM	Graphics RAM (Random Access Memory)
HZ	Hertz
int	Integer
ISO	Independent System Operator
M	Mega (almost one million)
Max	Maximum
ms	milli second
MSVC	Microsoft Visual C
nvcc	Nvidia CUDA Compiler
oneDPL	One Data Parallel Library
OS	Operating System
par	Parallel
par_unseq	Parallel or Un-sequential
P-Core	Performance Core (CPU)
RAM	Random Access Memory
RTX	Ray tracing Texel eXtreme - Ray Tracing eXperiance
seq	Sequential
SM	Streaming Multiprocessors
STL	Standard Template Library (C++)
SYCL	System wide Compute Language
unseq	Un-sequential
V	Version
VS	Visual Studio

12- Disclaimer

The performance benchmarks and results presented in this article are based on specific tests conducted using C++ STL library, Nvidia's CUDA Thrust library, and Intel's oneAPI oneDPL library. The tests were performed using the author's developed C++ codes and represent the performance under the given conditions and configurations. The results may vary based on different hardware, software versions, compiler optimizations, and other factors.

The author does not claim the absolute superiority of one library / toolkit / compiler / hardware over another, as each library / toolkit / compiler / hardware may have different strengths and weaknesses depending on the use case. The information provided is intended for educational and informational purposes only and should not be considered as professional or investment advice.

Readers are encouraged to conduct their own tests and research to validate the findings and make informed decisions based on their specific requirements and environments.

13- References

1- **mini-Handbook of Parallel Programming**, post on Linked-in on 4 December 2024

https://www.linkedin.com/posts/pouya-ebadollahyvahed_mini-handbook-of-parallel-programming-activity-7270045199630749696-DZ24?utm_source=share&utm_medium=member_desktop

2- **Heterogeneous Programming; C++ with CUDA, HIP, openCL, SYCL & Intel oneAPI**, post on Linked-in on 14 January 2025

https://www.linkedin.com/posts/pouya-ebadollahyvahed_heterogeneous-programming-cpus-gpus-npus-activity-7284898204398063618-OCtT?utm_source=share&utm_medium=member_desktop