Extending and Reproducing pSTL-Bench: oneDPL Backend Integration and Enhanced Reproducibility

Oleg Bilovus
Department of Computer Science, University of Salerno
Fisciano (SA), Italy

Abstract

Describe in concise words what you do, why you do it (not necessarily in this order), and the main result. The abstract has to be self-contained and readable for a person in the general area. You should write the abstract last.

1 Introduction

Writing performance-portable and efficient parallel applications remains a significant challenge due to the diversity of modern hardware architectures. The emergence of parallel programming frameworks and, more recently, the introduction of parallel algorithms in the *C++17 Standard Template Library (STL)* aim to address this challenge by enabling developers to write code that runs efficiently across CPUs and GPUs using a standardized interface.

pSTL-Bench [1] is a benchmark suite originally developed to quantitatively evaluate the performance of individual parallel STL algorithms across various compiler frameworks (such as GCC, Intel's icpx, NVIDIA HPC SDK) and backends (including TBB, HPX, OpenMP, and CUDA). It provides a focused, micro-benchmarking approach that avoids the complexities of full applications and highlights the relative performance characteristics of different STL implementations.

This Work. In this work, the main results and experiments from the original pSTL-Bench paper are reproduced. Beyond reproduction, the project is then extended by incorporating support for the *oneAPI DPC++ Library (oneDPL)* backend, enabling the evaluation of Intel's parallel STL implementation based on the *SYCL programming model* on both CPU and GPU platforms

To further enhance the usability and reproducibility of the benchmark suite, automation tools are introduced, including Ansible playbooks for environment setup and benchmark execution, R scripts for performance analysis, and expanded documentation for ease of deployment and adding new backends.

Related Work. This work builds directly on pSTL-Bench. While performance portability and parallelism have been widely studied in the context of full applications [2] and frameworks such as Kokkos [3] and RAJA [4], no prior work has reproduced or extended pSTL-Bench, nor added support for the oneDPL backend or focused on improving its reproducibility.

2 Background

This section introduces the relevant concepts and context for evaluating the scalability of parallel algorithms implemented in the C++ Standard Template Library (STL). It defines the scope of the problem, outlines the algorithms under analysis, and describes the supporting frameworks and benchmarking strategies used for performance assessment.

Parallel STL in C++. Since C++17, the Standard Template Library (STL) includes parallel versions of many common algorithms, enabling hardware parallelism through familiar interfaces. Compiler support varies, relying on different runtimes and libraries to target platforms such as multi-core CPUs and GPUs, with differences in scheduling, memory use, and synchronization.

Problem Definition. The primary objective is to analyze how different implementations of parallel STL algorithms scale with increasing problem sizes and hardware resources. This includes identifying the computational overheads and the performance gains achieved through parallel execution. A detailed performance comparison is necessary to understand which combinations of compilers and backends provide the most efficient use of available computing resources.

Algorithms. The study focuses on five representative algorithms that embody diverse computational patterns:

- **find** – Searches for a target value within a range using linear traversal.
- **for_each** – Executes a loop with k_{it} increments per element to simulate computation and stores the result. k_{it} is a parameter that can be adjusted to control the workload.
- **inclusive_scan** – Produces a prefix sum, where each element in the output is the sum of all previous elements plus the current element, requiring synchronization.
- **reduce** – Computes a single result by summing all elements in a range, which can be parallelized but requires synchronization to combine results.
- **sort** – Rearranges elements into ascending order, requiring synchronization and data movement.

Backends. In this work, support for the *oneAPI DPC++ Library (oneDPL)* is added, which is based on the SYCL programming model and enables parallel STL execution on both CPUs and GPUs.

The original pSTL-Bench suite includes the following backends:

- **GNU Parallel STL** – The GCC implementation of parallel algorithms using OpenMP.
- **TBB** – A C++ library offering task-based parallel algorithms and data structures.
- **HPX** – A C++ runtime for fine-grained parallel and distributed applications.
- **OpenMP** – A standard API for shared-memory parallel programming.
- **CUDA** – NVIDIA's platform for general-purpose GPU computing.

The evaluation covers various compiler-backend combinations, including those based on GNU, Intel, and NVIDIA compilers.

Automation. To facilitate reproducibility and ease of use, the project includes automation tools:

- **Ansible** – An agentless automation tool that simplifies the setup of the benchmark environment, including installation

of dependencies, configuration of the system, compilation of the benchmark suite, and execution of tests.
- **Semaphore UI** – A web-based interface for managing and monitoring the execution of ansible playbooks, providing a user-friendly way to trigger tests and view results.
- **R Scripts** – Scripts for analyzing performance data, generating plots similar to those in the original paper. and providing insights into the performance characteristics of different implementations.

3 Benchmark Setup

This section describes the hardware and software setup used for the benchmarking experiments, including platform specifications, compiler versions, and the configuration of the parallel STL implementations.

Hardware Platform. The benchmarks were executed on a *DELL G16 7630* laptop connected to external power, with the hardware specifications shown in Table 1.

Table 1: Hardware configuration

CPU [5]	
Processor	Intel Core i9-13900HX
Cores	24 (8 P-cores + 16 E-cores)
Core Frequency	2.20 GHz (up to 5.40 GHz)
Threads	32 threads
Memory	32 GB RAM
GPU [6]	
GPU	NVIDIA GeForce RTX 4070 (mobile)
CUDA Cores	4608 CUDA cores
Core Frequency	1230 MHz (up to 2175 MHz)
Memory	8 GB GDDR6 VRAM

Software Platform. The benchmarks were executed on a Linux-based operating system, specifically *Ubuntu 25.04 Desktop*. The desktop version was used due to GPU driver compatibility issues with the server edition. Prior to execution, the CPU frequency governor was set to *performance* mode to ensure maximum processor performance [7].

Compilers and Libraries. The compilers and libraries used in the experiments are listed in Table 2 and Table 3, respectively.

Table 2: Compilers

Compiler	Version
g++	14.2.0
Intel icpx	2025.1.1
NVIDIA nvc++	25.3-0

Table 3: Libraries

Library	Version
oneDPL	2022.8
TBB	2022.1
HPX	1.9.1
NVOMP	25.3
CUDA	12.8

4 Experiment Details

All aspects of the original pSTL-Bench experiments are replicated in this work, with one exception: the problem size in certain experiments is reduced from 2^{30} to 2^{29}. This adjustment was necessary due to runtime crashes encountered by the oneDPL backend when using the larger problem size. The corresponding error message is shown in Listing 1.

Listing 1: oneDPL GPU runtime error output

```
<CUDA>[ERROR]:
UR CUDA ERROR:
  Value:           2
  Name:            CUDA_ERROR_OUT_OF_MEMORY
  Description:     out of memory
  Function:        allocateMemObjOnDeviceIfNeeded
  Source Location: /builds/oneapi-core/oneapi-release-
      ↪ package/intel-llvm-mirror/build/_deps/unified-
      ↪ runtime-src/source/adapters/cuda/memory.cpp:442

<CUDA>[ERROR]:
UR CUDA ERROR:
  Value:           2
  Name:            CUDA_ERROR_OUT_OF_MEMORY
  Description:     out of memory
  Function:        allocateMemObjOnDeviceIfNeeded
  Source Location: /builds/oneapi-core/oneapi-release-
      ↪ package/intel-llvm-mirror/build/_deps/unified-
      ↪ runtime-src/source/adapters/cuda/memory.cpp:442

terminate called after throwing an instance of 'sycl::_V1
    ↪ ::exception'
  what():  Native API failed. Native API returns: 38 (
      ↪ UR_RESULT_ERROR_OUT_OF_HOST_MEMORY)
```

Plots. All plots presented in this work were generated using the R scripts available in the forked repository. Figures were exported at a resolution of 300 DPI to preserve detail and support zooming without quality loss.

The speedup plots are presented using a log-linear scale, where the x-axis is logarithmic, representing the number of threads, and the y-axis is linear, indicating the speedup factor. This format provides a clearer comparison of performance across varying thread counts and backend implementations.

Backends. The compiler names used in the plots differ from those in the original pSTL-Bench paper, as they are directly generated by the benchmarking tool. These names can be mapped to their original counterparts as follows:
- *GNU-OMP* corresponds to *GCC-GNU*.
- *GNU-** corresponds to *GCC-**.
- *IntelLLVM-TBB* corresponds to *ICC-TBB*.
- *NVHPC-** corresponds to *NVC-**.

To maintain consistency, the structure and subsection titles align with those of the original work.

5 Experimental Results

This section presents the results of experiments designed to evaluate the performance of parallel STL algorithms across various compiler-backend combinations. While each experiment references the original results from the pSTL-Bench paper, those plots are not reproduced here. Readers are encouraged to consult the original publication for direct visual comparisons.

5.1 The Impact of Memory Allocation

The first experiment investigates the effect of the custom memory allocator used in pSTL-Bench on the performance of parallel STL algorithms. Whereas the original study employed a problem size of 2^{30}, this setup uses a reduced size of 2^{29}, as described in Section 4.

In the original paper, the custom allocator demonstrated significant performance improvements—particularly for the *for_each* and *reduce* algorithms—achieving speedups of up to 1.5x over the default allocator.

In this evaluation, no substantial performance difference was observed between the custom and default allocators. The measured results are presented in Figure 1.

Despite the minimal performance impact in this setting, the custom allocator was retained to ensure consistency with the methodology of the original study.

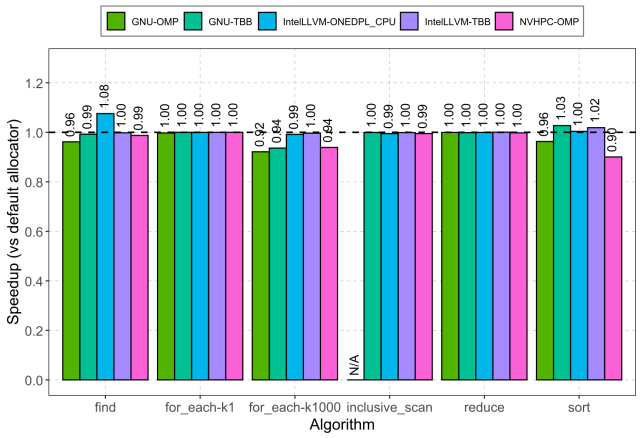


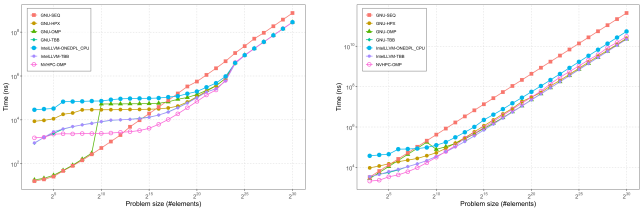
Figure 1: Speedup when using custom parallel allocator with 32 threads and a problem size of 2^{29}. Higher is better.

5.2 X::for_each

This experiment evaluates the performance of the *for_each* algorithm. The parameter k_{it} represents the arithmetic intensity of the workload. For higher values of k_{it}, performance is expected to approach ideal speedup.

Figure 2 presents the execution time for *for_each* with minimal and maximal k_{it} values. The results closely resemble those reported in the original pSTL-Bench paper. For $k_{it} = 1$, the NVIDIA OpenMP backend achieves the best performance. In contrast, for $k_{it} = 1000$, all backends exhibit comparable execution times.

The oneDPL backend on CPU consistently performed worse than the other backends across both configurations.

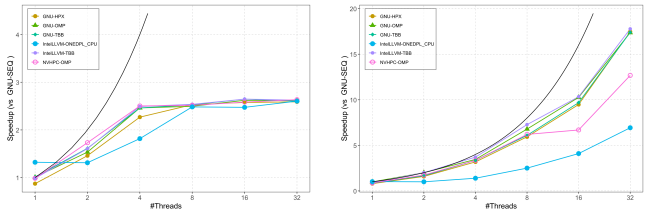


(a) $k_{it} = 1$. **(b)** $k_{it} = 1000$.

Figure 2: Results for benchmark X::for_each. Problem size scaling using all cores except for the GNU-SEQ implementation. Lower is better.

Figure 3 shows the strong scaling results for the *for_each* algorithm with 2^{29} elements. Compared to the original paper, the results differ slightly. In the original study, all backends achieved ideal speedup for $k_{it} = 1000$ and near-ideal speedup for $k_{it} = 1$.

In this evaluation, the oneDPL backend on CPU underperforms relative to the others. Additionally, all backends begin to lose scalability beyond 8 threads, which is likely due to the hardware architecture—specifically, the presence of 8 performance cores (P-cores) and 16 efficiency cores (E-cores).



(a) $k_{it} = 1$. **(b)** $k_{it} = 1000$.

Figure 3: Results for benchmark X::for_each. Strong scaling with 2^{29} elements. Higher is better.

5.3 X::find

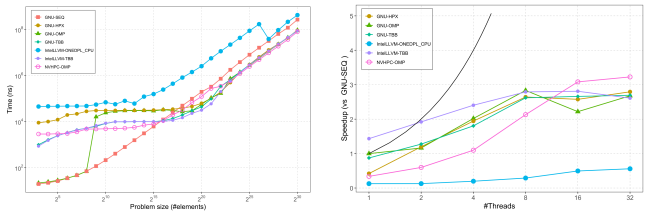
The *find* algorithm performs a linear search over a range to locate a target value.

Figure 4 presents the execution time and speedup for the *find* algorithm. The results are consistent with those reported in the original pSTL-Bench paper, where sequential execution significantly outperforms parallel implementations for small problem sizes.

As the problem size increases, parallel backends gradually begin to outperform the sequential version.

However, the maximum observed speedup remains relatively modest, with the best performance achieved by the NVIDIA OpenMP backend.

Across all problem sizes, the oneDPL backend on CPU consistently delivered the lowest performance—even falling below that of the sequential implementation.



(a) Problem scaling. Lower is better. **(b) Strong scaling with 2^{29} elements. Higher is better.**

Figure 4: Results for X::find.

5.4 X::inclusive_scan

The *inclusive_scan* algorithm computes a prefix sum, where each element in the output is the sum of all preceding elements plus the current element.

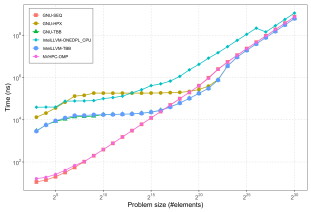
The GNU OpenMP implementation is omitted from the results, as it does not support this algorithm. Although the NVIDIA OpenMP backend also lacks native support, it falls back to a sequential implementation.

Figure 5 shows the execution time and speedup for the *inclusive_scan* algorithm. The results are consistent with those in the original pSTL-Bench paper, where sequential execution significantly outperforms parallel implementations for small problem sizes.

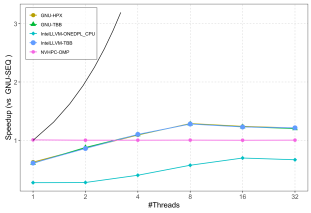
As the problem size increases, the performance of the parallel backends begins to approach that of the sequential version, with the TBB implementations achieving the best results overall.

Observed speedups remain modest and tend to degrade when using more than 8 threads. This decline is likely due to the underlying hardware architecture, which consists of 8 performance cores (P-cores) and 16 efficiency cores (E-cores).

Across all problem sizes, the oneDPL backend on CPU consistently exhibited the lowest performance—even falling below that of the sequential baseline.



(a) Problem scaling. Lower is better.



(b) Strong scaling with 2^{29} elements. Higher is better.

Figure 5: Results for X::inclusive_scan.

5.5 X::reduce

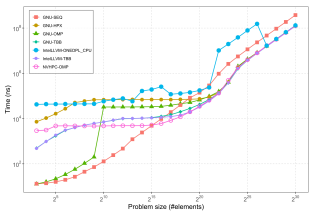
The *reduce* algorithm computes a single result by summing all elements in a given range.

Figure 6 presents the execution time and speedup for the *reduce* algorithm. The results are consistent with those reported in the original pSTL-Bench paper and align with the behavior observed in the *inclusive_scan* experiment.

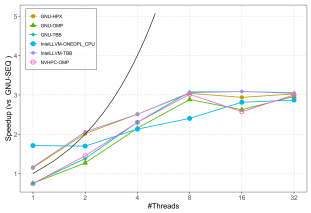
Sequential execution significantly outperforms parallel implementations for small problem sizes. However, as the problem size increases, parallel backends begin to surpass the sequential version in performance.

At a problem size of 2^{17} elements, most backends reach performance levels comparable to the sequential baseline, with the exception of the oneDPL backend on CPU, which achieves similar performance only at 2^{27} elements and beyond.

The maximum observed speedup is typically reached at 8 threads for most backends. Beyond this point, performance begins to degrade, likely due to the limitations of the underlying hardware architecture.



(a) Problem scaling. Lower is better.



(b) Strong scaling with 2^{29} elements. Higher is better.

Figure 6: Results for X::reduce.

5.6 X::sort

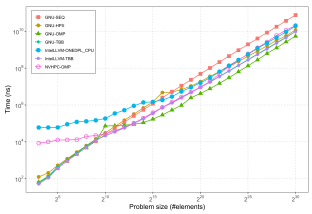
The *sort* algorithm rearranges elements in ascending order.

Figure 7 presents the execution time for the *sort* algorithm.

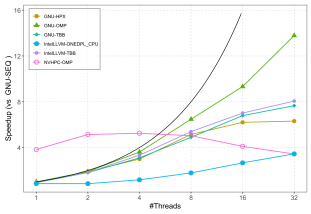
The results are consistent with those reported in the original pSTL-Bench paper. The NVIDIA OpenMP backend achieves the best performance for small problem sizes, while the GNU OpenMP backend performs best at larger scales.

Among all backends, only GNU OpenMP achieves near-ideal speedup. In contrast, the others exhibit diminishing returns as the number of threads increases.

The oneDPL backend on CPU ranks among the weakest in the problem size scaling experiment and performs the worst in the strong scaling scenario, falling significantly short of ideal speedup.



(a) Problem scaling. Lower is better.



(b) Strong scaling with 2^{29} elements. Higher is better.

Figure 7: Results for X::sort.

5.7 Summary of Results

Speedup. Table 4 summarizes the experimental results, reporting the maximum speedup achieved by each backend across the five parallel STL algorithms. Speedup is shown for 8 and 32 threads to reflect the underlying hardware architecture, which includes 8 performance cores (P-cores), 16 efficiency cores (E-cores), and 8 hyperthreads.

As observed in the original pSTL-Bench paper, all parallel implementations outperform the sequential version across all algorithms. The average speedup across all backends is approximately 3x.

Among the backends, oneDPL consistently achieved the lowest speedup but still outperformed the sequential baseline in all algorithms except *find* and *inclusive_scan*.

Efficiency. Table 5 reports the maximum number of threads for which the efficiency—defined as $\frac{\text{Speedup}}{\text{Number of Threads}}$—remains above 70% relative to sequential execution.

The results indicate that most backends maintain high efficiency up to 8 threads, which aligns with the processor's architecture, specifically the presence of 8 performance cores.

Binary Size. Table 6 presents the binary sizes generated by the different compiler-backend combinations.

The results show considerable variation in binary size, with the oneDPL backend on CPU producing the largest binaries. This is consistent with the complexity of its implementation, which is generally more elaborate compared to the other backends.

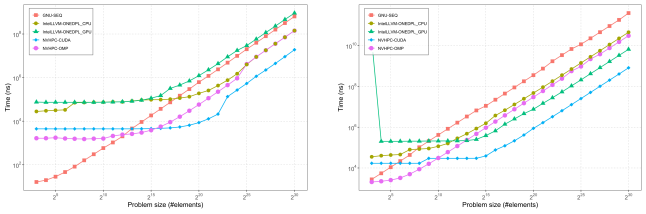
5.8 Performance on GPU

This experiment focuses on the performance of parallel algorithms on GPU, evaluated using the NVIDIA HPC SDK and oneDPL with the CUDA compiler. All benchmarks are executed using *float* precision instead of *double*, as GPUs are traditionally more efficient with single-precision operations.

for_each. Figure 8 shows the performance of the *for_each* algorithm on GPU for two levels of arithmetic intensity: $k_{it} = 1$ and $k_{it} = 1000$.

The results differ from those in the original pSTL-Bench paper. When the arithmetic intensity is low ($k_{it} = 1$), GPU implementations previously performed significantly worse than CPU implementations. In this evaluation, however, the NVHPC-CUDA backend outperforms the sequential baseline, while the oneDPL-GPU backend performs slightly worse.

When the arithmetic intensity is high ($k_{it} = 1000$), the results align more closely with the original study. Both GPU backends outperform the sequential version, with NVHPC-CUDA maintaining a clear advantage over oneDPL-GPU.



(a) $k_{it} = 1$. **(b)** $k_{it} = 1000$.

Figure 8: Results for benchmark X::for_each on GPU. Problem size scaling using *float*. Lower is better.

Float vs Double. To evaluate the impact of data type on GPU performance, the *for_each* benchmark with $k_{it} = 1000$ was executed using both *float* and *double* types. The results are shown in Figure 9.

The NVHPC-CUDA backend shows a substantial performance gain when using *float*, highlighting the typical advantage of single-precision on GPUs. In contrast, the oneDPL-GPU backend exhibits minimal difference between *float* and *double*, indicating less sensitivity to data type.

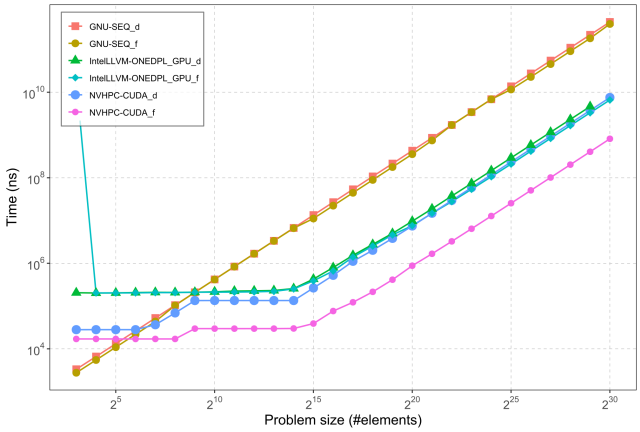


Figure 9: Performance comparison of *for_each* with $k_{it} = 1000$ using *float* and *double*. Lower is better.

6 Conclusions

Here you need to summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the report, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across such as high-level statements (e.g., we believe that .... is the right approach to .... Even though we only considered x, the .... technique should be applicable ....) You can also formulate next steps if you want. Be brief. After the conclusions there are only the references.

References

[1] R. Laso, D. Krupitza, and S. Hunold, "Exploring Scalability in C++ Parallel STL Implementations" in *Proceedings of the 53rd International Conference on Parallel Processing (ICPP '24)*, Association for Computing Machinery, New York, NY, USA, 2024, pp. 284–293. https://doi.org/10.1145/3673038.3673065

[2] W. -C. Lin, T. Deakin and S. McIntosh-Smith, "Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems" in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Dallas, TX, USA, 2022, pp. 36-47. https://doi.org/10.1109/PMBS56514.2022.00009

[3] H. C. Edwards and C. R. Trott, "Kokkos: Enabling Performance Portability Across Manycore Architectures" *2013 Extreme Scaling Workshop (xsw 2013)*, Boulder, CO, USA, 2013, pp. 18-24. https://doi.org/10.1109/XSW.2013.7

[4] D. A. Beckingsale et al., "RAJA: Portable Performance for Large-Scale Scientific Applications" *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Denver, CO, USA, 2019, pp. 71-81. https://doi.org/10.1109/P3HPC49587.2019.00012

[5] Intel, "Intel Core i9-13900HX Processor" *Intel SKU*, https://www.intel.com/content/www/us/en/products/sku/232171/intel-core-i913900hx-processor-36m-cache-up-to-5-40-ghz/specifications.html

[6] NVIDIA, "GeForce RTX 4070 Laptop GPU" *GeForce RTX 40 Series Laptops*, https://www.nvidia.com/en-us/geforce/laptops/40-series/

[7] Google Benchmark, "Reducing Variance, Disabling CPU Frequency Scaling" *Google Benchmark Documentation*, https://google.github.io/benchmark/reducing_variance.html

Table 4: Speedup against GCC's sequential implementation with for 8 and 32 threads. Problem size is 2^{29}. Higher is better.

Backend	find	for_each $k_{it} = 1$	for_each $k_{it} = 1000$	inclusive_scan	reduce	sort
GNU-TBB	2.6 | 2.6	2.5 | 2.6	6.0 | 17.3	1.2 | 1.2	3.0 | 3.0	4.8 | 7.6
GNU-OMP	2.8 | 2.6	2.4 | 2.6	6.7 | 17.3	N/A | N/A	2.8 | 2.9	6.4 | 13.7
GNU-HPX	2.6 | 2.7	2.5 | 2.5	5.9 | 17.4	1.2 | 1.2	3.0 | 3.0	5.1 | 6.3
IntelLLVM-TBB	2.7 | 2.6	2.5 | 2.6	7.2 | 17.7	1.2 | 1.2	3.0 | 3.0	5.3 | 8.0
NVHPC-OMP	2.1 | 3.2	2.5 | 2.6	6.2 | 12.6	1.0 | 1.0	3.0 | 3.0	5.0 | 3.4
IntelLLVM-ONEDPL_CPU	0.2 | 0.5	2.4 | 2.6	2.4 | 6.9	0.5 | 0.6	2.4 | 2.8	1.7 | 3.4

Table 5: Maximum number of threads such that efficiency = $\frac{\text{Speedup}}{\text{Number of Threads}}$ is above 70% (compared to the seq. execution). Problem size is 2^{29}. Higher is better.

Backend	find	for_each $k_{it} = 1$	for_each $k_{it} = 1000$	inclusive_scan	reduce	sort
GNU-TBB	1	2	8	N/A	1	4
GNU-OMP	1	2	8	N/A	1	8
GNU-HPX	N/A	2	8	N/A	2	4
IntelLLVM-TBB	2	2	8	N/A	2	4
NVHPC-OMP	N/A	2	8	1	2	4
IntelLLVM-ONEDPL_CPU	N/A	1	1	N/A	2	2

Table 6: Binary sizes for the different compilers and backends used in the experiments. Lower is better.

Compiler Backend	GNU SEQ	GNU TBB	GNU OMP	GNU HPX	IntelLLVM TBB	NVHPC OMP	NVHPC CUDA	IntelLLVM ONEDPL_CPU	IntelLLVM ONEDPL_GPU
Bin. size (MiB)	0.9	1.9	0.9	4.1	3	1.6	8	30	18