# GPU Acceleration for the C++ Standard Template Library

Christian DeLozier
delozier@cis.upenn.edu
University of Pennsylvania

## Abstract

Modern programmers must exploit parallelism for performance gains, possibly through the use of an attached or on-chip GPU. To take advantage of the GPU in C++ programs, the programmer must use either a new language (CUDA or OpenCL) or an external library (Thrust). Rather than requiring that programmers learn new tools, modify existing code, and change software development practices, the C++ Standard Template Library (STL) can be modified to automatically accelerate common algorithms using the GPU. This paper presents a GPU accelerated version of the C++ STL, *libcxxgpu*. Using the *thrust* library, function calls to the algorithms provided by the C++ STL are executed on the GPU, depending on a set of heuristics that determine when to use the CPU and when to use the GPU. In this paper, we detail the implementation of the accelerated library, highlight challenges encountered, and analyze the performance factors that determine which device should be used.

**CR Categories:** D.1.3 [Programming Techniques]: Concurrent Programming—Parallel Programming D.2.2 [Software Engineering]: Design Tools and Techniques—Software Libraries

**Keywords:** GPGPU, C++, STL, performance, parallel

## 1 Introduction

Due to the lack of frequency scaling in modern processors, applications must exploit parallelism for increased performance. For massively parallel algorithms, an attached or on-chip Graphics Processing Unit (GPU) can enable large performance gains (as much as 100x in some cases). Unfortunately, many applications have been developed for sequential execution and do not take advantage of parallel execution, which would otherwise provide increased performance.

The C++ Standard Template Library (STL) provides common containers and algorithms, which have been heavily optimized for sequential execution. Using STL functions, C++ programmers can avoid implementing and optimizing complex algorithms, such as `sort`. Though these algorithms may benefit from parallel execution, the STL versions of these algorithms are sequential, which limits their performance.

To maintain its reputation as a high-performance language, C++ must allow programs to take advantage of GPU execution. To enable GPU acceleration in C++ programs, the *thrust* library implements matching GPU algorithms for the algorithms provided by the STL [Hoberock and Bell 2012]. *Thrust* abstracts away the details of low-level CUDA function calls, such as `cudaMemCpy`. However, to use *thrust*, programmers must modify existing code, call *thrust* functions instead of STL functions, and tune programs to determine when the use of *thrust* will be beneficial for performance.

*This paper was written for the final project of CIS 565: GPU Programming and Architecture in the Spring 2012 semester.*

```
template <typename T>
void
sort(Iterator<T> first, Iterator<T> last){
  if(should_use_gpu(last - first, sizeof(T))){
    gpu_sort(first,last);
  }else{
    ... Perform Sequential Sort ...
  }
}

template <typename T>
void
gpu_sort(Iterator<T> first, Iterator<T> last){
  device_vector<T> device(last-first);
  copy_to_gpu(first,last,device);
  thrust::sort(device.begin(),device.end());
  copy_from_gpu(first,last,device);
}
```

**Figure 1:** *Simplified Implementation of GPU Accelerated* `sort`

Rather than require that programmers use *thrust* directly in C++ programs, *libcxxgpu* provides a version of the STL that automatically (1) determines when GPU acceleration will be beneficial and (2) uses *thrust* to execute STL algorithms on the GPU. Through a comparison of the performance of CPU and GPU execution of STL algorithms, we have identified heuristics that inform the runtime decision of which device to use. This paper describes how an implementation of the STL can be modified for GPU acceleration using *thrust* and analyzes the performance of GPU acceleration for STL algorithms, including some cases in which GPU acceleration was not beneficial.

## 2 Approach

Ideally, GPU acceleration would be applied to a single implementation of the C++ STL and used for all platforms. Though the behavior of the STL is standardized [International Standard ISO/IEC 14882:2011. 2011], multiple implementations of the STL exist for various operating systems and compilers. Therefore, *libcxxgpu* attempts to minimize the amount of invasive code changes necessary to GPU accelerate each implementation of the STL. Figure 1 shows an example of the basic code modifications applied to each accelerated STL function. The functions provided in the *libcxxgpu* headers can be separated into algorithms and utilities.

**Algorithms** The `gpu_algo` header provides functions like `gpu_sort`, `gpu_find`, and `gpu_transform`. In general, the GPU accelerated functions simply create a `device_vector`, copy the input data to the GPU, execute a *thrust* function on the data, and copy the output data back to the CPU. At this time, the functions `sort`, `find`, `min_element`, `max_element`, `transform`, `set_union`, `set_intersection`, `set_difference`, and `set_symmetric_difference` have been implemented and evaluated.

**Heuristics** The `gpu_util` header provides the `should_use_gpu` function which uses runtime heuristics to determine whether to execute an algorithm using the CPU or the GPU. Overall, these heuristics must accomplish three goals. First, the heuristics must execute efficiently to minimize their impact on the runtime of CPU execution. Second, the heuristics must accurately predict whether CPU or GPU execution will be more efficient. Finally, the heuristics must be conservative to ensure that GPU execution will never diminish the performance of STL algorithms.

The heuristics are based on two factors, referred to as the `bandwidth_factor` and the `instruction_factor`. We describe the performance results that motivated these heuristics in Section 3. The `bandwidth_factor` approximates the work required to copy data between the CPU and GPU, which was found to be a major bottleneck for GPU execution. The `instruction_factor` approximates the total amount of work done on the input data, which was found to be the major bottleneck for CPU execution. To determine whether or not to use the GPU, *libcxxgpu* computes the difference between the two factors and chooses the GPU if the `instruction_factor` is greater than the `bandwidth_factor`. These factors are computed as follows, where bandwidth, cores, and clock are normalized to the attributes of the GTX 570. *Instructions* is an approximation of the number of instructions executed per input data element. This value can be found through experimentation and should remain constant across all hardware and software platforms. *Size* and *Element_Size* are determined from the iterator arguments to the called STL function.

$$\begin{aligned} instruction\_factor &= Size * Instructions \\ bandwidth\_factor &= Element\_Size * GPU\_factor \\ GPU\_factor &= DF * SF * bandwidth * cores * clock \\ DF &= GPU\_factor(\text{GTX 570}) \\ SF &= \text{Default Size Factor (constant)} \end{aligned}$$

Before computing and comparing both factors, the `should_use_gpu` function first checks whether *Instructions* is less than a minimum cutoff. For the evaluated GPUs, the minimum cutoff was placed at fifteen instructions per element. As a motivation for this early cutoff, the `find` algorithm executes approximately three instructions per data element (two compares and an increment). In the isolation of a single STL function call, GPU acceleration is not beneficial for such a small number of instructions per element. After computing the `instruction_factor`, a second early cutoff is applied for values less than a minimum total number of instructions. For the evaluated GPUs, the minimum `instruction_factor` was set at three-million.

## 3 Evaluation

The performance of CPU and GPU execution of the implemented STL functions was evaluated on two systems, one with an Intel Core2 processor and an NVIDIA 9600 GT GPU and another with an Intel Core i7 processor and an NVIDIA GTX 570 GPU. The 9600 GT has a bus bandwidth of 58 GB/s and 64 cores with a clock of 1625 MHz. The GTX 570 has a bus bandwidth of 152 GB/s and 480 cores with a clock of 1464 MHz. To test the performance of STL functions, we developed a set of microbenchmarks that time the execution of multiple iterations of STL algorithms on a container with a specified number of elements. All microbenchmarks were compiled with `O3` optimizations using the *nvcc* compiler.

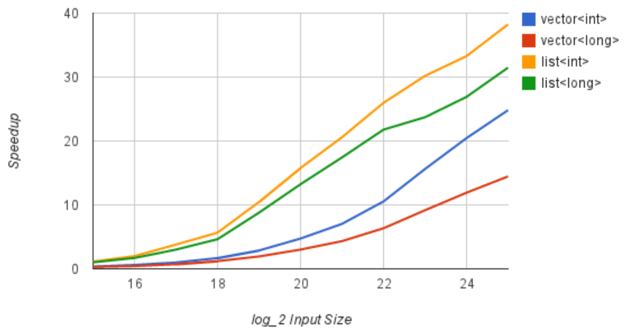

**Figure 2:** *Comparison of GPU acceleration gains for `sort` on `vector` and `list` on the GTX 570*

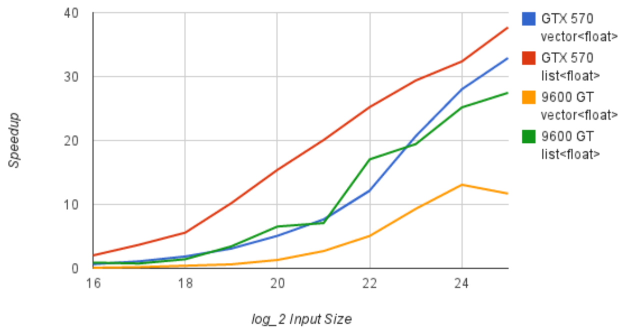

**Figure 3:** *Comparison of `sort` on 9600 GT and GTX 570*

**sort** In Figure 2, we demonstrate the performance benefits of vector and list sorting on the GTX 570. Though copying between devices takes longer for a list than a vector, the increase in performance for GPU acceleration is greater for lists because the STL `sort` function is better optimized for random-access containers, such as vectors and arrays, than for linked containers, such as lists.

In Figure 3, we compare the speedups provided by GPU acceleration on both evaluated systems. On the GTX 570, vector `sort` is beneficial for any container size larger than 128,000 elements. At a maximum size of 32,000,000 elements, GPU acceleration provides a speedup of 32.9x for vector `sort` and 37.7x for list `sort`. On the 9600 GT, vector `sort` is beneficial for any container size larger than 1,000,000 elements. At the maximum tested size, GPU acceleration provides a speedup of 11.7x for vector `sort` and 27.5x for list `sort`.

**transform** The `transform` function accepts a user-defined functor that is executed for each input element. Using this capability, we tested the `transform` function with various functor sizes to determine how large a functor must be to benefit from GPU acceleration. In Figure 4, we show the speedup from GPU acceler-

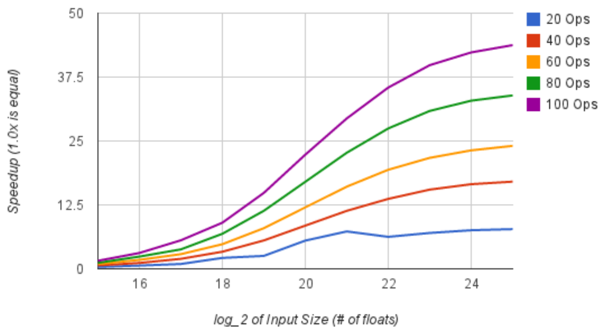

**Figure 4:** *GPU Speedup of std::transform with N floating-point additions on a GTX 570*

ation for functor sizes ranging from 20 to 100 floating-point additions. As the instruction size of the functor increases, the speedup due to GPU acceleration also increases. Therefore, the runtime of `transform` under CPU execution is more dependent on the number of instructions in the user-defined functor than the runtime for GPU execution. We found similar results for experiments involving functors using integer addition and floating-point multiplication as well.

We note that, to accurately choose which device to execute `transform` on, the programmer or compiler must provide the approximate number of functor instructions as an argument to the `transform` function. This is required because C++ does not provide a reliable mechanism for determining the number of instructions in a functor at runtime. In some cases, the number of instructions in a function can be determined by iterating through memory until a `RET` opcode is found, but this is not possible for functors because the address-of operator is not valid for functors.

**Device Data Transfer** We calculated the device data transfer overheads for `gpu_transform` with a functor of 100 floating-point multiplies. On the GTX 570, the data transfer overheads for input sizes above 1 million elements ranged from 50% to 81% of the total runtime. Thus, for large inputs, a system-on-a-chip design could yield even greater speedups due to the elimination of costly data transfer.

**Other Algorithms** Though we observed large performance increases for some algorithms, other functions, such as `find`, `set_union`, and `min_element`, did not benefit from GPU acceleration. These functions execute a small number of instructions per data element, which leaves less room to amortize the cost of copying data to and from the GPU and starting kernels. For these functions, another parallel approach, such as vectorization [Mytkowicz and Marron 2011], may provide performance benefits in place of GPU acceleration.

## 4 Related Work

Many prior efforts have explored GPU acceleration for specific algorithms, often yielding large performance increases. Bakkum and Skadron implemented a GPU accelerated version of `SQLLite` that required no application code modifications [Bakkum and Skadron 2010]. On an NVIDIA Tesla GPU, the accelerated version of `SQLLite` achieved speedups between 20x and 70x.

MCSTL provides a multi-core parallel implementation of some algorithms using OpenMP and achieves a speedup of up to 21x for sorting on an 8-core machine [Putze et al. 2007]. A vectorized version of the STL has also been developed [Mytkowicz and Marron 2011]. Vectorized STL algorithms can provide speedups of up to 20x for functions that execute a small number of instructions per element, such as `binary_search`.

## 5 Future Work

For this project, two CPU and GPU combinations were evaluated to determine heuristics for CPU and GPU execution. Due to the small sample size, the heuristics may overfit the results of the experimental data and not perfectly apply to other systems. Further experimentation is required to ensure that the heuristics correctly predict which device to use for as many systems as possible.

At the time of this paper, the LLVM version of the C++ STL, *libcxx*, only supports Mac OS X. The linux systems evaluated provide the *libstdc++* STL implementation, which is generally coupled to a specific version of *gcc*. Reimplementing GPU acceleration for each version of *libstdc++* is tedious, but once *libcxx* supports Linux, *libcxxgpu* could be implemented on top of *libcxx* for a cross-platform GPU accelerated implementation of the STL.

Finally, certain algorithms, such as `find`, cannot realize a speedup from GPU acceleration due to the need to copy data to the GPU before each function call. To enable speedups on these low instruction algorithms, it may be possible to provide constructs to combine multiple algorithms into a single copy and multiple GPU function calls. However, this project was focused on acceleration without code changes, which would be required for this type of mechanism. Another approach to accelerating low instruction algorithms might be to use vectorization for low instruction count algorithms and GPU acceleration for high instruction count algorithms.

## 6 Conclusion

Using *libcxxgpu*, C++ programs that use certain STL algorithms can be automatically accelerated through GPU execution. Through experimentation, device data transfer and kernel startup costs were identified as the major bottleneck for GPU execution and the number of instructions per data element was identified as the major bottleneck for CPU execution. In the future, a combination of vectorization and GPU acceleration may provide a high-performance, parallel STL for C++ applications.

## References

BAKKUM, P., AND SKADRON, K. 2010. Accelerating sql database operations on a gpu with cuda. In *GPGPU '10*, ACM, 94–103.

HOBEROCK, J., AND BELL, N. 2012. *Thrust: Code at the Speed of Light*. Thrust. http://code.google.com/p/thrust/.

INTERNATIONAL STANDARD ISO/IEC 14882:2011. 2011. *Programming Languages – C++*. International Organization for Standards.

MYTKOWICZ, T., AND MARRON, M. 2011. Single-core performance is still relevant in the multi-core era. In *In the PLDI Fun Thoughts and Ideas Session*, ACM.

PUTZE, F., SANDERS, P., AND SINGLER, J. 2007. Mcstl: The multi-core standard template library. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, 144–145.