CS59200: Final Project Report

Vidush Singhal and Vedant Paranjape

Section 1: Summary

OpenMP [1] is a language agnostic application programming inferface (API). It allows the programmer to annotate certain regions in the program using pragmas. These pragmas are directives that inform the compiler to generate certain type of code. For example, a ***parallel for*** pragma allows the programmer to parallelize trivial loops without dependencies.

As an example consider listing 1 that can execute the inner computation in parallel across 4 threads on a CPU.

Listing 1: OpenMP CPU Example

```
#include <omp.h>
#include <stdio.h>

#pragma omp parallel for num_threads(4)
for (i = 1; i < SIZE; i++) {
    a[i] = b[i] + c[i];
}
```

Listing 2: OpenMP Offload Example

```
#include <omp.h>
#include <stdio.h>

#pragma omp target teams num_teams(3)
for (i = 0; i < 12; i++) {
    a[i] = b[i] + c[i];
}
```

Each thread compute the ith element of the array a, using, the ith element of arrays b and c. This reduces the burden on the programmer by not worrying about target dependent instructions for parallelization for instance, the OpenMp runtime can handle efficient parallelism.

The goal with OpenMP offload is to ”offload”, that is, send computation to the device from the host machine. The programmer can use directives to annotate certain parts of the program that are computationally intensive. These parts can benefit from execution on the GPU device rather than the host machine. This can allow the user to speed up their application.

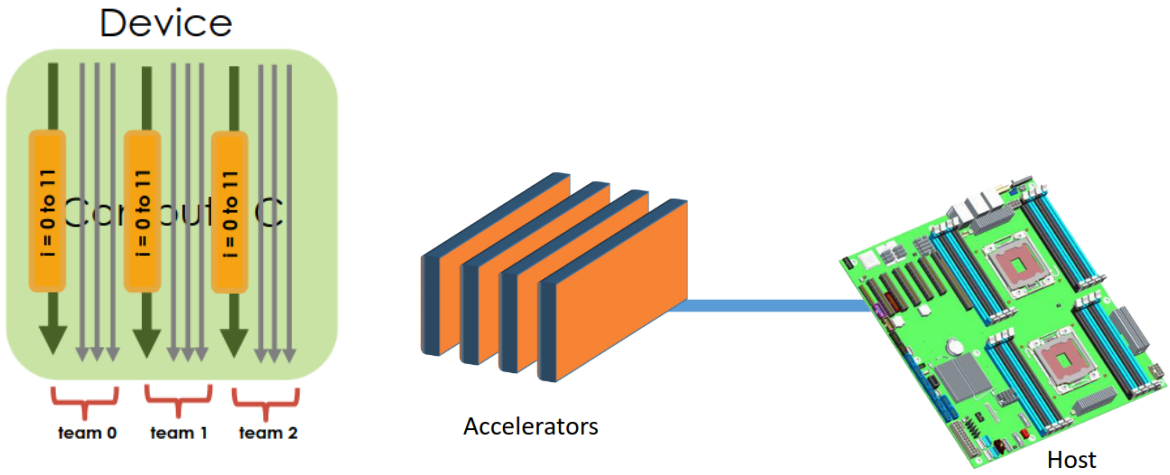
A compiler pragma like ***pragma omp target*** [2, 3] allows the programmer to automatically send the computation to devies like the GPU. The compiler with automatically generate code that will copy the data from the host to the device and vice-versa once the computation finishes. The device and the host have both have a copy of the same data and the host would not be able to access that date when it is executing on the device.

Additional pragmas like ***teams*** allows to form groups of threads for mapping the execution on GPUs, similar to warps in cuda.

For example, listing 2 shows an example where the loop executes in teams of 3 with each team having 4 threads. The visualization for such a case is shown in figure 1b

The following is a description of all the offload related OpenMp pragmas and their description.

- **omp declare target:** Specifies functions and variables that are created or mapped to a device.



(a) OpenMP teams visualization

(b) OpenMP Offload visualization

- **omp declare variant:** Identifies a variant of a base procedure and specifies the context in which this variant is used.
- **omp dispatch:** Determines if a procedure variant is called for a given procedure.
- **omp distribute:** Specifies that the iterations of one or more loops should be distributed among the initial threads of all thread teams in a league.
- **omp interop:** Identifies a foreign runtime context and identifies runtime characteristics of that context, enabling interoperability with it.
- **omp requires:** Lists the features that an implementation must support so that the program compiles and runs correctly.
- **omp target enter data:** Specifies that variables are mapped to a device data environment.
- **omp target exit data:** Specifies that variables are unmapped from a device data environment.
- **omp teams:** Creates a league of thread teams inside a target region to execute a structured block in the initial thread of each team.

In this project, we want to see how efficient offloading C++17 parallel STL functions would be on the GPU [4]. STL [5] provides various algorithms and we can can self implement them for testing performance then testing. We can also try to then automatically do it using a compiler like LLVM.

Section 2: Proof of concept and intended results

To establish a proof of concept, we decided to hand implement some of the standard algorithms and compare their performance in different configurations. For each example, we compare the performance of the GPU offloaded version and CPU OpenMP version and the CPU fully serial version. For this project we compared the performance of the following algorithms which we specifically handwrote using OpenMp pragmas.

- **copy**: This program is similar to STL copy such that it takes an input array and copied it to an output location, i.e, another array.

- **fill**: This program is similar to the STL fill function, such that it takes an input array and the value that should be used to fill the array. This means that every element in the array will be set to this value.
- **transform_reduce**: This program takes an array and a unary function to be applied to each array. It then does function application using the specified function for each array element and also reduces all the values of the array into a single value to be returned
- **for_each**: This program takes an array and then iterates over each element of the array and applies that function onto each element of the array.
- **any_of**: This program takes an array and then iterates over each element of the array to check if any one of the element matches the result of the function. If it does, it will set the result variable to true.
- **count**: This program takes an array and then counts each element of the array only if it is equal to the result of the function. It then returns the count as the result variable.

Section 3: Results

We were able to make LLVM generate code for GPU using OpenMP pragmas. The original plan was to use libc++ source to automatically transform to GPU code, but due to some upstream bugs in libc++ it was not possible to due in this time frame. So instead we implemented a custom version of libc++ using OpenMP pragmas. We wrote simple benchmarks which used these OpenMP pragma enabled libc++ functions to benchmark the efficieny of our implementation. Using the LLVM Compiler with Nvidia (nvptx) codegen backend enabled, we were able to automatically generate OpenMP code for Nvidia GPUs. For validation purposes we were also able to get it running on AMD GPUs. Our custom libc++ implementation supports std::copy, std::fill, std::transform_reduce, std::for_each, std::any_of and std::count. These implementation use OpenMP pragmas to parallelize code on GPUs and CPUs alike. Our implementation supports offloading to Nvidia, AMD GPUs and CPUs.

Section 3.1: Automatically generating GPU Code using LLVM

As mentioned in the previous section we were able to implement few algorithms, in this section we will explain in depth about std::count() function which we implemented. All the other algorithms are similar in terms of GPU code generation, but for the brevity of this report we stick to std::count. All the other benchmarks are uploaded to the github repository.

In the code listing 2, we show our implementation of std::count which has been modified to have a tiled loop and use OpenMP pragmas to speedup the code. The pragmas are device agnostic, and it can be deployed to CPU or GPUs alike. In the code listing 7 we use the std::count function defined below to implement a small benchmark to compare the performance of our implementation across CPUs and GPUs. We use the following commands to compile this for Nvidia GPU and AMD CPU. Clang has an option (-fopenmp-targets=) to specify which target to compile this code for, so if we specify Nvidia GPU it will generate code for it. The compile commands we used are shown below in the code listing. As mentioned before, we use -fopenmp-targets to specify nvptx64-nvidia-cuda to target Nvidia GPUs. LLVM is able to support a wide variety of targets as LLVM supports many compiler code gen backends like Nvidia's PTX, AMD's HIP, etc.

```
1 # Compile command for GPU
2 clang++ -O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda count_offload.cpp -o
      count_offload
3
4 # Compile command for CPU
5 clang++ -O3 -fopenmp count_offload.cpp -o count_omp
```

```
void count(long *array, int size, int *result, long (*func_ptr)(long)) {
  #pragma omp target teams distribute parallel for map(tofrom: array[0:size])
  for(int i = 0; i < (size/5000); i++) {
    for(int j = 0; j < 5000; j++) {
      if (array[i*5000 + j] == func_ptr(array[i*5000 + j])) {
        *result += *result + 1;
      }
    }
  }
}
```

Figure 2: Figure showing implementation of std::count using OpenMP pragmas.

In the implementation of std::count, we tiled the for loop so that it can better perform on GPUs. It checks if each array element is equal to the result of the function passed, and if it is, it will increment the result. We added omp target distribute pragmas at the start of the for loop so that OpenMP is able to parallelize it. The benchmark simply defines a large array, fills it with dummy data and then invokes std::count with this data. We see significant performance gains when run on a GPU vs on a CPU.

Section 3.2: Offloading OpenMP code to GPUs using OpenMPOffload

To offload OpenMP code to GPUs, we need to build LLVM to support generating code for NVPTX. This is trivial to do, as LLVM support NVPTX as a code generation backend. The CMake command for configuring LLVM is depicted in code listing 3. In that we enable NVPTX backend by adding it to **DLLVM_TARGETS_TO_BUILD="NVPTX;X86"**.

We then use LLVM to compile our benchmark codes using the following commands:

```
# Compile command for GPU
clang++ -O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda count_offload.cpp -o
    count_offload
```

Section 3.3: Analysing Nvidia PTX code generated using LLVM OpenMP Offload

In this section we analyze the generated code to understand how LLVM generates code for the NVPTX backend and how it offloads the CUDA kernels to the GPU. We add the -S -emit-llvm command to dump the LLVM IR for the generated code. Similarly we also add -save-temps to dump all the intermediate files generated during compilation. The files created during compilation are shown in Figure 4. As you can see, there is a *.cubin file that is generated by LLVM, and there are other files as well (count_offload-openmp-nvptx64-nvidia-cuda.o) which contain the actual NVPTX assmebly generated by LLVM. We have added a reduced snippet of the NVPTX assembly in code listing 9. The code has PTX assembly generated by the LLVM Compiler.

Now we explain the exact flow of compilation followed by LLVM to generate the PTX binary. We compile the std::count benchmark using the command given below. As a result the LLVM Compiler generates a CUDA kernel for the functions that use OpenMP pragma, so it internally converts the pragmas to CUDA kernel parameters. It will generate a wrapper function which will setup the CUDA kernel parameters like size, warps, threads, etc and then calls the actual implementation of the CUDA kernel. The wrapper function can be seen in code listing 6, on Line 7, it alloca's a target kernel object and following that it sets the target kernel parameters. On Line 24, there is a call to __tgt_target_kernel() which is a generic OpenMP function to

```
    cmake -G Ninja -DCMAKE_BUILD_TYPE=Release \
        -DCLANG_INCLUDE_TESTS=ON \
        -DCMAKE_INSTALL_PREFIX="/local/scratch/a/paranjav/llvm-project/install-
            release" \
        -DCMAKE_CXX_STANDARD=17 \
        -DCMAKE_C_COMPILER=gcc \
        -DCMAKE_CXX_COMPILER=g++ \
        -DLIBOMPTARGET_ENABLE_DEBUG=ON \
        -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
        -DCMAKE_CXX_LINK_FLAGS="-Wl,-rpath␣,$LD_LIBRARY_PATH" \
        -DLLVM_ENABLE_ASSERTIONS=ON \
        -DLLVM_ENABLE_LIBCXX=ON \
        -DLIBC_GPU_BUILD=OFF \
        -DLLVM_LIBC_FULL_BUILD=ON \
        -DOPENMP_ENABLE_LIBOMPTARGET=ON \
        -DLIBOMPTARGET_NVPTX_ENABLE_BCLIB=true \
        -DLLVM_TARGETS_TO_BUILD="NVPTX;X86" \
        -DLLVM_ENABLE_PROJECTS="clang;lld" \
        -DLLVM_ENABLE_RUNTIMES="openmp;offload;libcxxabi;libcxx;libunwind" \
        -DLIBCXX_ENABLE_THREADS=ON \
        /local/scratch/a/paranjav/llvm-project/llvm
```

Figure 3: CMake command

call a target kernel. The target kernel can be a GPU kernel or a CPU kernel, it entirely depends on the target specified. At this stage LLVM generates platform agnostic code.

The LLVM IR for the function called in __tgt_target_kernel is in code listing 8. The LLVM IR is of a nested loop, and thus it matches the std::count implementation. On Line 31, it calls _kmpc_fork_call which calls the NVPTX implementation of the function. The assembly for this function is listed in code listing 9, as you can see on Line 24, the actual PTX assembly implementation of the count function starts. This NVPTX assembly dump was generated from the *.o file using cuobjdump which is an disassembly utility provided by CUDA. The said assembly was generated by LLVM NVPTX backend.

To reiterate, the steps for compilation are as follows:

- Compile OpenMP code to OpenMP kernel LLVM IR.
- Compile the OpenMP kernel LLVM IR to the target backend specified during compilation.
- Generate CUDA wrapper code for the OpenMP LLVM IR.
- Compile the OpenMP kernel to NVPTX backend using LLVM Compiler.
- Link the files and generate a unified assembly.

```
clang++ -O3 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda count\_offload.cpp -o
    count\_offload -S -emit-llvm
```

Section 3.4: Running the binary generated by LLVM

Now that LLVM has generated offload binary for our std::count benchmark, we can simply run it and it gets offloaded to the Nvidia GPU. The output of the program is shown below.

```
paranjav@io:~/offload_stl_examples/llvm-ir$ ls -ltr
total 4656
-rw-r--r-- 1 paranjav ecnuser    1355 Dec 14 22:56 count_offload.cpp
-rw-r--r-- 1 paranjav ecnuser  987815 Dec 14 22:56 count_offload-host-x86_64-unknown-linux-gnu.ii
-rw-r--r-- 1 paranjav ecnuser   26796 Dec 14 22:56 count_offload-host-x86_64-unknown-linux-gnu.bc
-rw-r--r-- 1 paranjav ecnuser 1069147 Dec 14 22:56 count_offload-openmp-nvptx64-nvidia-cuda.ii
-rw-r--r-- 1 paranjav ecnuser  267212 Dec 14 22:56 count_offload-openmp-nvptx64-nvidia-cuda.bc
-rw-r--r-- 1 paranjav ecnuser   36074 Dec 14 22:56 count_offload-openmp-nvptx64-nvidia-cuda.s
-rw-r--r-- 1 paranjav ecnuser   20328 Dec 14 22:56 count_offload-openmp-nvptx64-nvidia-cuda.o
-rw-r--r-- 1 paranjav ecnuser   20472 Dec 14 22:56 count_offload-openmp-x86_64-unknown-linux-gnu.out
-rw-r--r-- 1 paranjav ecnuser   97621 Dec 14 22:56 count_offload-host-x86_64-unknown-linux-gnu.s
-rw-r--r-- 1 paranjav ecnuser   30528 Dec 14 22:56 count_offload-host-x86_64-unknown-linux-gnu.o
-rw-r--r-- 1 paranjav ecnuser   20328 Dec 14 22:56 count_offload-host-x86_64-unknown-linux-gnu-nvptx64-nvidia-cuda-sm_89.o
-rw-r--r-- 1 paranjav ecnuser   19720 Dec 14 22:56 count_offload.nvptx64.sm_89.img
-rw-r--r-- 1 paranjav ecnuser   20328 Dec 14 22:56 count_offload-host-x86_64-unknown-linux-gnu-nvptx64-nvidia-cuda-sm_89.cubin
-rw-r--r-- 1 paranjav ecnuser   22188 Dec 14 22:56 count_offload.openmp.image.wrapper.bc
-rw-r--r-- 1 paranjav ecnuser   22280 Dec 14 22:56 count_offload.openmp.image.wrapper.o
-rwxr-xr-x 1 paranjav ecnuser   39752 Dec 14 22:57 count_offload
-rw-r--r-- 1 paranjav ecnuser  699427 Dec 14 22:58 count_offload.asm
-rw-r--r-- 1 paranjav ecnuser 1166144 Dec 14 23:02 count_offload-openmp-nvptx64-nvidia-cuda.ll
-rw-r--r-- 1 paranjav ecnuser  169957 Dec 14 23:06 cuda-kernel.asm
paranjav@io:~/offload_stl_examples/llvm-ir$
```

Figure 4: OpenMP Offload file dump

```
./count_offload
Execution time: 2.00091 seconds
Count: 0
```

We ran all our benchmarks on Nvidia GPU, CPU OpenMP offload and vanilla CPU runtime. The results are explained in the next section.

Section 3.5: Issues with upstream LLVM libc++

The initial plan was to offload LLVM's libc++ implementation to Nvidia GPUs. We spent several days trying to get it working, but due to some incorrect CMake configurations, LLVM always generated code for Intel TBB. It was unable to change the backend to say Nvidia for the libc++ implementation. We have filed a upstream issue for the same as addressed in the Future works section. We will port our implementation to LLVM's libc++ as soon as the upstream issue is resolved. But we believe that our implementation gives a proof of concept that standard C++ algorithms can be effortlessly offloaded to GPUs.

Section 4: Evaluation

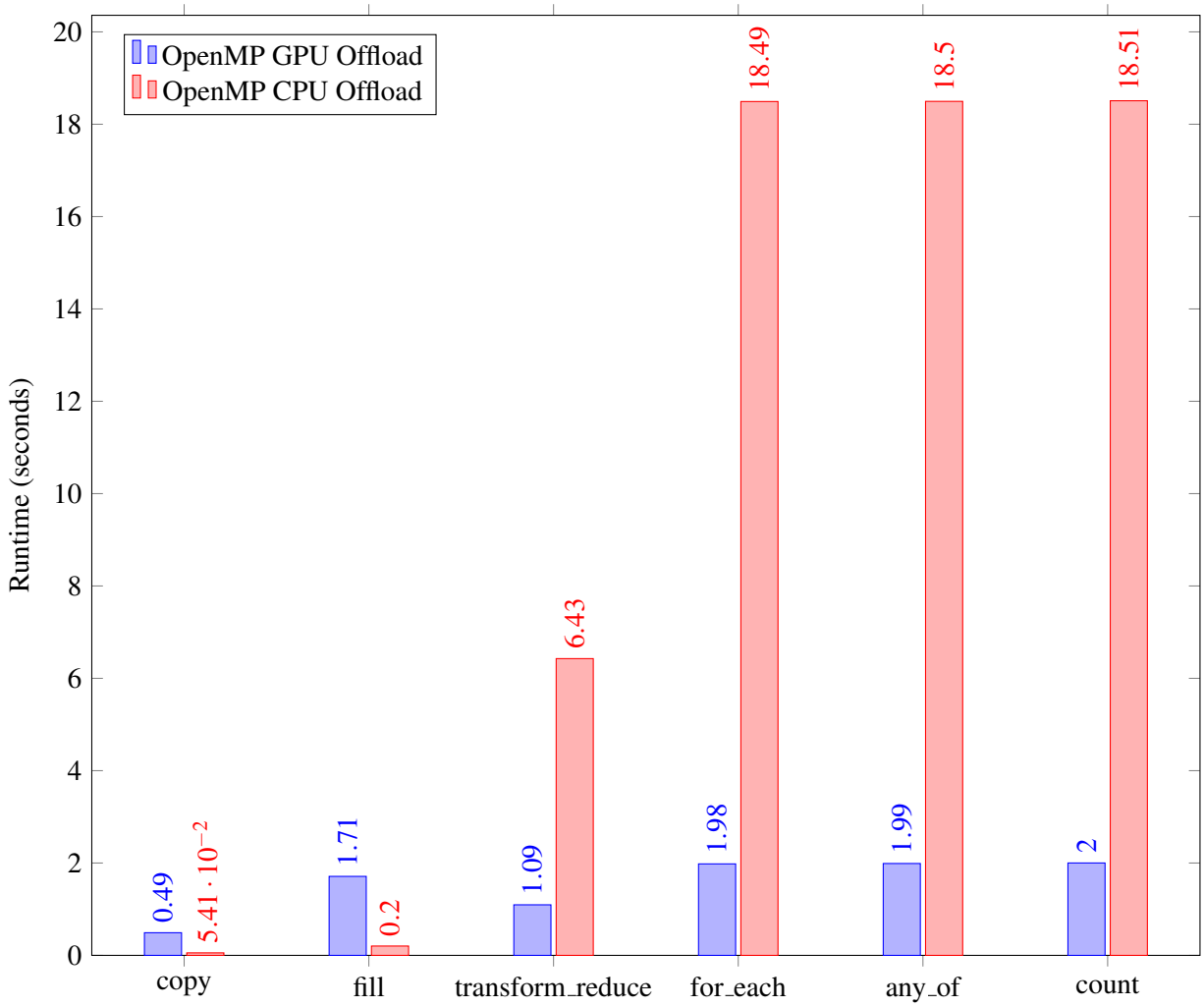
We performed our test on an AMD Ryzen Threadripper PRO 5965WX 24-Cores Processor with an NVIDIA GeForce RTX 4070 Ti GPU. The L1D and L1I cache size was 768K, the L2 cache size was 12M and the L3 cache size was 128M. The CPU max clock speed was 7.021 GHz.

For offloading support, we build LLVM using the CMake command shown in listing 3 command. Table 2 shows the results of our experiments. We now provide explanation of the results and why the offloading is slow for copy and fill but faster for transform_reduce and for_each. We also show the compile commands that we used for each program in listings 5. The programs used can be found in the linked public github repository https://github.com/vidsinghal/offload_stl_examples.

Offloading involves a significant overhead. The memory for the input and output arrays must be allocated on the device and the output arrays must be copied back from the device to the host. If the arrays are very large this may take significant time as well. The cost of the overhead is reduced is the computation on the arrays is computationally intensive. If the computation is large and parallel, the significant number of threads on GPUs can reduce the time it takes to do the computation thus reducing the overall time of the program. On the other hand, if the computation is lightweight, the overhead to copy the arguments will dominate thus not providing enough speedup.

This is exactly what we see in our results shown in Table 2 and Figure 1. Copy and Fill are lightweight traversals over the array. The computational intensity is low thus copying is an overkill. The offload versions are slower than the CPU OpenMP versions for this reason. Even the serial CPU version is slightly faster than the offloaded versions. However, for transform, any_of, count and foreach, since the computational intensity is high, we see significant speedups with the offloaded version than the OpenMP CPU and serial CPU version.

Table 1: Bar plot for Runtime of benchmarks



Section 4: Conclusion

We have shown through some simple hand crafted examples that offloading some STL function can be beneficial as it may speed up the computation by executing them on the GPU. However, that's only the case if the function is computationally intensive. For lightweight functions, there is a slowdown from offloading that function. We will work on the cost model that can infer which functions may be beneficial to offload. For the future work, we plan to offload the STL function automatically using the LLVM compiler. We are in process to do so and are dealing with compiler bugs atm. With this ability we can potentially speed up the user's application significantly automatically using OpenMP offload.

Table 2: Performance comparison of different handwritten STL functions.

Benchmark	OpenMP Offload	OpenMP CPU	CPU
copy	0.490246 (s)	0.0541481 (s)	0.256001 (s)
fill	1.71053 (s)	0.202087 (s)	1.2373 (s)
transform_reduce	1.09482 (s)	6.42661 (s)	148.325 (s)
for_each	1.9809 (s)	18.4916 (s)	600.848 (s)
any_of	1.98938 (s)	18.4952 (s)	620.799 (s)
count	2.00091 (s)	18.5082 (s)	600.873 (s)

```
[singhal2@linux-5:algorithm_tests]$ make all
    clang++ -O3 -fopenmp --offload-arch=native \
             for_each_offload.cpp -o for_each_offload
    clang++ -O3 -fopenmp --offload-arch=native \
             transform_reduce.cpp -o transform_reduce_offload
    clang++ -O3 -fopenmp --offload-arch=native \
             fill_offload.cpp -o fill_offload
    clang++ -O3 -fopenmp --offload-arch=native \
             copy_offload.cpp -o copy_offload
    clang++ -O3 -fopenmp for_each_offload.cpp -o for_each_omp
    clang++ -O3 -fopenmp transform_reduce.cpp \
             -o transform_reduce_omp
    clang++ -O3 -fopenmp fill_offload.cpp -o fill_omp
    clang++ -O3 -fopenmp copy_offload.cpp -o copy_omp
    clang++ -O3 for_each_offload.cpp -o for_each_cpu
    clang++ -O3 transform_reduce.cpp -o transform_reduce_cpu
    clang++ -O3 fill_offload.cpp -o fill_cpu
    clang++ -O3 copy_offload.cpp -o copy_cpu
```

Figure 5: Figure showing compile commands that were used.

Section 5: Future Work

Now that we have proved that offloading some parallel STL algorithms to GPUs using OpenMP offloading can be beneficial, we will implement code to do this automatically. For this we will use LLVM, we will parse the C++ STL functions and write a custom OpenMP backend for these functions such that they can be automatically offloaded to the GPU. There is some ongoing work on this in the form of this PR: https://github.com/llvm/llvm-project/pull/116869. We tried to build the PR with the OpenMP backend for PSTL. However, it seems like there is a bug with their implementation and the build defaults to the TBB backend. As it can be seen in the PR, we posted a comment with the error and the possible explanation of the error and we are in progress to fix these errors. With the help of the contributors we should be able to get the initial proof of concept compiler working. However, as our results show, this is only beneficial for computationally intensive functions. Thus, we would probably add a static cost estimation of the function and only generate offloaded code for parallel STL algorithms that are computationally intensive, that is, costly according to our cost model.

References

[1] Accessed: 2024-12-11. URL: https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf.

[2] Accessed: 2024-12-11. URL: https://www.olcf.ornl.gov/wp-content/uploads/2021/08/ITOpenMP_Day1.pdf.

[3] Accessed: 2024-12-11. URL: https://discourse.llvm.org/t/gsoc-2024-offloading-libcxx/77238.

[4] Accessed: 2024-12-11. URL: https://llvm.org/OpenProjects.html#offload-libcxx.

[5] Accessed: 2024-12-11. URL: https://www.geeksforgeeks.org/the-c-standard-template-library-stl/.

Appendix A: Code Listings

```
; Function Attrs: nounwind uwtable
define dso_local void @_Z5countPliPiPFllE(ptr noundef %array, i32 noundef %size, ptr noundef %result, ptr noundef %func_ptr) local_unnamed_addr #3 {
entry:
  %.offload_baseptrs = alloca [4 x ptr], align 8
  %.offload_ptrs = alloca [4 x ptr], align 8
  %.offload_sizes = alloca [4 x i64], align 8
  %kernel_args = alloca %struct.__tgt_kernel_arguments, align 8
  %1 = getelementptr inbounds i8, ptr %.offload_baseptrs, i64 8
  ...
  ...
  store i32 3, ptr %kernel_args, align 8
  %9 = getelementptr inbounds nuw i8, ptr %kernel_args, i64 4
  store ptr %.offload_ptrs, ptr %11, align 8
  ...
  ...
  %16 = getelementptr inbounds nuw i8, ptr %kernel_args, i64 64
  call void @llvm.memset.p0.i64(ptr noundef nonnull align 8 dereferenceable(36) %16, i8 0, i64 36, i1 false)
  %17 = call i32 @__tgt_target_kernel(ptr nonnull @3, i64 -1, i32 0, i32 0, ptr nonnull @.__omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55.region_id, ptr nonnull %kernel_args)
  %.not = icmp eq i32 %17, 0
  br i1 %.not, label %omp_offload.cont, label %omp_offload.failed

omp_offload.failed:                               ; preds = %entry
  call void (ptr, i32, ptr, ...) @__kmpc_fork_teams(ptr nonnull @3, i32 4, ptr nonnull @__omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55.omp_outlined, i64 %size.casted.sroa.0.0.insert.ext, ptr %array, ptr %func_ptr, ptr %result)
  br label %omp_offload.cont

omp_offload.cont:                                 ; preds = %omp_offload.failed, %entry
  ret void
}
```

Figure 6: Figure showing LLVM IR of the OpenMP offload function

```cpp
#include <stdio.h>
#include <omp.h>
#include <chrono>
#include <iostream>
#include <cassert>

void count(long *array, int size, int *result, long (*operation)(long));
long complex(long i);
#pragma omp declare target indirect to(complex)

int main() {
  int size = 2e8;
  long *array = (long*) malloc(sizeof(long) * size);
  int val = 10;

  #pragma omp parallel for
  for(int i = 0; i < size; i++){
       array[i] = val;
  }

  long (*func_ptr)(long) = complex;
  int result = 0;
  auto start = std::chrono::high_resolution_clock::now();

  count(array, size, &result, func_ptr);

  auto end = std::chrono::high_resolution_clock::now();

  std::chrono::duration<double> duration = end - start;
  std::cout << "Execution time: " << duration.count() << " seconds" << std::
      endl;

  std::cout << "Count: " << result << std::endl;
}

long complex(long i){
 long j = i;
 j = j * 1;
 j = j + 1;
 j = j * 10;
 j = j / 4;
 j = j + 1;

 for(int k = 1; k < 10000; k++){
   j = j * k;
 }

  return j;
}
```

Figure 7: Figure showing benchmark utilizing std::count using OpenMP pragmas.

```
; Function Attrs: alwaysinline norecurse nounwind uwtable
define internal void @__omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55.omp_outlined(ptr noalias nocapture noundef readonly %.global_tid., ptr noalias nocapture readnone %.bound_tid., i64 noundef %size, ptr noundef %array, ptr noundef %func_ptr, ptr noundef %result) #4 {
entry:
  %.omp.comb.lb = alloca i32, align 4
  %.omp.comb.ub = alloca i32, align 4
  %.omp.stride = alloca i32, align 4
  %.omp.is_last = alloca i32, align 4
  %size.addr.sroa.0.0.extract.trunc = trunc i64 %size to i32
  %cmp = icmp sgt i32 %size.addr.sroa.0.0.extract.trunc, 4999
  br i1 %cmp, label %omp.precond.then, label %omp.precond.end

omp.precond.then:                                 ; preds = %entry
  %div7 = udiv i32 %size.addr.sroa.0.0.extract.trunc, 5000
  %sub3 = add nsw i32 %div7, -1
  call void @llvm.lifetime.start.p0(i64 4, ptr nonnull %.omp.comb.lb) #6
  ...
  ...
  %0 = load i32, ptr %.global_tid., align 4, !tbaa !9
  call void @__kmpc_for_static_init_4(ptr nonnull @1, i32 %0, i32 92, ptr nonnull %.omp.is_last, ptr nonnull %.omp.comb.lb, ptr nonnull %.omp.comb.ub, ptr nonnull %.omp.stride, i32 1, i32 1)
  ...
  br i1 %cmp6.not8, label %omp.loop.exit, label %omp.inner.for.body.lr.ph

omp.inner.for.body.lr.ph:                         ; preds = %omp.precond.then
  %size.casted.sroa.0.0.insert.ext = and i64 %size, 2147483647
  br label %omp.inner.for.body

omp.inner.for.body:                               ; preds = %omp.inner.for.body.lr.ph, %omp.inner.for.body
  %3 = phi i32 [ %cond, %omp.inner.for.body.lr.ph ], [ %8, %omp.inner.for.body ]
  %.omp.iv.09 = phi i32 [ %2, %omp.inner.for.body.lr.ph ], [ %add, %omp.inner.for.body ]
  ...
  call void (ptr, i32, ptr, ...) @__kmpc_fork_call(ptr nonnull @3, i32 6, ptr nonnull @__omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55.omp_outlined.omp_outlined, i64 %5, i64 %6, i64 %size.casted.sroa.0.0.insert.ext, ptr %array, ptr %func_ptr, ptr %result)
  %7 = load i32, ptr %.omp.stride, align 4, !tbaa !9
  %add = add nsw i32 %7, %.omp.iv.09
  ...

omp.loop.exit:                                    ; preds = %omp.inner.for.body, %omp.precond.then
  call void @__kmpc_for_static_fini(ptr nonnull @1, i32 %0)
  br label %omp.precond.end

  ...
  ...
}
```

Figure 8: Figure showing LLVM IR of the OpenMP for loop which does the computation

```
//
// Generated by LLVM NVPTX Back-End
//

.version 8.2
.target sm_89
.address_size 64
...
...
;
.weak .global .align 4 .u32 __omp_rtl_assume_teams_oversubscription;
.weak .global .align 4 .u32 __omp_rtl_assume_threads_oversubscription;
.visible .global .align 8 .u64 __omp_offloading_fd04_8135d223_complex_18 =
    _Z7complexl;
...
(__unnamed_2), generic(
    __omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55_dynamic_environment)
    };
// _ZN4ompx5state9TeamStateE has been demoted
// _ZN4ompx5state12ThreadStatesE has been demoted
.weak .global .align 8 .b8 __omp_rtl_device_memory_pool[16];
.weak .global .align 8 .b8 __omp_rtl_device_memory_pool_tracker[32];
.weak .global .align 4 .u32 __omp_rtl_debug_kind;
.weak .global .align 4 .u32 __omp_rtl_assume_no_thread_state;
...
...
// .weak    __omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55 // -- Begin
    function __omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55
.weak .entry __omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55(
    .param .u64 __omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55_param_0,
    .param .u64 __omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55_param_1,
    .param .u64 __omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55_param_2,
    .param .u64 __omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55_param_3,
)
.maxntid 128, 1, 1                          //
    @__omp_offloading_fd04_8135d223__Z5countPliPiPFllE_l55
{
    .reg .pred      %p<70>;
    .reg .b32       %r<144>;
...
...
    setp.ne.s32     %p1, %r1, 0;
    @%p1 bra        $L__BB1_2;
// %bb.1:                                   // %if.then.i.i1
    mov.b32         %r46, 0;
    st.shared.v2.u32        [_ZN4ompx5state9TeamStateE], {%r46, %r46};
    st.shared.v2.u32        [_ZN4ompx5state9TeamStateE+8], {%r46, %r46};
    mov.b32         %r47, 1;
...
...
$L__BB1_8:                                  // %if.then.i.i.i.i.i9
    mov.u64         %rd217, %rd5;
    bra.uni         $L__BB1_9;
$L__BB1_12:                                 // %if.then.i.i.i.i.i.i.i19
    mov.u64         %rd218, %rd7;
    bra.uni         $L__BB1_13;
                                            // -- End function
}
```

Figure 9: Figure showing NVPTX for the benchmark generated using LLVM