Optimization of Tele-Immersion Codes

Albert Sidelnik, I-Jui Sung, Wanmin Wu,
María Jesús Garzarán, Wen-mei Hwu, Klara Nahrstedt, David Padua, and Sanjay J. Patel
University of Illinois at Urbana-Champaign
{asideln2,sung10,wwu23,garzaran,w-hwu,klara,padua,sjp}@uiuc.edu

ABSTRACT

As computational power increases, tele-immersive applications are an emerging trend. These applications make extensive demands on computational resources through their heavy use of real-time 3D reconstruction algorithms. Since computer vision developers do not necessarily have parallel programming expertise, it is important to give them the tools and capabilities to naturally express computer vision algorithms, yet retain high efficiency by exploiting modern GPU and large-scale multi-core platforms.

In this paper, we describe our optimization efforts for a tele-immersion application by tuning it for GPU and multi-core platforms. Additionally, we introduce a method to obtain portability, high performance, and increase programmer productivity.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming—*Parallel Programming*; C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms

Program Optimization

Keywords

Tele-immersion codes

1. INTRODUCTION

Tele-immersion [14] is emerging as an important class of applications [1]. Tele-immersion technology allows remote users in geographically distributed areas to collaborate and interact in a shared 3D environment, all in real-time. It gives users the illusion they are in the same physical room, while in reality they could be on different continents. An example rendering of the tele-immersion environment is shown in Figure 1, which depicts the merging of three images from different locations.

Tele-immersive applications make heavy demands on both computational and network resources. The computational demand comes from real-time 3D reconstruction algorithms that are necessary to convert 2D frame images into their respective 3D frame containing the depth information. The network demand comes from the immense amount of data which is sent in multiple streams to and from active participants of the shared virtual environment. In addition, computational cycles are required to render the 3D streams received from the other remote parties. In this paper, we will only focus on the 3D reconstruction algorithms.

There are certain classes of applications that can leverage tele-immersion in its current state [15], but there are other potential users whose needs cannot be met due to tele-immersion's large computational demands [10]. For these tele-immersion applications to be feasible on general-purpose hardware, we need to examine methods of optimizing its most computational costly algorithms. In this paper, we describe an optimization strategy for this class of application that makes use of traditional loop-based compiler transformations. The transformations are applied to 3D reconstruction stereo code that is further transformed into a sequence of high-level data-parallel operations. This last transformation enables portability across parallel architectures including GPUs, multi-cores, and clusters.

Another goal of this project is to study the impact of using data-parallel primitives for productivity, portability, and the ability to optimize automatically. By implementing the tele-immersion application using a collection of high-level data-parallel constructs developed at Illinois, named Hierarchically Tiled Arrays (HTA) [2], one could have a system that encapsulates the parallelism and maps the data parallel operations across different architectures, thus simplifying the task of the domain programmer. Additionally, by expressing data-parallel operations using these constructs, the system would be able to perform autotuning on these data-parallel operations.

This paper is organized as follows: Section 2 gives a high level background and description of tele-immersion and describes the overall flow of the application. Section 3 describes the approach taken in restructuring the 3D reconstruction algorithms into a data-parallel form, and their respective optimizations. Section 4 discusses the results. Finally, Section 5 proposes future work and concludes.

2. TELE-IMMERSION BACKGROUND

Our work is based on TEEVE (Tele-immersive Environments for EVErybody) [14], a joint development project between

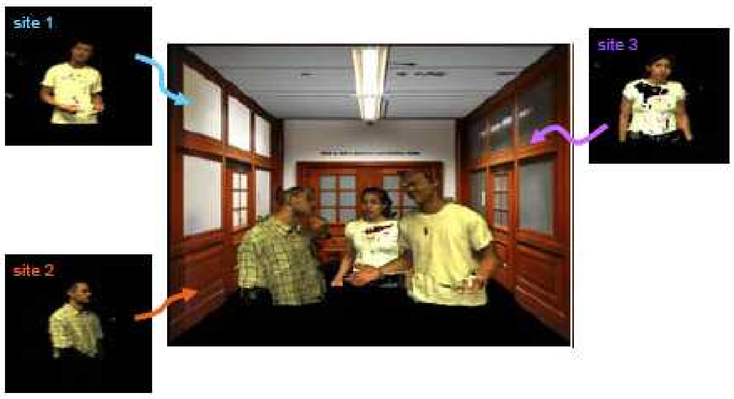


Figure 1: Tele-Immersion Collaboration

the University of Illinois at Urbana-Champaign and the University of California at Berkeley. The goal of TEEVE is to provide a framework for tele-immersive interaction to capture three-dimensional full-body human motion in real time and merge data from different sites to construct a virtual space that immerses all remote participants.

The system consists of three components. The first component is the 3D capturing tier that consists of 3D camera clusters where each cluster reconstructs a 3D video stream. The second component is the network transmission tier for sending of 3D data. The third component is the rendering system tier that maps the received 3D streams into an immersive video. A pictorial view is shown in Figure 2.

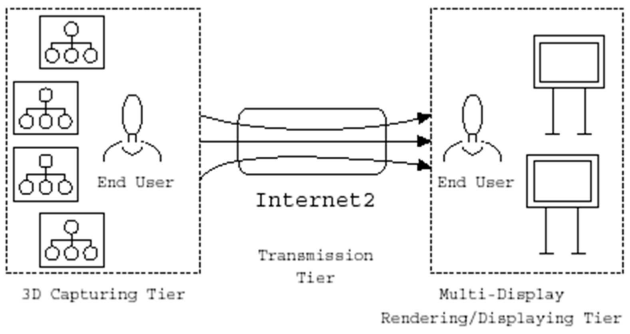


Figure 2: High-Level View of Tele-Immersion Components

For 3D reconstruction, an array of 3D camera clusters is employed to capture the local participants from different viewpoints. Each 3D camera cluster consists of four 2D cameras (three black and white that are used for depth mapping and one color camera that is used for texture mapping), and is hosted by a separate computer that performs real-time 3D reconstruction. Additionally, there is a trigger server used to synchronize all camera clusters so that the image snapshots are taken at the same instant of time.

In this paper we focus on the 3D reconstruction algorithm that runs in the computer connected to the four 2D cameras. The 3D reconstruction (see Figure 3) is done using a trinocular stereo algorithm that computes the 3D information from the images captured by the left, center, and right 2D black and white cameras on a cluster. The captured images are pre-processed by means of invocations to the Intel Image Processing Library for image rectification, image moment computation, background subtraction, and edge extraction. After the pre-processing, a Delaunay Triangulation algorithm [4] and a Modified Normalized Cross-Correlation (MNCC) algorithm [7] are applied to the data stream. The Triangulation and MNCC algorithms are executed concurrently. The Delaunay Triangulation algorithm is applied to the edge points extracted for the center reference image to tessellate it into triangular regions. The MNCC algorithm is applied to the edge points to estimate their depth. For each edge point in the center camera reference image, MNCC computes the correlation between the the corresponding points in the images captured by the left and right cameras at each hypothetical depth level. Later, the correlations are used to determine the most likely depth value of each point by other kernels following the MNCC kernel to re-construct the 3D depth map. A simplified version of MNCC is presented in Algorithm 1. More details about this algorithm can be found in [9]. The output of these two algorithms is used as input to the Compute Homogen algorithm [7]. Lastly, a final post-processing stage that consists of color look-up and image compression is executed. The generated compressed data is packetized and sent over the network to the remote party.

3. OPTIMIZATION APPROACH

This section describes what was done to optimize the kernels involved in the 3D reconstruction algorithm for both multi-cores and GPUs.

To determine the most time consuming parts of the appli-

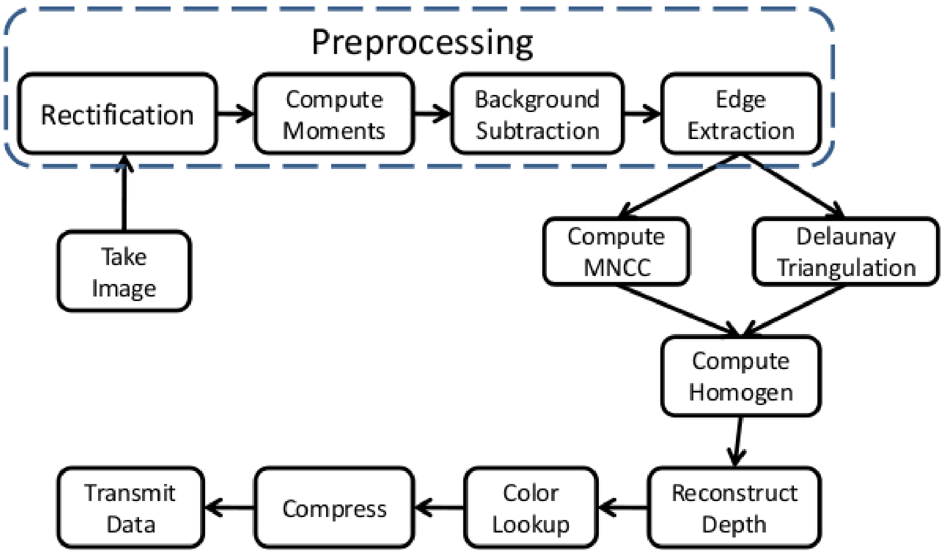


Figure 3: Application Flow of 3D reconstruction

Algorithm 1 Outline of MNCC

```
Require: Images from Left, Center, and Right cameras are acquired as L, C, and R, and pre-processed
Require: Edges in C are extracted by the edge detection algorithm during the pre-process step
Require: Precomputed table C2L($e$, $d$) that maps each *edge* point in C with the corresponding *edge* point in L for each depth level $d$
Require: Precomputed tables C2R($e$, $d$) that maps each *edge* point in C with the corresponding *edge* point in R for each depth level $d$

  $maxcorr(e) = 0, \forall e$
  for all $e$ in C do
    for all $d$ in all possible depth levels do
      $eL \leftarrow C2L(e, d)$
      $eR \leftarrow C2R(e, d)$
      $corr(e, d) \leftarrow$ sum of pixelwise correlations between $eL$, $eR$, and $e$ given depth $d$
      $if(corr(e, d) > maxcorr(e))$ then $maxcorr(e) = corr(e, d)$
    end for
  end for
```

cation, we profiled the code using Intel VTune [12] on an Intel Q9550 Quad Core Penryn that runs at 2.83GHz and has 4GB memory with 2×6 MB of L2 cache. The profile information is shown in Figure 4. The profiled code was written in C and controlled parallelism with the Windows Threads API. As the figure shows, there is one thread per camera during the pre-processing stage. After the pre-processing stage there are three phases separated by barriers. Four threads run in parallel in each phase. During the first phase, one thread runs the Delaunay Triangulation algorithm in parallel with the three threads running the MNCC code. Notice that the MNCC algorithm can be easily decomposed into parallel threads since correlations can be computed independently for each edge point of the image. Compute Homogen runs during the second phase in parallel using four threads, and Depth Reconstruction runs in the third phase, also in parallel. Finally, the post processing step runs sequentially. The profile shows that the most time consuming kernels are MNCC and Compute Homogen. Thus, we focused our attention in these two algorithms. TEEVE uses a serial implementation of the Delaunay Triangulation algorithm. This algorithm runs in parallel with the MNCC kernel. Because of this, Delaunay becomes a bottleneck after MNCC was optimized. The obvious next step in the continuation of this project, is to parallelize the Delaunay Triangulation algorithm [8].

Next, we describe our efforts to translate the two main algorithms, MNCC and Compute Homogen into a data-parallel form suitable for execution in both the GPUs and multi-cores. With this, one could easily map the data-parallel operations using the HTA interface [2]. We also discuss optimization transformations.

3.1 Transformations Applied to MNCC

To transform MNCC into the data-parallel form needed for execution on both the GPU and CPU based multi-core platforms, the code was restructured in several ways. The original code for MNCC is shown in Figure 5-(a). First, traditional data-flow and loop-based compiler transformations [13], such as loop invariant code motion and loop fu-

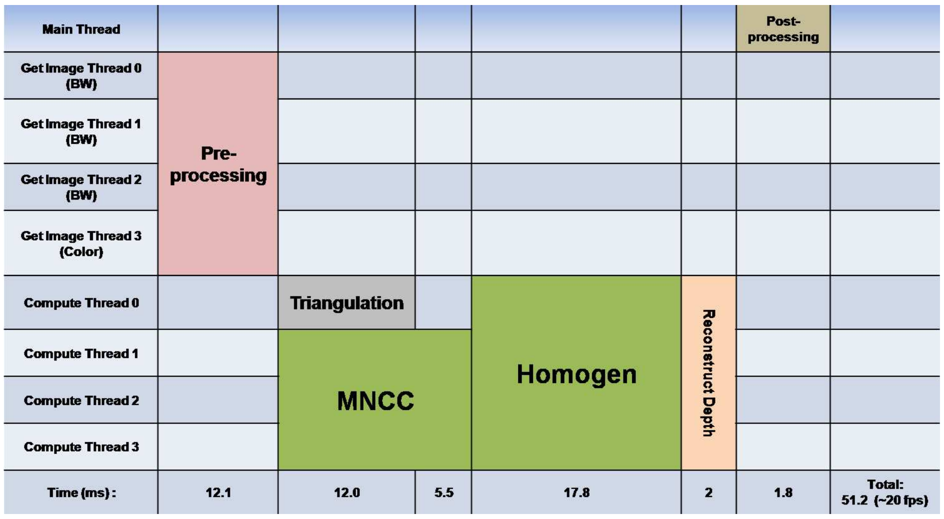


Figure 4: Tele-Immersion Execution Flow

sion, were applied to obtain perfectly nested loops. Second, the computation of the maximum correlation for each edge was extracted and placed in a separate function in order to execute the separate handling of reductions [6]. In addition, array accesses using pointer arithmetic were transformed into arrays subscripts to improve readability. The resulting code is shown in Figure 5-(b).

3.1.1 GPU Specific Optimizations

Next, we report the optimization steps starting from the restructured code in Figure 5-(b). The *find_maximum* function can also be optimized to run on the GPU. However, because a profile execution showed that it consumes an insignificant amount of time, we ran *find_maximum* on the CPU. Because of this, all the optimizations explained in this section only affect the *compute_mncc* function.

1. **Naive GPU Implementation**
   We took the *compute_mncc* in Figure 5-(b) and wrapped it with the essential CUDA specific constructs necessary to execute the kernel on the GPU device. We parallelized the outermost loop. To determine the best block size, we ran the code varying the thread block size and selected the one that minimized the execution time. For the selection of thread block size, we took into account that it needs to be a factor of 32, as this is currently the active warp size in CUDA [11] and multiples of 32 result in better utilization of the GPU processors.

2. **Introduction of multiple dimensions of parallelism**
   Since we modified MNCC so that the two loops were perfectly nested, and there are no dependences between them, we can parallelize both loops. This allows for the parallel execution of the kernel on an element by element granularity across all threads. In order to have multiple dimensions of parallelism, we went from a 1D to a 2D GPU thread block [11]. Just as in the naive implementation, we empirically searched for the best 2D thread block size. Our criteria for selecting thread block sizes was that they needed to both be factors of 32, due to the active warp size, and of 40, as this is the constant iteration bound for the inner loop (the j loop in the *compute_mncc* function in Figure 5-(b)) on this algorithm. Due to these restrictions, only three different block sizes were tested: 4×40 (4 iterations of the i loop and 40 of j loop), 8×40, and 12×40.

3. **Transposing the thread block structure**
   Since the arrays $C2LX$, $C2LY$, $C2RX$, and $C2RY$ are indirectly accessed through $x1$ and $y1$, which are not necessarily consecutive, we do not know if there is spatial locality in the the outermost loop of the *compute_mncc* function. However, there is spatial locality in the innermost loop, the j loop, as consecutive elements of the arrays are accessed for each value of *num_disp*. Thus, it could be beneficial to form warps from adjacent inner loop iterations that have spatial locality, because accesses to the global memory can be coalesced [11]. Therefore, we transposed the thread-block structure by mapping the j inner loop to the X dimension of a CUDA thread block, effectively performing loop interchange. As described below, this improved performance by reducing DRAM traffic, apparently substantiating our hypotheses. As in the previous optimization, only three possible thread block sizes were used: 40×4, 40×8, and 40×12.

4. **Utilizing texture memory for read only data**
   Given this algorithm's heavy demands on memory accesses, it is important to optimize for locality. A few observations for the initial GPU implementation can be made. First, most of the loads in an outer loop iteration are indirect accesses to arrays in global memory (such as those that access $C2LX$). In addition, the data accessed through the previous indirect access is later used as an index to another array. Thus, we do not know statically if there is spatial or temporal reuse and, as a result, we cannot use the shared memory. However, we can use the hardware caching texture memory [11] to take advantage of any spatial or temporal locality that there might be in the code. For this optimization we tested the same thread block sizes that were tested in the previous optimization.

```
compute_mncc(data, Thread ID) {
  int start = start edge for ID
  int end   = end edge for ID
  for i=start, end {
    x1=x_edge[i];
    y1=y_edge[i];

    for j=0, num_disp {
      //find corresponding edges in L and R cameras
      x1_eL = (float *)(C2LX + x1*num_disp);
      y1_eL = (float *)(C2LY + y1*num_disp);
      x1_eR = (float *)(C2RX + x1*num_disp);
      y1_eR = (float *)(C2RY + y1*num_disp);
      ...
    }
    ...
    maxcorr(i) =0;
    for j=0, num_disp {
      ...
      corr1= ...
      corr2= ...
      corr3=...

      // find maximum correlation
      corr[i*num_disp+j]= corr1+corr2+corr3;
      if ( corr[i*num_disp+j]> maxcorr[i]) then
           maxcorr[i] = corr[i*num_disp+j];
    }

}
```

(a) Original

```
compute_mncc(data, Thread ID) {
  int start = start edge  for ID
  int end   = end edge for ID
  for i=start, end {
    for j=0, num_disp {
      x1=x_edge[i];
      y1=y_edge[i];
      //find corresponding edges in L and R cameras
      x1_eL = (float *)&C2LX [x1*num_disp];
      y1_eL = (float *)&C2LY [y1*num_disp];
      x1_eR = (float *)&C2RX [x1*num_disp];
      y1_eR = (float *)&C2RY [y1*num_disp];
      ...

      corr1= ...
      corr2= ...
      corr3=...
      corr[i*num_disp+j]= corr1+corr2+corr3;
    }
  }
}

find_maximum(data, Thread ID) {
  int start = start edge for ID
  int end   = end edge for ID
  for i = start, end {
    maxcorr[i] =0;
    for j = 0, NUM_DISP {
      if (corr[i*num_disp+j]> maxcorr[i]) then
           maxcorr[i] = corr[i*num_disp+j];
    }
  }
}
```

(b) Restructured

Figure 5: High-level View of Source Code Structure for MNCC

3.1.2 *CPU Specific Optimizations for MNCC*

Our CPU specific optimizations were not as extensive when compared to the GPU implementation. The two major optimizations applied were changing the block size of the outermost loop and selecting a thread scheduling policy (static versus dynamic).

3.2 Optimizations for Compute Homogen

As shown in Figure 3, the Compute Homogen code takes as input the output of the Delaunay Triangulation algorithm and MNCC. This code is not as regular as in MNCC. Homogen has an outermost loop that iterates across all the triangles obtained in the previous phase. Each iteration of the loop executes numerous if statements. Inside some of the if conditions there are for loops. Because of this, we were not able to come up with perfectly nested loops as in the MNCC code.

Nevertheless, we restructured the code by changing array accesses using pointer arithmetic to array subscripts. An important optimization that we applied was the removal of memory heap allocations called upon invocation of the kernel. In a majority of cases, they were unnecessary, and in cases when they were not, memory was instead allocated on the stack.

3.2.1 *GPU Specific Optimizations*

For Compute Homogen, since we do not have perfectly nested loops, we were not able to apply as many optimizations as in the case of MNCC.

1. **Naive GPU Implementation**
   We took the baseline implementation and, as before, we added the necessary constructs to execute in a GPU. We manually ran experiments to determine the best thread block sizes. We varied the thread block size between 32 and 384 in steps of 32. Larger thread block sizes were not tested because of this algorithm's heavy demand on GPU resources, in both size of code and register pressure, and CUDA not allowing any larger thread block sizes.

2. **Utilizing texture memory for read only data**
   Just as in MNCC, we changed the loads to read from texture memory. This is primarily used in order to take advantage of any spatial or temporal locality that the code may have. As before, we searched for the best thread block size.

3. **Compiler Flags**
   It is worth noting that this kernel has a large loop body and exhibits a high register pressure (more than 37 registers as reported by the NVCC compiler [11]). This significantly impacts the occupancy and performance for this kernel. Thus, we run experiments to find the best value using the `-maxregcount` compiler flag [11]. This flag sets the maximum number of registers the compiler can use before it needs to spill registers to global memory.

3.2.2 *CPU Specific Optimizations*

This particular algorithm suffers from a large number of control statements which results in load imbalance, as some iterations take more time than others to execute. To alleviate this problem, we used dynamic scheduling and varied the block size of the outermost loop.

4. EVALUATION

In this section, we report our evaluation results. First, we describe in Section 4.1 the environmental setup that we used to run the experiments. Then, we show the experimental results. Section 4.2 shows the impact in performance of the compiler choice. Section 4.3 shows the results for MNCC, while Section 4.4 shows the results for Compute Homogen.

4.1 Environmental Setup

We ran experiments on two different platforms:

1. For evaluation of GPU implementations, we used a Nvidia GTX280 GPU on an Intel Core 2 Quad Penryn clocked at 2.83GHz with 2×6 MB of L2 and 4GB of memory. This platform uses three compilers: Microsoft Visual C++, Intel ICC 10.1, and NVCC from CUDA 2.0.

2. The multi-core implementation was evaluated on a 24-core (composed out of four six-core chips) Intel Xeon Dunnington running at 2.40GHz and 48 GB of memory, where each chip has 3×3 MB L2 and 12 MB L3. The compiler used is Intel ICC 10.1 with flags `-O3 -xS`.

For the experiments reported in this paper, rather than running the TEEVE application in an actual tele-immersive environment with cameras, we used a simulated test environment where 3D reconstruction was made using a small set of sample images. Additionally, for all of the GPU experiments, we used the Asynchronous IO API provided by CUDA to transfer data to and from the device asynchronously. This had an overall positive effect on application performance, but not on kernel execution time.

4.2 Switching Compilers

Switching compilers can have a noticeable impact on overall performance. Table 1 shows the performance difference when the original unmodified TEEVE code is run using Microsoft Visual C++ compiler or the ICC compiler on the Intel Quad Penryn described in Section 4.1. When compiling with the ICC compiler we can set the flag `-fp-model precise`, which guarantees value-safe optimizations on floating-point data. We compiled the application with and without this flag.

As the table shows, when the application is compiled with the ICC compiler, there is a speedup of up 1.55 than when it is compiled with the Microsoft Visual C++ compiler. We attribute the difference in performance to ICC's auto-vectorization optimization. When the application was run without the `-fp-model precise` flag, the application has a higher frame rate (31 FPS versus 28 FPS), but we observe a significant floating point precision loss. It is yet to be determined how this precision loss affects the quality of the resulting image.

4.3 MNCC

We now examine the impact of each of the optimization steps described in Section 3.1.

4.3.1 Thread Block Size

Figure 6 shows the effect of different thread block sizes on the naive CUDA based implementation of the MNCC algorithm. For the baseline run of MNCC, the best one dimensional thread block size was 160, which resulted in MNCC running in 4.7 ms on the GPU. For this algorithm, a speedup of 1.14 can be seen just by selecting the best thread block size between the range 128 and 480, in steps of 32.

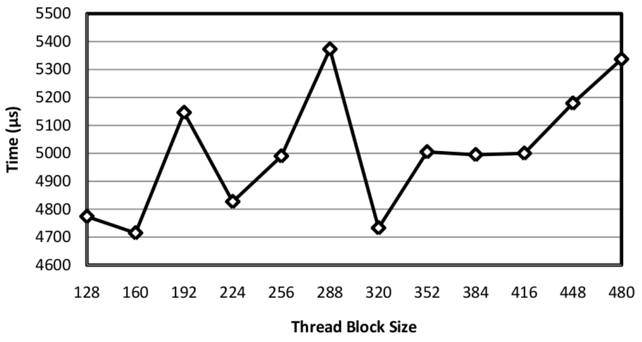


Figure 6: Performance for 1D Thread Block Sizes

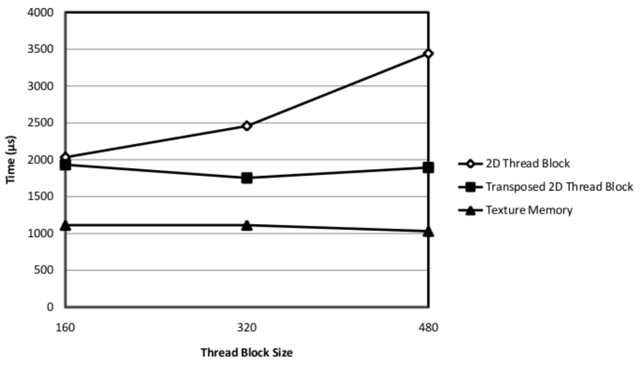


Figure 7: MNCC Optimizations

Figure 7 shows the performance impact of the rest of the optimizations described for MNCC in Section 3.1 as we vary the tested thread block sizes. The figure shows that for each optimization the best tested thread block size is different, so this is a parameter that needs to be optimized independently for each transformation.

The Line 2D Thread Block in Figure 7 shows the benefit of parallelizing both the outer and the inner loop of MNCC. The best 2D thread block size is 160 (4×40), which runs in 2.0 ms, a speedup of 2.32 compared with the baseline algorithm. Choosing a block of 480 (12×40) can result in only a performance speedup of up to 1.69, so for this optimization, it is important to choose the appropriate thread block size. The line Transposed 2D Thread Block shows the performance impact as we vary the block size when the i and j loops are exchanged. In this case, MNCC executes in 1.8 ms, a speedup of 1.1 when compared to the worst tested block size for this optimization. Finally, when texture memory is used in combination with the transposed 2D Thread Block, MNCC runs in 1.0 ms when using a block size of 480 (40×12). For this last optimization, the effect of different thread block sizes are not as important, as it results in only a speedup of 1.08.

These results show that for some of the transformations, the impact in performance is not as large when compared to the transformation itself. There can be other non-evaluated block sizes that might result in a change in performance, thus increasing the need to have a mechanism to empirically search for ideal block sizes.

4.3.2 Overall Impact

Compiler	MS VC++	ICC -O3 -xS -fp-model precise	ICC -O3 -xS
Frames Per Second	20	28	31

Table 1: Performance from switching compilers on 4-core Intel

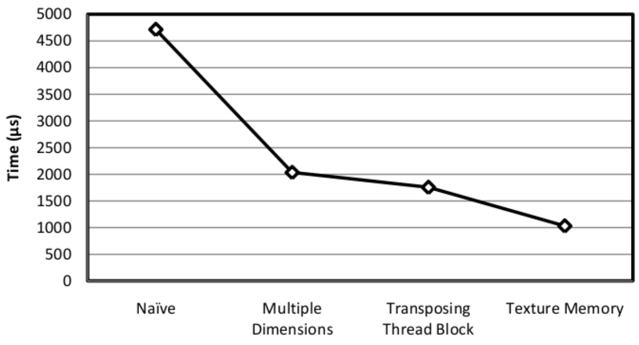


Figure 8: MNCC GPU Optimization Trend

Figure 8 shows the overall effect of all the GPU optimizations on MNCC using the best thread block size. From this figure, we see that the most optimized code using texture memory runs more than 4 times faster than the naive version (1.0 ms versus 4.7 ms). The most noticeable performance optimization was in converting from a single level of parallelism to an element by element parallelism that resulted in a code that runs more than 2 times faster than the previous baseline optimization (2.0 ms versus 4.7 ms).

Finally, Figure 9a shows the performance of the original and restructured MNCC code shown in Figure 5 as we vary the number of threads from 8 to 24 for the CPU implementation. As the figure shows, the restructured code performs slightly worse than the original one. This is due to two factors. First, the overhead of the separation of one kernel into two separate kernels, in which the same control code is replicated in both. The second reason is the extra barrier that is necessary between MNCC and the find-maximum reduction operations. Th overhead is almost negligible when MNCC is run using 24 threads. As the figure shows, MNCC scales linearly as the number of threads increases. For this algorithm, static scheduling was the best thread scheduling policy, as all the iterations perform the same amount of work.

Our tuned GPU version of MNCC outperforms our fastest CPU implementation by over two times (1.0 ms versus 2.1 ms). This result is an example of the advantage GPUs can provide for an algorithm such as MNCC.

4.4 Homogen Computation Results

We now examine the impact of each of the optimization steps described in Section 3.2.

4.4.1 *Baseline and Texture Memory Homogen*

Figure 10 shows the execution time of the Compute Homogen kernel as the thread block size changes for both the naive version and the one that uses texture memory, as explained in Section 3.2. When using the best block size, the naive implementation executes in 17.4 ms. When using the texture memory (and it's best thread block size), the algorithm runs in 15.1 ms, thus giving the texture memory optimization a speedup of 1.15. For this algorithm, a thread block of size 32 results in bad performance, but blocks of sizes larger than that perform similarly to each other.

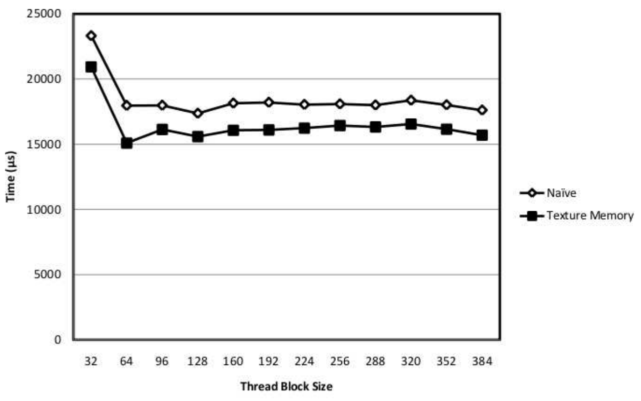


Figure 10: Homogen GPU Optimization Performance

4.4.2 *Setting Maximum Register Count*

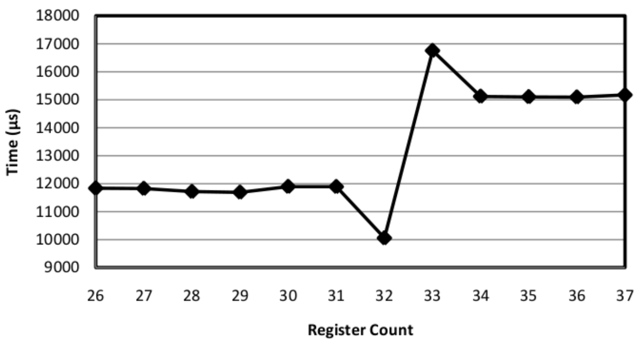


Figure 11: Maximum Register Count Performance

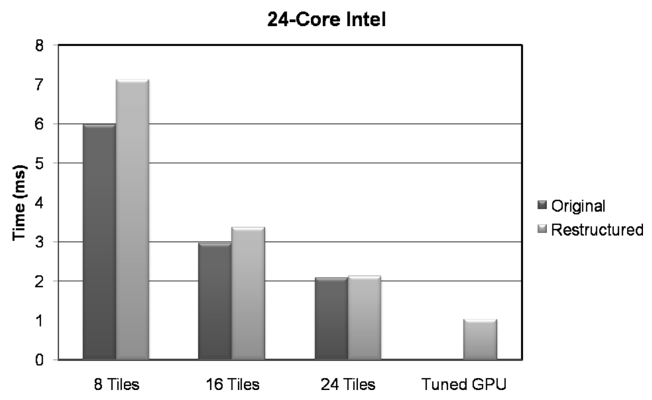
For the texture memory implementation of Homogen, the compiler uses a spill register threshold of 37 live registers, as shown in the compiler output. This is important since anymore than 37 live registers will need to be spilled into global memory.

Using the method described in Section 3.2.1, by forcing the compiler's register spill threshold from 37 registers to 32, the Homogen kernel will now execute in 10.1 ms, as shown in Figure 11. This corresponds to a performance speedup of 1.51, making this a valuable tuning parameter. By having the compiler use a lower register spill threshold, the occupancy of the device increases, thus increasing the parallelism and potentially increasing performance.

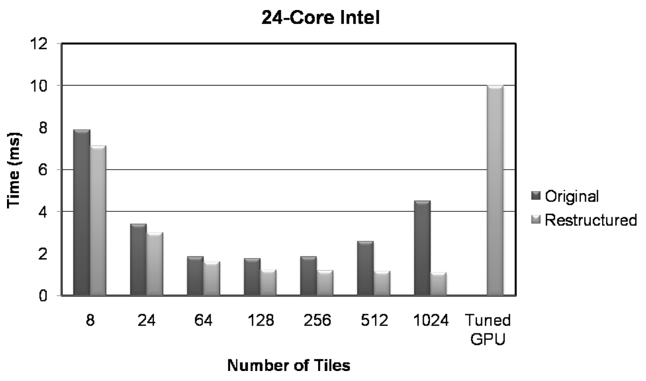
4.4.3 *Overall Results*

Figure 12 shows the overall effect of all the GPU optimizations using Homogen. For this particular algorithm, we see that the most noticeable performance optimization was using the CUDA `-maxregcount` compiler flag. This leads to an overall performance speedup of 1.73 when compared with the naive implementation of Homogen.

Figure 9b shows the performance of the original and restructured Homogen computation kernel. Since the amount of computation done by each iteration of this code is different, this code benefited from dynamic scheduling. Thus,



(a) MNCC Results



(b) Homogen Computation Results

Figure 9: Kernel Results

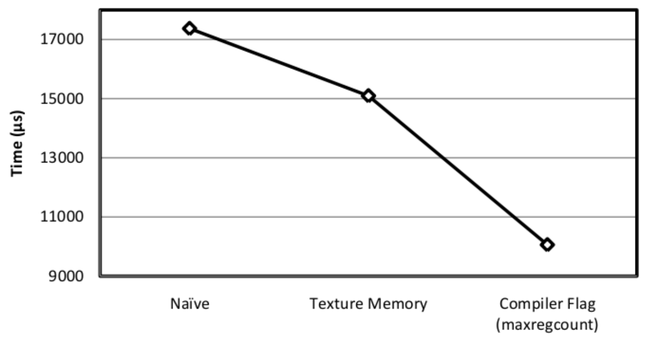


Figure 12: Homogen GPU Optimization Trend

by reducing the block size and then increasing the number of blocks, performance improves. Additionally, as the problem is over-decomposed, the modified version achieves its best performance (1.12 ms) and eventually tapers off, unlike the original whose performance worsens after 128 blocks. A reason for this is because in the original code, there are numerous memory allocations that are called upon invocation of the kernel by all threads. As the number of calls to the memory allocator increases, the overhead associated with the calling of a system library will limit the kernel scalability. We were able to remove most of the library calls in the restructured code, and in cases where we could not, the data is allocated on the stack rather than on the heap.

In the case of our tuned GPU implementation, the performance is worse than all the parallel versions that run in the CPU. There are several reasons for this. First, the code is still overly complex and has high demands of the GPU resources, which then limits the occupancy of the device. Second, the code has numerous control statements. This is particularly bad for GPUs since the different branches are executed one after the other when threads within a warp start to diverge.

As these results show, CPUs are better suited than GPUs for codes with heavy control flow. Additionally, we see that by over-decomposing the problem into more blocks than the number of threads (24 in our case), performance improves.

4.5 Overall Application Results

Figure 13 shows the effect of all optimizations to the original TEEVE application. We estimate the FPS (Frames Per Second) by running each of the kernels independently. With the new optimizations, our estimation of the new frame rate is 44FPS, a noticeable improvement compared to the original 20FPS, as was shown in Figure 4. The performance improvement is evident in the new time for MNCC and Homogen. In addition, the newly profiled times of the other algorithms have been increased due to using a different compiler.

We have successfully optimized the two most computationally intensive kernels. Now the major bottleneck occurs in the Delaunay Triangulation algorithm, as we expected.

5. FUTURE WORK AND CONCLUSION

Our final goal is to add a GPU backend to the HTA data structure that we presented in [3, 5] so that applications written using the HTA data structure and primitives can be executed on CPU based multi-cores or GPU platforms without modifying the code. Work is in progress to add an empirical autotuning framework that will enable tuning for architectural features using HTAs. In addition, we believe further optimizations can be applied to the Compute Homogen kernel, which directly correlates to GPU resources and occupancy. Through further simplification we can extract additional device occupancy. Also, after optimization of the MNCC algorithm, it is evident that the Delaunay Triangulation algorithm has become the biggest bottleneck. For this, we are investigating parallel implementations of it [8]. Lastly, the results were measured in isolation, and the next step is to integrate them into the original application.

This paper presented the process of restructuring and then optimizing the most computationally expensive sections of the tele-immersion application. Thanks to the transformations, we can now efficiently run the algorithms on large-scale multi-core processors and GPU platforms.

We had three goals in optimizing tele-immersion codes and met them. We achieved portability across data-parallel operations. By restructuring the 3D reconstruction algorithms into data parallel form, we can adapt the code for any parallel platform. We sped up the original code through the use of different optimizations, leading the optimized GPU implementation of MNCC to be over two times faster than on a 24-core server platform. Our CPU implementation of the Homogen kernel showed the value of over-decomposition when there is a load imbalance. Also, we believe that as a result of code restructuring, maintainability has improved by having the code made simpler and more readable. Finally, a systematic procedure for optimizing data-parallel computer

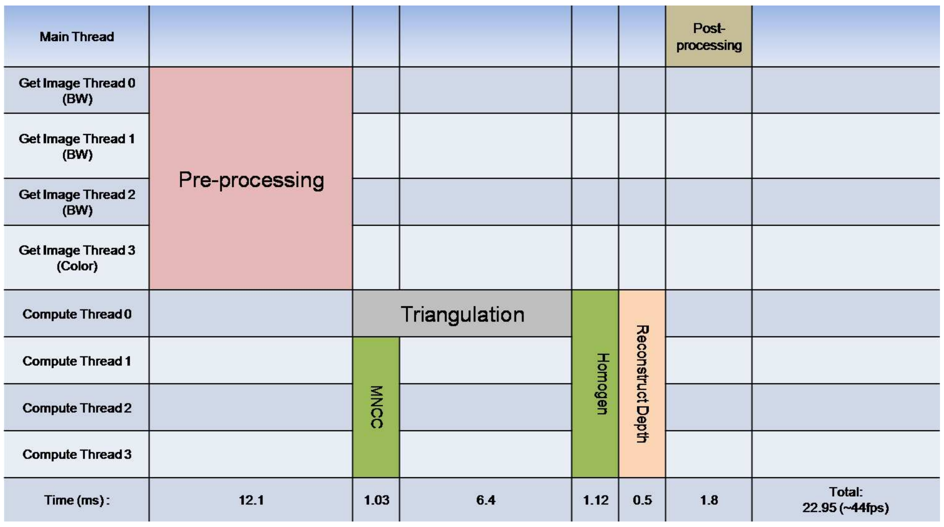


Figure 13: Modified Tele-Immersion Execution Flow

vision kernels for parallel architectures is demonstrated.

6. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Awards CCF 0702260, CNS 0509432, the Gigascale Research Center (GSRC) funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign, sponsored by Intel Corporation and Microsoft Corporation. Additionally, we would like to thank Megan J. Osfar and James Brodman for their comments.

7. REFERENCES

[1] Parallel Computing Research at Illinois: The Upcrc Agenda. Technical report, Dept. of Computer Science, Dept. of Electrical and Computer Engineering, Corrdinated Science Laboratory, Nov 2008.

[2] G. Almasi, L. D. Rose, J. Moreira, and D. Padua. Programming for locality and parallelism with hierarchically tiled arrays. In *In Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2003*, pages 162–176. Springer-Verlag, 2003.

[3] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. ría J. Garzarán, D. Padua, and C. von Praun. Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In *Proc. PPoPP'06*, pages 48–57, 2006.

[4] B. Delaunay. Sur la sphère vide. *Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7:793–800, 1934.

[5] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzarán, and D. Padua. Programming with Tiles. In *Proc. PPoPP'08*, pages 111–122, 2008.

[6] M. Harris. Optimizing Parallel Reduction in Cuda, 2007.

[7] S.-H. Jung and R. Bajcsy. A Framework for Constructing Real-time Immersive Environments for Training Physical Activities. *Journal of Multimedia*, 1(7):9–17, 2006.

[8] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proc. of the 2007 ACM SIGPLAN conf. on Programming Language Design and Implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[9] J. Mulligan, V. Isler, and K. Daniilidis. Trinocular stereo: A real-time algorithm and its evaluation. *International Journal of Computer Vision*, 47:51–61, 2002.

[10] K. Nahrstedt, 2008. private communication.

[11] NVIDIA. Nvidia Cuda Programming Guide 2.0, 2008.

[12] J. H. Wolf. Programming methods for the Pentium III processor's streaming SIMD extensions using the VTune performance enhancement environment, May 1999.

[13] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] Z. Yang, K. Nahrstedt, Y. Cui, B. Yu, J. Liang, S. hack Jung, and R. Bajcsy. Teeve: The next generation architecture for tele-immersive environment. In *ISM '05: Proceedings of the Seventh IEEE International Symposium on Multimedia*, pages 112–119, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Z. Yang, B. Yu, W. Wu, R. Diankov, and R. Bajcsy. Collaborative dancing in tele-immersive environment. In *MULTIMEDIA '06: Proceedings of the 14th annual ACM international conference on Multimedia*, pages 723–726, New York, NY, USA, 2006. ACM.