Multi-GPU Accelerated Software Tool for Virtual Hand Repositioning

Adam Gorski[1], O. Remus Tutunea-Fatan[2], Louis M. Ferreira[3]

[1]Western University, agorski3@uwo.ca
[2]Western University, rtutunea@eng.uwo.ca
[3]Western University, lferreir@uwo.ca

Corresponding author: O. Remus Tutunea-Fatan, rtutunea@eng.uwo.ca

Abstract. The progress of additive manufacturing and scanning technologies has made the use of 3D printed splints in clinical settings increasingly practical. However, certain obstacles continue to impede the wider adoption of this fabrication avenue. More specifically, one of the main challenges is constituted by the difficulties encountered by some of the patients while attempting to reach predefined finger positions. In case of traditional thermoplastic splints, physiotherapists can manually adjust patient's fingers while the splint material molds to their hand. This direct adjustment is not possible when 3D scanners are used since any physiotherapist intervention would disrupt the scanning process and thereby invalidate the acquired data. Although some patients are capable of holding their fingers in predetermined positions, many who suffer from injuries lack the ability to do so. This significantly complicates the acquisition of high-quality 3D scans.

The current study showcases a joint repositioning tool based on finite element analysis (FEA), equipped with advanced algorithms for generating real-time joint repositioning outcomes. This tool employs a specialized multi-GPU accelerated solver to swiftly carry out mesh voxelization and solve the finite element system for a geometrically complex hand model. Furthermore, the finite element solver is meticulously optimized to reduce both time and memory consumption, ensuring that diverse systems can achieve rapid performance in joint repositioning.

Keywords: Hand Kinematics, Splint Generation, Finite Element Method, Multi-GPU Parallelization
DOI: https://doi.org/10.14733/cadaps.2025.912-926

1 INTRODUCTION

Physiotherapist commonly use splints to immobilize and stabilize limbs to accelerate patient healing and recovery. In the context of hand splints, advancements in scanning and 3D printing technology allow for cheap

[2] splints that can potentially better capture the curvature of hands. While software tools to create splints from hand scans were already developed [6] and low cost 3D printers [23] and scanners [20] have eliminated many barriers towards the wide scale adoption of this technology, few obstacles continue to exist.

The creation of digital splints faces ongoing hurdles, especially during the non-contact scanning of hands for finger positioning. Traditionally, physiotherapists manually adjust patient's fingers during splint molding or setting process in order to achieve optimal positions for hand recovery. However, many patients struggle to hold finger positions on their own. This challenge is exacerbated when non-ionizing 3D scanners (laser or photogrammetry type) are used to digitize patient's hand. These devices cannot capture any part of the hand that is obscured from their sensors. This includes interference from physiotherapist's hands and/or body during positioning. These occlusions can lead to incomplete data, potentially compromising the utility of the hand scan. Alternative methods, including the use of thin wires to position fingers, exist but can be cumbersome, uncomfortable for the patient, and time-intensive to set-up. Because of this, when non-contact scanning solutions are used, fingers have to be repositioned prior to entering the splint design process.

Finite element analysis (FEA) stands out as a promising method for digitally adjusting the position of fingers. To this end, commercial FEA software is capable of implementing displacements, as noted in the literature [8]. However, these platforms often lack intuitiveness and interactivity, primarily because they cater to a broad spectrum of engineering needs. Typically, in such scenarios, the required displacements are predetermined, eliminating the need for on-the-fly adjustments. As a result, solving for the desired outcomes can consume a substantial amount of time. This is in stark contrast to the practices of physiotherapists, who prefer to iteratively modify the positioning of fingers until it aligns with their empirical clinical judgments.

To address the aforementioned shortcomings, this study introduces a specialized software for virtual joint repositioning, built on finite element analysis (FEA) and tailored for the needs of the physiotherapists. The developed software tool is equipped with a straightforward, step-by-step interface aimed at guiding those without technical expertise through the process of joint repositioning. The software tool works as an independent application and leverages graphics processing unit (GPU) parallelization to expedite the solution-finding process. Special emphasis is placed on optimizing meshing and solving algorithms to reduce computational time and enhance user experience. Custom algorithms were developed to address issues encountered with GPU-accelerated meshing and solving. These include eliminating the need for sparse matrix assembly by using matrix-less solving approaches. Given that large FEA models demand substantial memory and GPUs typically have less memory than CPUs, the matrix-less approach and memory-less preconditioner enable the use of much higher mesh resolutions. Furthermore, the software benefits from the utilization of multiple GPUs or OpenCL computing devices in order to facilitate real-time interaction, thus enhancing its practicality for clinical use.

2 PROCESSING WORKFLOW

The process of repositioning joints is divided into four distinct stages, with each stage leading linearly to the next. However, the final two stages are designed to be employed concurrently to fine-tune the joint angles. Additionally, the software allows for the simultaneous loading of multiple models, enhancing its versatility. The typical processing workflow (Fig.1) consists of the following steps:

1. Geometry loading: user loads a 3D model of the hand and confirms its scale.
2. Joint placement: user sequentially select joints, manually positioning them or placing them by clicking on the virtual hand model. An automatic placement feature is also available.
3. Joint rotation: user relies on rotation tools to adjust the virtual bones of the model in order to their correct positions.
4. Angle verification: software calculates the angles between specific joints to enable the application of precise recovery angles within the model.

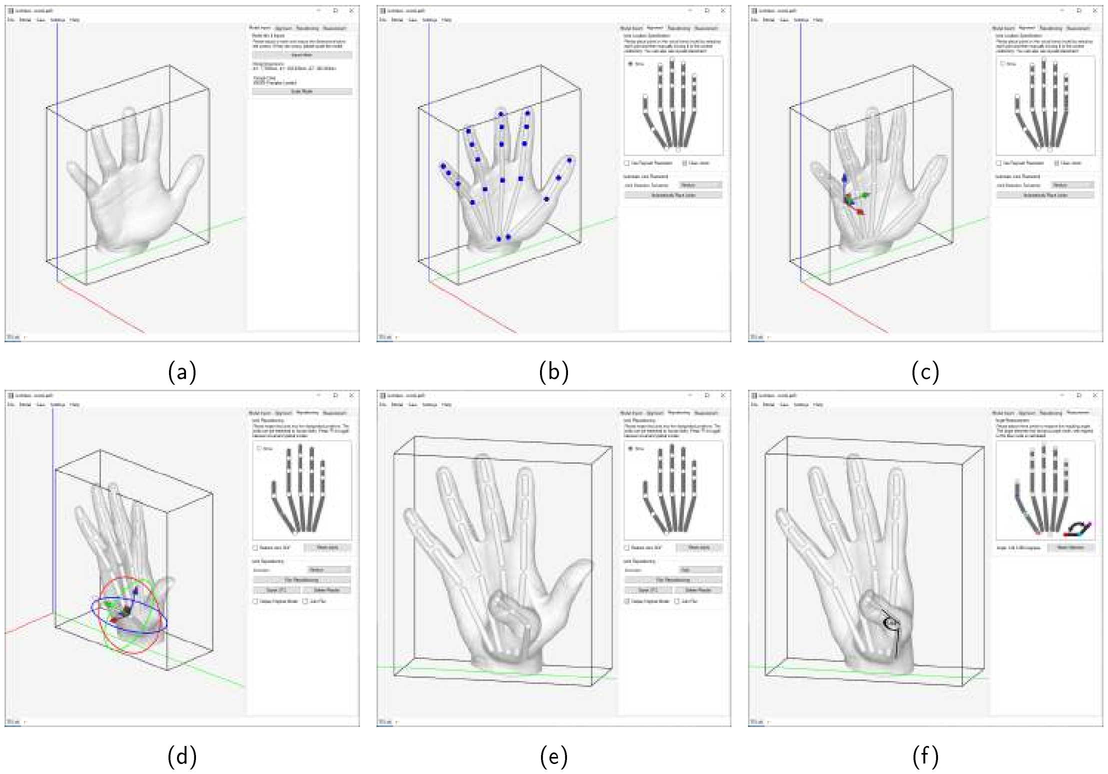


Figure 1: Typical workflow: (a) geometry loading, (b) joint placement, (c) joint adjustment, (d) joint repositioning, (e) inspection, and (f) angle verification.

2.1 Geometry Loading

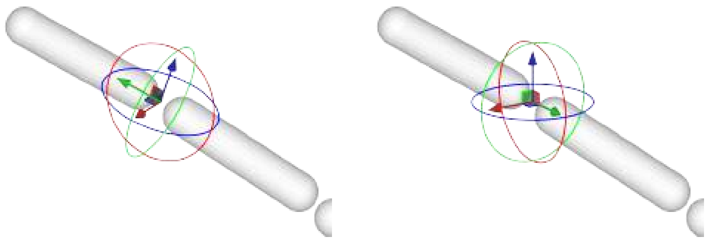
The initial step requires the user to import the model's geometry by selecting a mesh file for import and confirming its dimensions. Additional checks are in place to validate the dimensions of the imported model. The size of the bounding box of the mesh is displayed for a facile reference.

2.2 Joint Placement

This stage entails adding virtual joints to the digital model. This process can be executed using a raycast method according to which a ray is projected onto the model, and the midpoint of two intersection points is designated as joint's location. Another method involves adjusting joints' positions using translation controls. Furthermore, an automated tool for joint placement was also included in the software tool. This approach aims to identify accurate joint positions by analyzing the curvature of the input mesh.

2.3 Joint Rotations

The last phase involves adjusting the bones to align with their intended targets. Included in each joint is a tool for rotational control, enabling precise adjustments of the bone angles. This rotational control can operate in either a local or global mode. In local mode, the rotational axes align with the bone itself, whereas in global mode, the axes correspond with the overall coordinate system (Fig.2). When the auto run mode is activated, the software automatically processes the model after it has been repositioned.



(a) (b)

Figure 2: Joint angle control options: (a) local and (b) global

2.4 Angle Verification

During this stage, users have the opportunity to check the angles between joints, as physiotherapists frequently target precise angles for therapeutic purposes.

3 IMPLEMENTATION

The software employs OpenGL for graphical rendering and utilizes CUDA and OpenCL for computational tasks. Custom-written wrappers for OpenCL, OpenGL and CUDA ensure optimal performance. The user interface is developed using WinForms, while user interaction with the 3D scene is facilitated through raycasting and a click buffer. The core of the tool is programmed in C#, with the Advanced Vector Extension (AVX)-256 solver implemented in C/C++. This application operates independently, eliminating the need for external plugins or libraries. Automatic joint placement is achieved by using hashmaps to identify adjacent triangle edges and an iterative search algorithm to cluster areas of low curvature. These areas are then removed from the overall model to pinpoint regions of high curvature, through which the software plans the path for bones. Internal bone rotation controls are matrix-based, and quaternion-based spherical interpolation is employed for realistic joint positioning during each step of the finite element load. While certain simplifications - such as using basic bone structures or material data - were implemented to enhance performance, these are not expected to significantly impact the analysis of displacements that represents the primary focus of the developed software tool.

3.1 GPU Mesher

In constructing finite element meshes, Delaunay triangulation is often favored for producing meshes that exhibit good Jacobian ratios. However, rapidly creating meshes through Delaunay triangulation poses a challenge, particularly because, in the best-case scenario, it is an O(Nlog(N)) operation, even when employing the divide-and-conquer strategy [13]. Parallelized versions also exist [3] but in more general terms the parallel processing capabilities offered by voxelization - that relies on a grid-based approach - are difficult to match.

Upon importing a model, the finite element mesh can be calculated and saved. However, in many instances, repositioning leads to such significant distortions of the elements that the isoparametric mapping becomes ineffective when recalculating element stiffness for geometrically non-linear solutions. To address this issue, the finite element solver is required to remesh at each load step interval, rendering the storage of any prior meshes beneficial only for the initial load step. Storing the mesh beforehand could be a feasible option for geometrically linear solutions. Yet, given the substantial displacements observed in the finite element model, incorporating geometric non-linearity is essential.

Unfortunately, voxelized meshes (Fig. 3a) require very high resolutions for good surface accuracy since the hexahedral elements cannot accurately model a smooth surface. Additionally, lower resolutions result in volume gain or losses that can artificially shrink or expand the model. When multiple remeshes are performed, this error becomes very apparent. To solve these issues, a voxel based mesh with single contouring was used to generate a smoother mesh (Fig. 3b). Unlike dual contouring methods [17] that are more accurate for sharp edges, human hands tends to have no sharp corners and this makes the single contouring method ideal. The underlying algorithm is similar to the marching cubes algorithm [15], however a special emphasis has to be introduced to generate more accurate meshes. The single contour mesh generator also supports directly adding tetrahedron elements to the mesh, or alternatively interpolating the edge data based on a more loose voxelization approach. The former method yielded more accurate results, while the latter approach was faster. The meshing algorithm is depicted in Fig. 3.

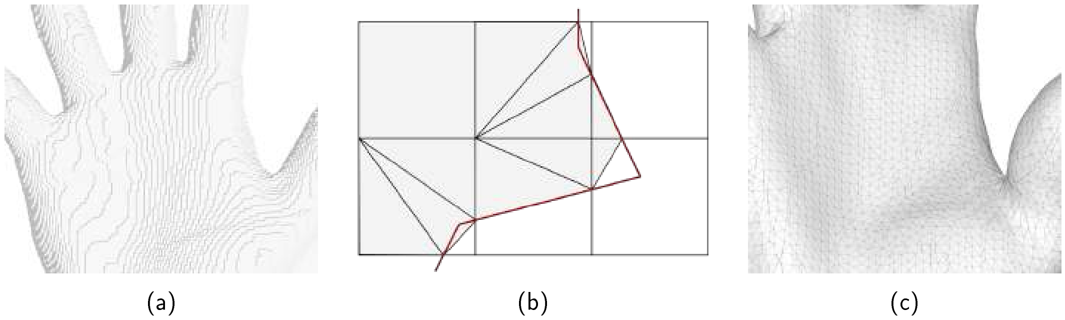


Figure 3: Meshing techniques: (a) regular voxel mesh, (b) voxel mesh data interpolation and (c) single contour mesh

The initial step involves computing the bounding box of the target object. This is done via parallel reduction on the GPU. After that, each object in the FEA system must be voxelized. The GPU voxelizer does this by shooting a grid of rays into into the mesh at 3 different angles such that it can determine the interior and interpolation values at each ray entrance and exit. Theoretically, this process could be further optimized by using the dedicated ray-tracing cores on the GPU [9], however this would place restrictions on the hardware to be used and this was regarded as undesirable. Moreover, a greater focus was given to enhancing the finite element solver because this is often the primary bottleneck in the process. It is also important to mention that after completing the voxelization, the finite element solver has the capability to operate directly on the voxel data. This approach could reduce the time spent on meshing, but it would eliminate the contouring system that mitigates volume gain and loss effects. Additionally, this method would lead to performance and memory challenges that are detailed further in section 3.2.

The subsequent phase is dedicated to creating the mesh data, a task executed entirely on the GPU, except for the node ordering and assignment phase, which is handled by the CPU. The use of addition and atomic counters is minimized because their extensive use results in diminished performance. [7, 4]. Assigning node stages could also be carried out on the GPU but this approach would lead to randomized node patterns. Such patterns would adversely impact the distribution of work across multiple GPUs and necessitate substantial use of atomic compare-and-exchange operations that would detrimentally influence performance.

This approach is similar to the one used for generating finite element meshes from computerized tomography (CT) data [5] however, this approach relies heavily on extra interpolation, and offloads almost the entirely

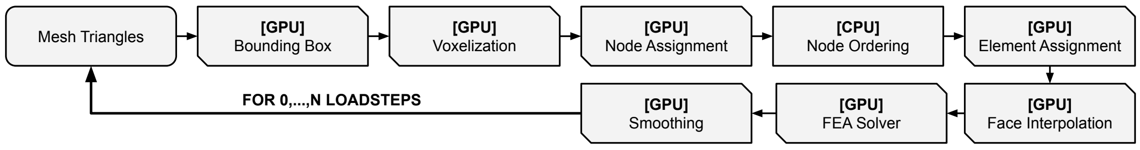


Figure 4: GPU accelerated meshing

of the workload onto the GPU. As noted earlier, the single contouring interpolation is necessary as voxelized hexahedron meshes tend to suffer from volume gain or loss, which depends on the voxelization tolerance. The complete flowchart of the solver is illustrated in Fig. 4.

3.2 GPU FEA Solver

The finite element solver selected for implementation is based on the Galerkin element formulation and relies on the penalty method for its boundary conditions. The Lagrange method was also considered since it is characterized by a superior stability in case of certain applications [18, 21], but the former method was just found to be easier to implement. Isoparametric mapping wa used to compute the local stiffnesses of each element. The setup only uses three materials: bone, muscle and joints. To keep the model simple, each material was assumed to be isotropic and characterized by Poisson's ratio of 0.3. The individual Young's moduli for the bone, muscle and joints are 20 GPa, 100 kPa and 5 kPa, respectively. While it was previously suggested that the skeletal muscle has a stiffness of around 24.7 kPa [16], a stiffness of 100 kPa was used in the current work to ensure a faster convergence. Ideally, joint stiffness should have been set to 0, but since this value leads to singularities, a non-zero value was used again for faster convergence purposes. Finally, bone stiffness values were taken from [10] since they represent good approximations.

After calculating all element stiffness, the solver proceeds by either directly computing the solution through a matrixless method or by employing a conventional block sparse technique to solve the system. Of note, meshless methods are viable since they can directly utilize the voxelized data from the GPU. [14]. Regrettably, this method demands considerably more memory due to a substantial rise in idle degrees of freedom within the finite element, which serve no purpose other than occupying space in the solving vectors. As a result, block sparse matrix and matrixless solvers were alternately employed to address the computational needs of the finite element model.

To quickly solve the large finite element system, the preconditioned conjugate gradient (PCG) method was selected primarily owed to its fast solving time, good memory usage as well as ease of implementation [19]. Since the PCG method only necessitates a matrix-vector multiplication, various specialized strategies can be adopted to streamline and expedite the computation. Both block Jacobi and Jacobi preconditioners were evaluated. However, block Jacobi preconditioners were found to enhance convergence efficiency, as evidenced by fewer iterations required during testing with the block Jacobi preconditioner. The principal drawback of block Jacobi resides in that it requires the sparse matrix-based solver to allocate three extra rows of vectors to store the preconditioner. This leads to a 1.75-fold increase in memory consumption for the PCG algorithm alone. Fortunately, the preconditioner can be generated dynamically, offering a memory-saving advantage albeit at the expense of the performance. The PCG algorithm used is detailed in Algoritm 1. The algorithm utilizes a tolerance level of 0.0001 and operates within a convergence window of 16 iterations. Double precision is adopted for the solver to mitigate the impact of round-off errors, which can significantly hinder achieving the set tolerance and result in a substantial rise in the number of iterations needed. Testing revealed that using single precision afforded only a twofold speed boost for dot product calculations, yet it necessitated a markedly higher number of iterations to reach acceptable convergence levels.

Algorithm 1 Preconditioned Conjugate Gradient

```
Input: $A$, $b$, $tol$
  rk $\leftarrow b$
  pk $\leftarrow M^{-1}$rk
  $\beta_{old} \leftarrow$ pk $\cdot$ rk
  w $\leftarrow A$pk
  $\alpha \leftarrow \beta_{old}/(pk \cdot w)$
  xk $\leftarrow \alpha$pk
  rk $\leftarrow$rk $- \alpha$w

  while rk $\cdot$ rk$> tol^2$ do
      w $\leftarrow M^{-1}$rk
      $\beta \leftarrow$ rk $\cdot$ w
      pk $\leftarrow$ w $+ (\beta/\beta_{old})$pk
      w $\leftarrow Apk$
      $\alpha \leftarrow \beta/($pk $\cdot$ w$)$
      xk $\leftarrow$ xk $+ \alpha$pk
      rk $\leftarrow$ rk $- \alpha$w
      $\beta_{old} \leftarrow \beta$
  end while
```

To achieve rapid GPU computation, the PCG algorithm incorporates parallel reduction alongside local synchronization. Performance evaluations were conducted using both OpenCL and CUDA frameworks, with no significant difference in efficiency observed between them. While both utilized workgroup/block reduction techniques, CUDA additionally leveraged warp level primitives. Nonetheless, despite these warp level optimizations, there was no noticeable impact on performance outcomes.

The sparse matrix solver operates similarly to a conventional finite element solver, with the distinction that its column indices are condensed into 3x3 blocks, treating each block as a single entry. In contrast, the matrixless solver operates by moving directly from node to element to node, thereby utilizing significantly less memory. For processing each sparse matrix block row, the matrix-based solver employs 32 work-items/threads, whereas the matrix-less approach requires only one work-item/thread *per block* row. Additionally, the matrix-less solver maintains a separate table of local stiffness matrices, which it integrates into the system dynamically. Table 1 illustrates the solver iteration rate. Since not all modes are supported by each device, N/A is used as the placeholder. It can be observed how the switch to a memory optimized model greatly boosts GPU performance, but has little effect on the CPU.

	AVX-256	**GTX 1080 Ti**	**RTX 3060 Ti**
Sparse Block Matrix	N/A	121.1	129.0
Matrix-Less	32.0	102.3	92.5
Matrix-Less Optimized	31.2	138.8	154.4

Table 1: Averaged iteration rates associated with various solvers (iterations/sec)

In addition to its primary capabilities, the solver offers a fallback to CPU with support for 256-bit wide Advanced Vector Extensions (AVX). This feature differs from the standard matrixless solver as the optimized version ensures vectors from overlapping node indices are loaded just once. This approach yields a significant boost in performance.

3.3 Multi-GPU Optimization

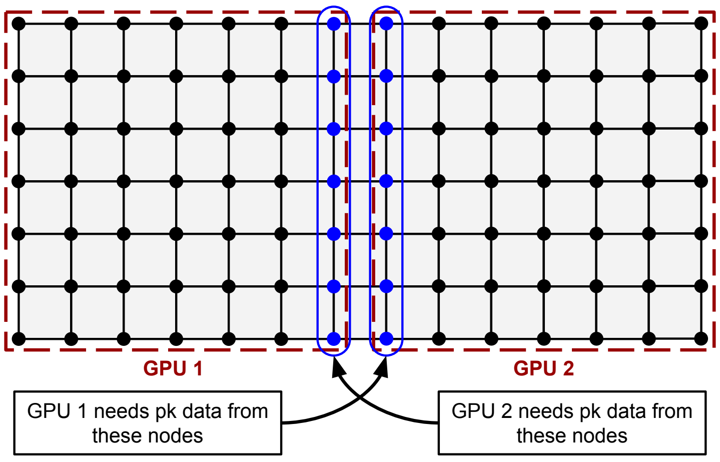
As the resolution increases, solving large finite element systems becomes progressively more time-consuming, making the development of a scalable multi-GPU finite element solver increasingly vital. In a multi-GPU PCG algorithm, there are three critical points of synchronization. Two of these points entail the main thread aggregating the dot product calculations from two GPUs. The third synchronization point requires the exchange of data between GPUs, a step essential for ensuring each GPU has access to pk data from nodal areas beyond its designated range. For instance, Fig. 5a demonstrates a finite element mesh consisting of quad elements and nodes. The mesh is divided nodally across two GPUs. Nodes that are exclusively managed by one GPU but are linked through an element to nodes managed by another GPU necessitate the sharing of values during the matrix vector multiplication stage of the PCG algorithm. To address this challenge, nodes needed by each GPU are marked in a shared global buffer. Subsequently, each GPU examines this buffer to identify if it possesses any of the marked nodes. The nodal data is then organized such that any data required for import by a GPU is positioned at the end of its local buffer, with data to be exported placed adjacent to it. This optimization is shown in Fig. 5b. By ensuring that the data for import and export is organized sequentially, the process not only facilitates the transfer of the entire memory segment from the GPU in a single OpenCL driver operation but also reduces the total amount of memory that needs to be accessed, eliminating the need for remapping or additional sorting. This approach drastically cuts down on the memory synchronization overhead for each PCG iteration. The buffer for exchanging the pk vector is kept in the host computer's memory because direct memory transfers between GPUs are not supported in OpenCL. Although CUDA offers capabilities for such transfers, OpenCL was chosen for its flexibility, allowing for the integration of diverse types of computing hardware.

While this approach does lead to some degree of memory overlap, it was observed that even models with multi-million degrees of freedom required less than a megabyte for data exchange. It is worth noting that this amount of memory used can be further reduced by minimizing the cross-sectional area at the boundaries between GPUs in the finite element mesh. The mesh shown in Fig. 5a features a vertical split which minimizes the cross-sectional, while if the split were to be redrawn horizontally, there would be significantly more data to exchange.

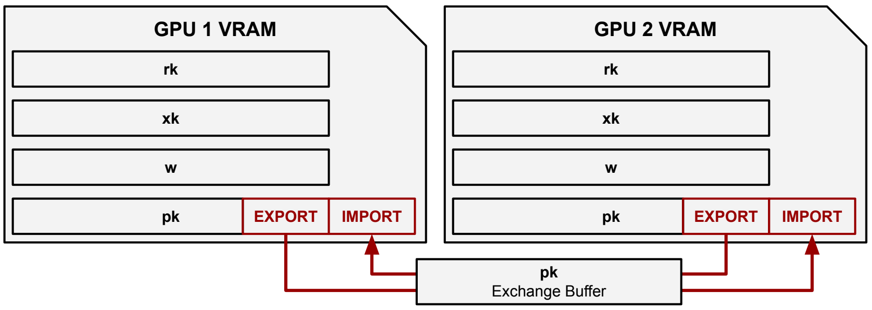
During testing, it was found that exchanging the pk vector incurred minimal performance cost. The primary source of performance degradation was due to one GPU completing its tasks slightly ahead of the other, leading to brief periods of idling. This discrepancy is particularly noticeable with smaller finite element models, though it's less of a concern since these models typically require minimal time to solve. To optimize GPU load distribution, an axial compression benchmark is performed at startup to assess each GPU's iteration rate. This allows the workload to be allocated in the most efficient way possible.

For the sake of simplicity, the multi-GPU solver exclusively employs OpenCL, given its capability to offload tasks to the system's CPU. Although this feature is seldom used in the context of joint repositioning, it proves beneficial for some finite element models that require slightly more memory than what is available on the GPU. Instead of upgrading the GPU or incorporating an additional one, the system's local memory can be utilized to compensate for the shortfall. While OpenCL theoretically has the capability to dynamically manage memory allocation between the GPU and system memory, such operations tend to be inefficient. Moreover, although it's possible to manually integrate CPU support into a separate solver, relying on OpenCL to manage CPU tasks significantly reduces the amount of code and complexity involved.

However, regular synchronization still has to occur during the PCG algorithm. Each GPU needs to know the global convergence, and two other scalar values. The *Barrier* class provided by C# threading namespace is used for its *SignalAndWait()* synchronization capability between threads. The *Synchronize* call in Fig. 6, implies that both the main thread and all GPU threads must reach that stage in order to proceed further.


GPU 1
GPU 2
GPU 1 needs pk data from these nodes
GPU 2 needs pk data from these nodes

(a)



(b)

Figure 5: GPU memory synchronization: (a) target synchronization data and (b) optimized synchronization data memory layout.

This allows the main thread to sum up all of the individual calculated dot products of each GPU and then return the resultant value to each GPU. In all cases, the GPU's wait until the global sum is returned, except when computing the convergence, as this value is only needed to know when to stop iterating. Obviously this results in a few extra computations being done in total, but it significantly improves GPU scaling. Please note that Fig. 6 only features the looped portion of the PCG algorithm. Each GPU has its local portions of the PCG vectors, with the exception of the pk vector containing extra import information from other GPU's. The local subscript implies that those dot products are the local GPU sums, while the subscript-less dot products are summed from the local GPU sums.

3.4 Algorithm Robustness Testing

To test the multi-GPU scaling performance, a variety of finite element models were tested under different setups. Table 2 depicts the scaling factor associated with each setup. Here, 0% scaling implies no change in performance, while 100% implies perfect GPU scaling without any losses. Please note that each individual device was tested for a optimum load split. Furthermore, even though a CPU is not a GPU, OpenCL enables various devices to be treated as computing platforms. The data presented in the table illustrates that scalability significantly enhances with the increasing size of the finite element system. This scalability is particularly

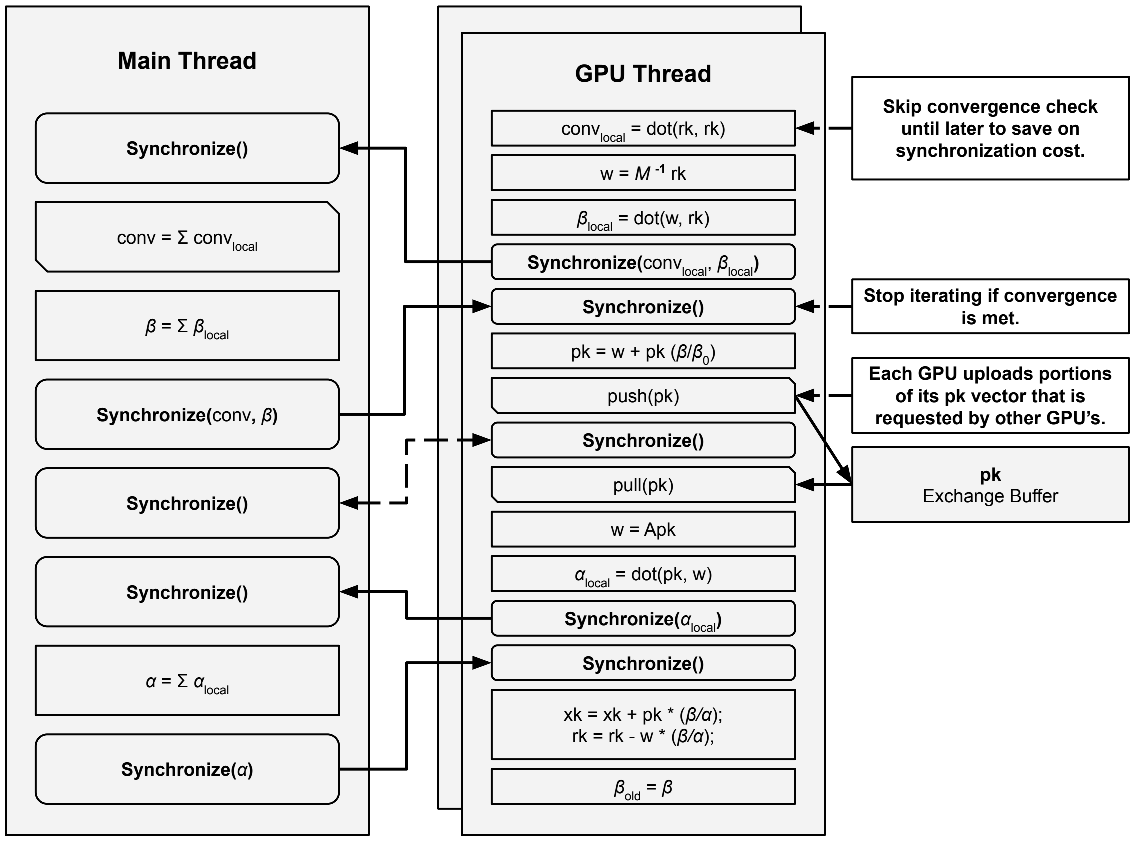


Figure 6: Overview of the multi-GPU synchronization

advantageous for large systems, which inherently require a longer time to solve compared to their smaller counterparts. An axial tensile test was ran as a benchmark in this scenario.

	GTX 1080 Ti	2x GTX 1080 Ti	Resultant Scaling
1.0M DOF	348	500	29.3%
5.0M DOF	83	143	72.9%
10M DOF	42	72	72.1%
100M DOF	4.5	8.7	93.3%
150M DOF	3.1	6.0	96.7%

Table 2: Multi-GPU iteration rates and resultant scaling (performance gain)

Further, the performance of the joint repositioning system is demonstrated in Table 3. These benchmarks were ran with a i9-10900KF CPU with a RTX 3060 Ti, or a GTX 1080 Ti with dual E5-2630 v4 CPU's. The AVX-256 solver was run on the i9-10900KF system. It is very interesting to see that the CPU solver edges out the GPU solvers at low resolutions. This is most likely due to the GPU initialization cost, and its corresponding data transfer. Since the tested repositioning system only had 19,710 degrees of freedom (DOF) at 5mm resolution, and the GPU solvers work best with larger models, it is well expected that the CPU outperforms at these low resolutions. The GTX 1080 Ti also had a significantly slower CPU, which contributed to its high mesh preparation cost.

	GTX 1080 Ti	**RTX 3060 Ti**	**AVX-256 CPU**
5 mm	489	186	124
4 mm	506	207	145
3 mm	685	279	322
2 mm	1315	578	1112

Table 3: Joint repositioning time (ms/loadstep)

4 APPLICATION-ORIENTED VALIDATION

Although the repositioning tool produces results that appear visually credible, various simplifications raise doubts about its precision. To assess the accuracy of the repositioning software, models that were repositioned virtually were evaluated against their real-world counterparts.

To achieve this, several individuals were scanned in a photogrammetry scanner adopting two distinct hand positions. The first position required a radially abducted position, while the second involved a palmar abduction of the thumb. After acquiring both scans, the acquired data was converted into 3D models and then aligned with each other. This alignment employed a standard point-to-point method, utilizing singular value decomposition to achieve a least squares best fit among points identified on both meshes. Following this alignment, the meshes were imported into the repositioning software, where virtual bones were assigned to the first mesh. These bones were then meticulously rotated to align with the second mesh, after which the joint repositioning software was used.

As the final step, a surface deviation map was generated and mean and maximum deviation values were extracted from it. Figures 7 and 8 depict the validation workflow. The protocol followed included the following steps:

1. Scan subject's hand with the thumb radially abducted (Fig. 7a)
2. Rescan the same hand with a palmar abduction (Fig. 7b)
3. Align the two resulting scans
4. Virtually reposition the other four fingers of the outwards extended thumb pose to match them with their counterparts from the inwards extension pose (Fig. 7c)
5. Determine surface deviation comparisons between the two scans

Comparisons performed between the hand scans of four different subjects revealed mean and maximum surface deviations of 0.87 mm and 5.10 mm, respectively (Tab. 4). With respect to accuracy, achieving large maximum deviations is possible because minor skin folds can easily increase deviations. Nonetheless, areas of minor displacement demonstrate exceptional precision, leading to minimal average surface deviations. Furthermore, qualitative visual inspections revealed that while small movements of the fingers led to relatively small deviations between the two scans, more significant repositionings of the thumb led to challenges caused by skin folds on the palmar side of the hand. Nevertheless, it is reasonable to conclude that the precision of

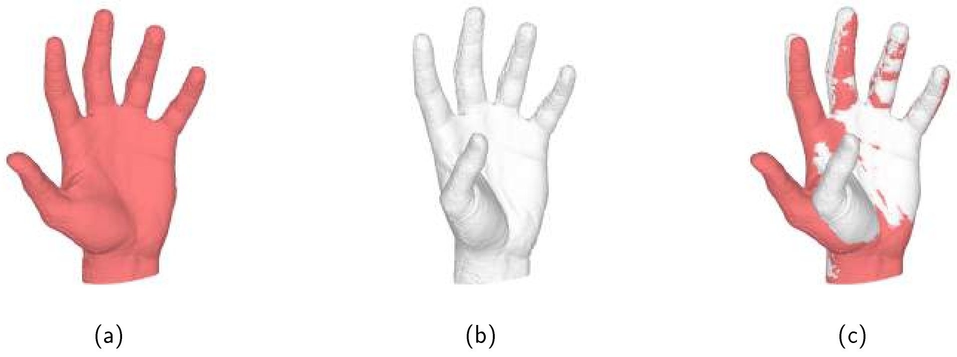


Figure 7: Determination of the surface deviation: (a) radial abduction, (b) palmar abduction and (c) aligned scans

the virtual repositioning remains within the acceptable limits. Additionally, any larger imperfections can be corrected by repositioning the skin on the palmar side of the hand. Figure 8 depicts graphical comparisons between virtually and physically repositioned thumb poses, both obtained from an initial one characterized by a radially abducted thumb.

	Mean abs. dev.	**Max. abs. dev.**
Subject 1	0.65	4.15
Subject 2	0.95	6.09
Subject 3	0.76	4.41
Subject 4	1.01	6.00
Subject 5	0.97	4.68
Average	**0.87**	**5.10**

Table 4: Surface deviation values between real and FEA predicted meshes.

5 CONCLUSIONS AND FUTURE WORK

The study presents a unique and innovative software tool capable to virtually reposition fingers. This operation is critical for patients characterized by reduced joint finger mobility that prevent the development of therapeutically effective hand splints. Since to the best of authors' knowledge no similar tools are commercially available, the significance of the tool developed in the context of the current study cannot be understated.

However, the strength of the developed software tool does not reside just in the uniqueness of the software solution proposed but also the large number of technical challenges that were solved through innovative approaches. More specifically, the software tool features several important advancements in GPU-accelerated FEA and mesh generation. One of the key innovations brought by the developed software tool include a GPU-based mesher that relies on a single contouring method optimized for smooth surfaces. This technique effectively balances accuracy and processing speed. The proposed approach enhances voxelization and mesh generation, particularly suitable for modeling non-angular forms like human hands. Furthermore, the software tool relies on a GPU FEA solver employing the Galerkin element formulation and a matrixless method to reduce memory usage and improve computational speed. To rapidly solve linear systems with millions of

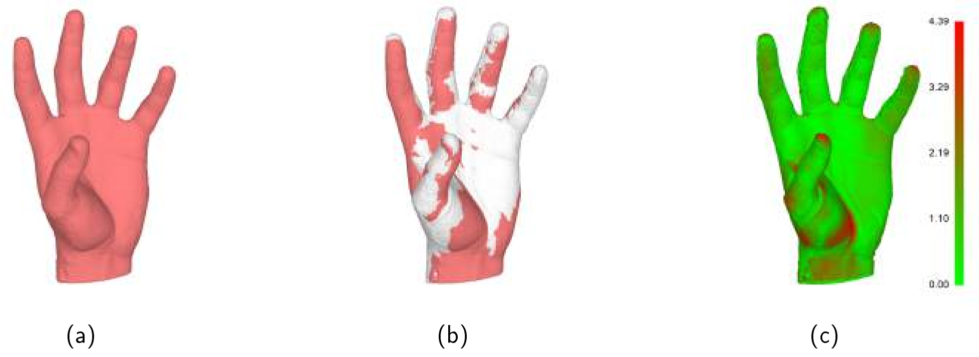


Figure 8: Comparison Approach: (a) Re-positioned Hand, (b) Overlaid and Re-positioned Hand (c) Surface Deviation Map

degrees of freedom, the tool employs the PCG method optimized through parallel reduction. Finally, multi-GPU optimization techniques are introduced for scalable FEA solving and they include efficient data exchange and load distribution among GPUs, leveraging OpenCL for broad hardware compatibility and integrating CPU fallback for memory-demanding models. All these innovative technical features enable an improved performance, accuracy, and scalability for the developed software tool. It is also noteworthy to emphasize that the software tool demonstrated robustness, versatility and accuracy when tested on extremely large models that were run on various hardware as well as on various hand geometries.

Future development avenues could include extension of this software to other types of orthoses, thereby expanding the utility of the tool to other areas of physiotherapy. Furthermore the algorithms developed in the context of this work hold great potential for repurposing in various scientific and entertainment sectors, such as accelerating finite element mesh generation from CT images or enhancing physical realism in video games. The multi-GPU solver's adaptability suggests that it could be further developed to address complex structural finite element systems, including micro-finite element models, thus offering a wider applicability in engineering and scientific research. These enhancements and expansions could significantly extend the software's impact across multiple disciplines, leveraging its accuracy, speed, and user-friendliness.

Since the approach selected relied on several simplifications in the finite element setup, a further need for a comparative testing of the accuracy yielded by Delaunay triangulation and voxel-based methods exists. Further, the incorporation of higher order elements could enhance the modeling of hand curvature. Future efforts should aim to refine the finite element setup to achieve more precise modeling of the human hand, building on previous studies' attempts at high-accuracy hand representation. Hyperelastic parameters could be used in order to better predict the deformation of the human skin [11, 12]. Future enhancements could involve replacing the simplified capsule-based bones with a detailed skeletal hand model, featuring parametric properties for adjustable bone lengths. In a less ideal scenario, simply scaling the skeleton bones could offer a rough approximation of bone structure.

Future updates could incorporate tendons and ligaments into the finite element model for enhanced realism. Currently, tendon representations can deform unrealistically during thumb repositioning. Modeling bone and ligament stiffness anisotropically with precise Young's moduli and Poisson's ratios could improve accuracy, though this presents challenges due to individual variations, such as age, affecting bone elasticity [22]. Improving skin contact modeling with collision checks between load steps could enhance accuracy, especially in areas with loose skin, where the current setup falls short. Using a simplified model for real-time preview and opting for a more accurate, time-intensive computation upon finalizing a repositioning could maintain software performance without compromising accuracy. Further exploration into single-precision arithmetic, possibly adopting a mixed precision system to balance speed and precision, could optimize iteration rates despite potential round-off errors [1]. Finding the right balance between single and double precision calculations is essential to maximize efficiency while minimizing errors.

ACKNOWLEDGEMENTS

This research was partially funded by the *Natural Science and Engineering Research Council of Canada* (NSERC) and Mitacs Canada.

Adam Gorski, https://orcid.org/0009-0004-4429-8256
O. Remus Tutunea-Fatan, https://orcid.org/0000-0002-1016-5103
Louis M. Ferreira, https://orcid.org/0000-0001-9881-9177

REFERENCES

[1] Buttari, A.; Dongarra, J.; Kurzak, J.; Luszczek, P.; Tomov, S.: Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. ACM Transactions on Mathematical Software (TOMS), 34(4), 1–22, 2008. ISSN 0098-3500. http://doi.org/10.1145/1377596.1377597.

[2] Choonara, Y.E.; du Toit, L.C.; Kumar, P.; Kondiah, P.P.D.; Pillay, V.: 3d-printing and the effect on medical costs: a new era? Expert Review of Pharmacoeconomics & Outcomes Research, 16(1), 23–32, 2016. http://doi.org/10.1586/14737167.2016.1138860.

[3] Cignoni, P.; Montani, C.; Perego, R.; Scopigno, R.: Parallel 3d delaunay triangulation. Computer Graphics Forum, 12(3), 129–142, 1993. http://doi.org/10.1111/1467-8659.1230129.

[4] Elteir, M.; Lin, H.; Feng, W.C.: Performance characterization and optimization of atomic operations on amd gpus. In 2011 IEEE International Conference on Cluster Computing, 234–243, 2011. http://doi.org/10.1109/CLUSTER.2011.34.

[5] Faieghi, M.; Knowles, N.K.; Tutunea-Fatan, O.R.; Ferreira, L.M.: Fast generation of cartesian meshes from micro-computed tomography data. Computer Aided Design and Applications, 16(1), 161–171, 2019. http://doi.org/10.14733/cadaps.2019.161-171.

[6] Gorski, A.; Tutunea-Fatan, O.R.; Ferreira, L.M.: Software tool for interactive design of customized hand splints. Computer-Aided Design and Applications, 21(6), 976–997, 2024. http://doi.org/10.14733/cadaps.2024.976-997.

[7] Gurumurthy, B.; Broneske, D.; Schäler, M.; Pionteck, T.; Saake, G.: An investigation of atomic synchronization for sort-based group-by aggregation on gpus. In 2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW), 48–53, 2021. http://doi.org/10.1109/ICDEW53142.2021.00016.

[8] Harih, G.; Tada, M.: Chapter 21 - development of a feasible finite element digital human hand model. In S. Scataglini; G. Paul, eds., DHM and Posturography, 273–286. Academic Press, 2019. ISBN 978-0-12-816713-7. http://doi.org/10.1016/B978-0-12-816713-7.00021-0.

[9] Inui, M.; Kaba, K.; Umezu, N.: Fast dexelization of polyhedral models using ray-tracing cores of gpu. Comput. Aided Des. & Appl, 18(4), 786–798, 2021. http://doi.org/10.14733/cadaps.2021.786-798.

[10] Knowles, N.K.; Kusins, J.; Columbus, M.P.; Athwal, G.S.; Ferreira, L.M.: Morphological and apparent-level stiffness variations between normal and osteoarthritic bone in the humeral head. Journal of Orthopaedic Research®, 38(3), 503–509, 2020. http://doi.org/doi.org/10.1002/jor.24482.

[11] Lapeer, R.; Gasson, P.; Karri, V.: Simulating plastic surgery: From human skin tensile tests, through hyperelastic finite element models to real-time haptics. Progress in Biophysics and Molecular Biology,

103(2), 208–216, 2010. ISSN 0079-6107. http://doi.org/10.1016/j.pbiomolbio.2010.09.013. Special Issue on Biomechanical Modelling of Soft Tissue Motion.

[12] Lapeer, R.J.; Gasson, P.D.; Karri, V.: A hyperelastic finite-element model of human skin for interactive real-time surgical simulation. IEEE Transactions on Biomedical Engineering, 58(4), 1013–1022, 2010. http://doi.org/10.1109/TBME.2009.2038364.

[13] Lee, D.T.; Schachter, B.J.: Two algorithms for constructing a delaunay triangulation. International Journal of Computer & Information Sciences, 9(3), 219–242, 1980. http://doi.org/10.1007/BF00977785.

[14] Lopes, P.C.F.; Pereira, A.M.B.; Clua, E.W.G.; Leiderman, R.: A gpu implementation of the pcg method for large-scale image-based finite element analysis in heterogeneous periodic media. Computer Methods in Applied Mechanics and Engineering, 399, 115276, 2022. ISSN 0045-7825. http://doi.org/10.1016/j.cma.2022.115276.

[15] Lorensen, W.E.; Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. In Seminal graphics: pioneering efforts that shaped the field, 347–353. Association for Computing Machinery, 1998. http://doi.org/10.1145/280811.281026.

[16] Mathur, A.B.; Collinsworth, A.M.; Reichert, W.M.; Kraus, W.E.; Truskey, G.A.: Endothelial, cardiac muscle and skeletal muscle exhibit different viscous and elastic properties as determined by atomic force microscopy. Journal of Biomechanics, 34(12), 1545–1553, 2001. ISSN 0021–9290. http://doi.org/10.1016/S0021-9290(01)00149-X.

[17] Nielson, G.: Dual marching cubes. In IEEE Visualization 2004, 489–496, 2004. http://doi.org/10.1109/VISUAL.2004.28.

[18] Ramirez-Salazar, J.F.; Mesa-Munera, E.; Bedoya, J.W.B.; Boulanger, P.: Comparison between lagrange multiplier and penalty methods to enforce essential boundary conditions in meshfree methods. Avances en Sistemas e Informática, 8(3), 51–56, 2011.

[19] Saint-Georges, P.; Warzée, G.; Beauwens, R.; Notay, Y.: High-performance pcg solvers for fem structural analysis. International journal for numerical methods in engineering, 39(8), 1313–1340, 1996. http://doi.org/10.1002/(SICI)1097-0207(19960430)39:8<1313::AID-NME906>3.0.CO;2-J.

[20] Straub, J.; Kerlin, S.: Development of a large, low-cost, instant 3d scanner. Technologies, 2(2), 76–95, 2014. http://doi.org/10.3390/technologies2020076.

[21] Weyler, R.; Oliver, J.; Sain, T.; Cante, J.: On the contact domain method: A comparison of penalty and lagrange multiplier implementations. Computer Methods in Applied Mechanics and Engineering, 205–208, 68–82, 2012. ISSN 0045-7825. http://doi.org/10.1016/j.cma.2011.01.011. Special Issue on Advances in Computational Methods in Contact Mechanics.

[22] Zioupos, P.; Currey, J.: Changes in the stiffness, strength, and toughness of human cortical bone with age. Bone, 22(1), 57–66, 1998. ISSN 8756–3282. http://doi.org/10.1016/S8756-3282(97)00228-7.

[23] Zucca, R.; Santos, R.C.; Lovatto, J.; Cesca, R.S.; Lovatto, F.: Development of a low cost 3d printer indicated to prototyping objects. Semina: Ciencias Exatas e Tecnologicas, 40(1), 47–54, 2019. http://doi.org/10.5433/1679-0375.2019v40n1p47.