Pacific Graphics (2015)
N. J. Mitra, J. Stam, and K. Xu (Editors)

Parallel Importing of OBJ Meshes in CUDA

Aidan. L. Possemiers and Ickjai Lee

Information Technology Academy
College of Business, Law and Governance
James Cook University, PO Box 6811, Cairns, QLD 4870, AUSTRALIA

Abstract

Alias | Wavefront OBJ meshes are a common text file type for transferring 3D mesh data between applications made by different vendors. However, as the mesh complexity gets higher and denser, the files become larger and slower to import. This paper explores the use of GPUs to accelerate the importing and parsing of OBJ files by studying file read-times, runtimes and load resistance. We propose a new method of reading and parsing that circumvents GPU architecture limitations and improves performance, seeing the new GPU method outperform CPU methods with a 6-8x speedup. When run on a heavily loaded system, the new method only received an 80% performance hit, compared to the 160% that the CPU methods received. The loaded GPU speedup compared to unloaded CPU methods was 3.5x, and, when compared to loaded CPU methods, 8x. These results demonstrate that the time is right for further research into the use of data-parallel GPU acceleration beyond that of computer graphics and high performance computing.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors—I.3.1 [Computer Graphics]: Parallel processing—I.3.6 [Computer Graphics]: Graphics data structures and data types—

1. Introduction

Graphics Processing Units (GPUs) have seen a lot of interest, outside of their original purpose of rendering computer graphics, as they offer considerable computation speed ups over their CPU counterparts in particular use cases [NHP07, RDV*12, SHUS10]. While research into N-body simulations, Global Illumination, fluid dynamics and other 'exciting' simulations have drawn the majority of the attention [NHP07, RDV*12, SHUS10]; this paper focuses on the more 'mundane' elements of programming, such as file importing and parsing, to show that these, too, can take advantage of the modern GPU and their impressive potential for parallelization. The area of importing and parsing has seen relatively little interest as GPU architecture and runtime differences mean that algorithms either are unsuitable, or require heavy rework to see any marginal speedup. There has been research into natural language parsing [HBKCK14, Joh11] and integrating the GPU into the file system under Linux [SFKW14] but the closest related research has been limited to optimizing and running queries in SQL or on data stored in XML [BS10, SYH*11]. GPU hardware, however, has not been neglected and, with the demand for higher performance and higher resolution devices, it is very difficult these days to find a device that does not have some form of integrated GPU -from cell phones to automobiles- and it is time to start using this untapped resource (`http://www.nvidia.com/object/cuda_home_new.html`).

The concept of General-purpose Programming on the Graphics Processing Unit (GPGPU) is not a new one but neither is it a solved problem. It is often that, to make an algorithm run fast on the GPU, one has to re-invent said algorithm, conversely, sometime tasks are 'embarrassingly parallel', such that minimal change is necessary. In this research we look at a task (importing an OBJ mesh to OpenGL) and investigate how a very linear task on the CPU, can be re-written so that it can take advantage of the data-driven parallelization that the GPU provides.

We chose to use the Alias | Wavefront OBJ file type as it is an open format, generally accepted as universal, and used in: engines, development tools and simulations, unlike binary files which can be software and platform specific. While we focus on this small edge-case the minor differences between the text-based file types means that STL or PLY could also be similarly implemented for the GPU.

DOI: 10.2312/pg.20151283

2. Preliminaries

The potential of a 60x or more speed up creates a lot of excitement about GPGPU, though the limitations the GPU hardware and architecture imposes often means, without heavy modification, most algorithms will actually run slower on a GPU. While GPUs are capable of running millions of threads at the same time, the actual clock speeds can be magnitudes slower than their CPU counterparts. There is also the effect of the underlying architecture: while a CPU is task parallel, a GPU is data parallel or, more precisely, Kernel parallel. A version of Single Instruction, Multiple Data (SIMD), the GPU contains multiple processing cores that perform the same operation on multiple pieces of data all in parallel: hence data parallel [KMMS10]. This not only means that CPU algorithms, but also multi-core algorithms, cannot run on a GPU without modification. Assuming the task is data parallel, dynamic memory allocation is also a heavy overhead as it requires global synchronization. While there have been attempts to circumvent this limitation [GPK*12, DWL*12], the general consensus is to pre-allocate memory. This limits the use cases; either memory has to be over allocated, assuming the worst case given the data, or the number of return values has to be already known and pre-allocated.

Amdahl's law [Hea15, HM08] is another big hurdle for GPGPU and parallel processing in general, which demonstrates that the potential speedup of a linear algorithm on a fixed problem size, as the algorithm is made more parallel and run on more cores. While it ignores costs like memory overhead and data transfer rate -which benefits GPUs as these are expensive for it to perform- the law is considered a double edged sword: it stipulates that as the number of cores increases there is a diminishing performance return limited by the percentage of the code that is run in serial. For example, if the serial fraction of code exceeds 1%, the speed up can never exceed 100x, no matter how many processors are used [Hea15, HM08]. Taking Amdahl's law into account for data parallel, GPGPU programming means coming up with more inventive ways to parallelize serial code sections as often applications use task parallelization to create a speed up.

CUDA is NVIDIA's foray into making GPGPU programming more accessible by extending C/C++ to take advantage of their GPU architecture (`http://www.nvidia.com/object/cuda_home_new.html`). It works by splitting code into functions to run on either the host; the CPU and system memory, or the device; the current GPU that the CUDA context is running on. Host code resembles C/C++ and is compiled by the native C/C++ compiler other than when it calls device code, or uses CUDA functions, in which it has to be compiled under NVIDIAs CUDA compiler. Device code or 'Kernels' resemble C/C++ as well but with certain functionalities, like realloc, missing, due to the GPU architecture and instruction set differences. Though with each new version of CUDA more and more C++ features are added.

Kernel functions are run in parallel by blocks of threads, with a maximum of 1024 threads per block on a device with compute capability 2.0+. These thread blocks are run on in grid of a maximum size of 231 - 1 blocks in the x direction with compute capability 3.0+ [GPK*12]. Blocks are processed by stream multiprocessors with threads processed in warps of 32 parallel threads with the warp scheduler picking which warp in a block to be executed. A grid can be launched in one dimension and a thread's global index is found by adding its thread index inside the block in the x direction, to its block index multiplied by its block dimension, both in the x direction. This one dimensional thread *id* is used to access relevant information from the GPUs memory, such as the particular element in an array that is to be acted upon by this thread. By accessing and writing to memory in this manner we are practicing memory coalition within our warps. Memory coalition is a high priority CUDA 'best practice' (`http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`), as mentioned before, each thread runs the same operation, and accessing concurrent memory in a warp is necessary to take full advantage of the architecture. Avoiding branch divergence is another best practice, and occurs when there is a decision statement like if or switch. Only threads that share the same path are executed synchronously, with the other paths running after the first path has been finished or a barrier is met. It can be avoided by having branches logically occur on separate warps, avoiding the diverged branches having to be run separately.

Thrust is a C++ template library for CUDA, based on the Standard Template Library (STL) (`https://developer.nvidia.com/Thrust`). Thrust provides access to two vector templates: one that stores data on the GPU or device, and another that stores it in system memory or the host. These generic containers allow simple transfer between the two memory locations, however, the real strength of the Thrust library comes with access to simple, yet powerfully parallel algorithms: Count, Sort, Scan, Reduce, Remove and Unique. These algorithms, when combined with our context-specific predicate functions, and other custom parallel code, can be used to simply and easily circumvent traditionally linear code section. We chose to use the Alias | Wavefront OBJ file type as it is an open format, generally accepted as universal, and used in: engines, development tools and simulations. Unlike PLY or STL, OBJ files store 3D mesh data as a series of single line elements prefixed by a character sequence: '#' for human readable commenting; 'v' for vertex coordinates; 'vt' for texture coordinates; 'vn' for vertex normal vector; and finally 'f' for the draw ordered indices of the other arrays that are used to build the triangles of the mesh in 3D [Mv96]. Importing OBJs on the CPU is traditionally very linear. As Algorithm 1 shows its broken into 3 stages: read the file line by line and parse the data into temporary vectors, pack unique vertices into dictionary and store draw ordered indices, unpack vertices and pass vertex coordinate, UV coordinate, normal vector and index arrays' to OpenGL.

3. Framework

3.1. Import

To begin the import, first the text file must be passed to the GPUs memory. This is a major hurdle that must be overcome as the GPU itself does not have access to the hard drive nor does it have any way to access the file system in the same manner as the CPU does. Because the data must be read into the system memory by the CPU: this creates a bottleneck as it takes time for the CPU to read the file, during which time, the GPU is sitting idle with nothing to process. We reduce this overhead by reading the data in chunks. As the current chunk is read by the CPU, the GPU searches for delimiting characters in the previously read chunk and records their position, per character, in parallel. This works fine if the GPU delimits the chunk faster than the CPU can read them. However, if the file is pre-buffered, due to O/S caching or the GPU model is simply not powerful enough, the GPU section will cause a bottleneck, with the CPU idling, waiting to read in the next chunk. To prevent this the GPU kernel is fired off from a separate CPU thread. The main CPU thread checks if the thread running the GPU kernel is finished, if not, the CPU will read in more data while it waits. This minor tweak creates a load balancing effect that takes into account different hardware configurations and bus speeds, as well as other elements that could un-balance the two processes.

3.2. Parsing and Indexing

As show in Figure 1, once the full file has been read to GPU memory, it is parsed, in parallel, into an array of interim objects; this section of the code is 'embarrassingly parallel', as the information in each line, at this stage, is un-reliant on any other piece of information as long as order is preserved. This also circumvents the GPU issue, in the lack of fast dynamic memory allocation, as we know the total number of elements from the delimitation step, yet not how many of each element type. These 'proxies' store the data type (in the OBJ case this is vertex, UV, normal, face and unknown) and has memory allocated equivalent to nine 32-bit integers (the amount needed to store indices for a single triangle) for the parsed values to be stored in binary format. If there is any undesired information, like comments, it is tagged as unknown and removed using Thrust's remove. If the file is formatted correctly in blocks of single data types, there is minimal warp branch divergence, as the kernel only diverges at the end of each block of types.

The face elements are then used to build an array of packed vertices that are a raw, draw order representation of the mesh. Each PackedVertex object holds the vertex's 3D coordinate, 2D texture coordinate and normal vector as well as its draw index: its current index in this array. The final

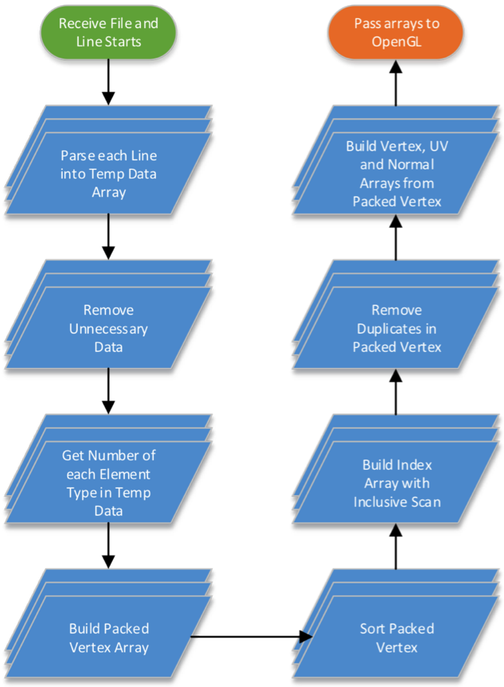
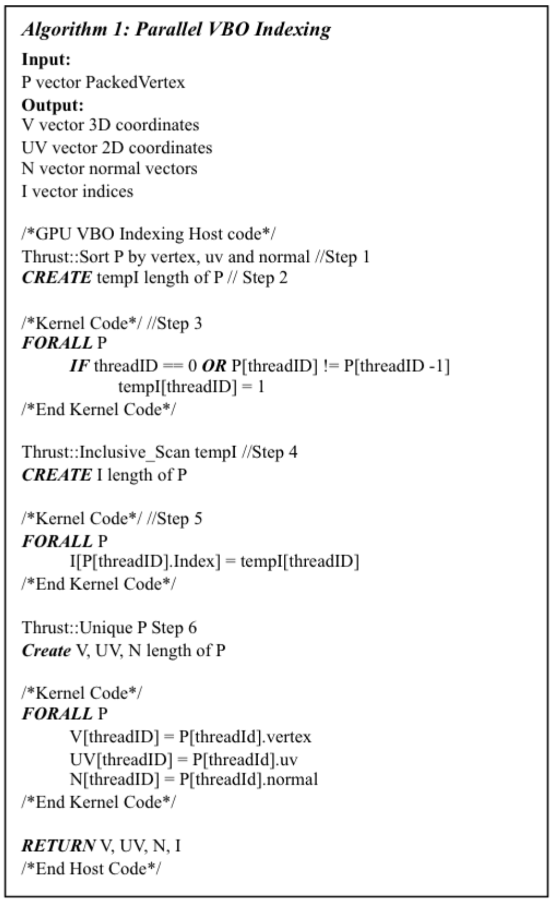

Receive File and Line Starts
Parse each Line into Temp Data Array
Remove Unnecessary Data
Get Number of each Element Type in Temp Data
Build Packed Vertex Array
Sort Packed Vertex
Build Index Array with Inclusive Scan
Remove Duplicates in Packed Vertex
Build Vertex, UV and Normal Arrays from Packed Vertex
Pass arrays to OpenGL

Figure 1: *Parse and Index stage.*

act of the import, before the arrays are passed to OpenGL for rendering, is to remove the duplicated data by removing created when triangles share common vertices. To display the mesh correctly, even if vertices share the same 3D coordinate, their normal vector may be different to allow for a sharp/smooth edge, or their 2D coordinate might be different to optimize texture space. Figure 2 outlines the process of parallel VBO indexing with the vector of PackedVertices as input, and outputs the vectors to be passed as arrays to OpenGL. The code shows branching parallelism, as the whole block is run from the CPU which fires off the Kernel Code sections which run on the GPU. While the Kernels are running, the main thread waits for them to finish then continues to the next section. Figure 4 shows this process visually, by breaking down each step of our algorithm. PackedVertices are represented by capital letters with their draw order index, with duplicates using the same letter.

Step 1, we sort the data so that identical vertexes are clustered together in a Thrust parallel sort, which has a complexity of $O(n \log n)$. The predicate function we use to sort, looks at each value of each element, and uses strict weak ordering, first by 3D coordinate, then 2D coordinate and finally normal vector, to give us the clusters of duplicates. As we have already stored the original index of each PackedVertex, we donâĂŹt need to preserve order. Step 2, we create an array of unsigned integers the same size as the sorted packed ver-



```
Algorithm 1: Parallel VBO Indexing
Input:
P vector PackedVertex
Output:
V vector 3D coordinates
UV vector 2D coordinates
N vector normal vectors
I vector indices

/*GPU VBO Indexing Host code*/
Thrust::Sort P by vertex, uv and normal //Step 1
CREATE tempI length of P // Step 2

/*Kernel Code*/ //Step 3
FORALL P
    IF threadID == 0 OR P[threadID] != P[threadID -1]
        tempI[threadID] = 1
/*End Kernel Code*/

Thrust::Inclusive_Scan tempI //Step 4
CREATE I length of P

/*Kernel Code*/ //Step 5
FORALL P
    I[P[threadID].Index] = tempI[threadID]
/*End Kernel Code*/

Thrust::Unique P Step 6
Create V, UV, N length of P

/*Kernel Code*/
FORALL P
    V[threadID] = P[threadId].vertex
    UV[threadID] = P[threadId].uv
    N[threadID] = P[threadId].normal
/*End Kernel Code*/

RETURN V, UV, N, I
/*End Host Code*/
```

Figure 2: *Parallel VBO indexing algorithm.*

tices array. This will eventually be used to build the OpenGL index buffer. Step 3, in parallel, we assign 1 to the element that shares the same index as the first packed vertex in each cluster of duplicates: a complexity of $O(n)$. Step 4, we run a Thrust inclusive scan on the new array, which sets the value of an element to the sum of all previous elements: another complexity of $O(n)$. Step 5, we use the original draw index of the packed vertex to reorder the integer array, which gives us the index array of the unique packed vertices to pass to an OpenGL index buffer object. This step is once again 'embarrassingly parallel', and as such has a complexity of $O(n)$. Step 6 shows the unique packed vertex array. This array is then split into 3 separate arrays of vertex coordinates, texture coordinates, and normal vectors, and along with the indexed array, are passed to OpenGL. Over all complexity of the indexing process is $O(n \log n)$ running in parallel.

4. Methodology

Getting the mesh data quickly into OpenGL for rendering, or modification, is the primary driving force of this research. With that in mind, overall speed of the import and parse is of the most importance. There are, however, considerations to be made as the hardware specifications differences shown in Table 1 between the CPU and GPU, make a direct implementation comparison difficult. While purely comparing parse times of the two systems would be ideal -this would always lead to a distinct GPU advantage due to most of the problem being 'embarrassingly parallel'- the real world limitations of GPGPU is the cost of memory transfers, and arranging data in a manner that GPUs can process. Therefore all the tests observe the total runtime, from reading the file from a hard drive, to outputting the final arrays for OpenGL.

Using the total runtime also allows for a wider set of sample cases. 3D modelling applications: Autodesk Maya (`http://www.autodesk.com.au/products/maya/overview`), and MeshLab (`http://meshlab.sourceforge.net/`), both output their total import times for OBJ meshes. Both these applications are complex systems that do much more than simply import a mesh so -the open source- Tiny OBJ Loader was also used as a direct code to code comparison (`http://syoyo.github.io/tinyobjloader/`). The last limitation accounted for was the operating system caching the file after it was first imported. Initial testing showed great time variances for all methods but, only on the first import of a particular file. We determined that this was due to caching so all data gathered after insured that the file was cached first. The mesh used, Asian Dragon, was sourced from Stanford's 3D scanning repository provided generously by XYZ RBG Inc. (`http://graphics.stanford.edu/data/3Dscanrep/`).

4.1. Import Test

The mesh was first taken into Pixologic's Zbrush and decimated at intervals of 10%. Table 1 shows the breakdown for each of the meshes and how the decimation level effects the total number of elements (lines) needed to parse, in relation to the number of triangles to render. The overall import speed tests were run 11 times for each application at each decimation level, with the first result discarded to account for O/S caching, as eliminating its effect was found to be impossible.

4.2. Under Load Test

In the second experiment, we ran the tests, but only for the un-decimated mesh (100% in Table 1) but this time with a CPU loading application in the background, flooding all cores with a normal priority process to test the robustness of each application.

4.3. GPU Section Test

The final experiment compared the GPU method on 2 different GPUs. Looking at the time that the reading section

Table 1: *Mesh breakdown.*

% decimated	Lines in file	Triangles to render
10%	3,248,494	721,886
20%	6,497,011	1,443,778
30%	9,745,561	2,165,668
40%	12,994,030	2,887,560
50%	16,242,535	3,609,450
60%	19,491,049	4,331,342
70%	22,739,554	5,053,232
80%	25,988,068	5,775,124
90%	29,236,573	6,497,014
100%	32,485,087	7,218,906

and parsing/indexing section take and comparing these values while running on 2 different pieces of GPU hardware. Due to the memory limitations of a lower end card -with re-writing the code to use a buffer was determined to be out of scope- the experiments were run 11 times on the mesh decimated to 60%: the first results were again, discarded.

5. Results

In parallel parsing the mesh runs between 5-8x faster under CUDA on the GPU, than it does sequentially in C++ on the CPU. Figure 6 shows the overall runtimes of each method, with the GPU methods running considerably faster than the CPU methods. Autodesk Maya, one of the most widely used 3D applications, unsurprisingly, ran consistently faster than the other two CPU applications. As there is no source code available for its OBJ importing code, it is difficult to tell if this is due to some CPU multithreading, or just a more efficient data structure behind the scenes.

The open source Tiny OBJ Loader runs on a single thread, and follows a similar algorithm to the one in Algorithm 1. It is import times were almost exactly in between Maya, and MeshLab, which ran the slowest, and as such, taking into account the Maya optimizations, serves as the code to code comparison. The GTX 970 with its 4GB of memory, easily fit both the large mesh file and the parsed arrays in memory in all cases. However, the GTX 750 with 2GB of memory, could only handle up to the 60% mesh before running out. Excluding the memory limitation the GTX 750, while slower than the GTX 970, has almost half the total power draw of the tested CPU, and yet, for this case, outperformed the CPU methods by a factor of 5x.

When running the tests again on a loaded system (Figure 3) the speedup between the loaded GPU and the loaded CPU method was the same as the unloaded speed up. The GPU method, with an approximate 3x speedup, still ran faster on a loaded CPU system than the CPU method running on an unloaded system.

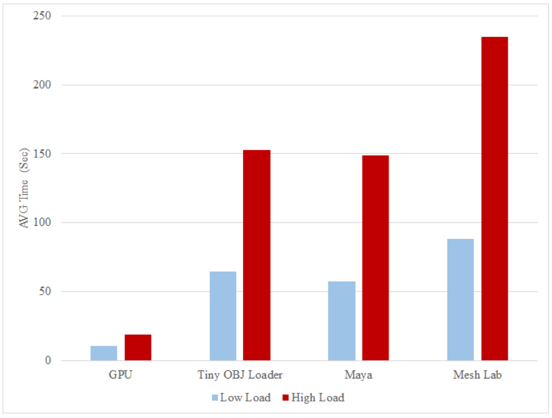


Figure 3: *Load comparison.*

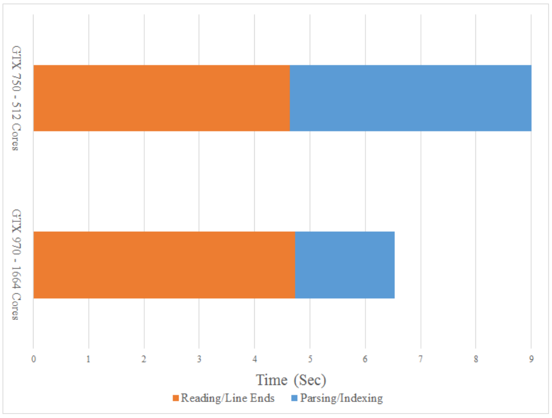


Figure 4: *GPU comparison.*

Figure 4 shows us a glimpse of Amdahl's law in effect by comparing GPU times, with minor micro benchmarking to split the read and parse times from the file import time,. The GTX 970 may have a slightly faster core clock speed, but it has 3x the number of CUDA cores however, the major performance differs only in the highly parallel parse and index section.

Figure 5 shows the percentage speedup that the GTX 970 has over the three different CPU methods as well as an average. This average speedup is almost always aligned with the speedup compared with that of Tiny OBJ Loader.

6. Conclusion

In this paper we have shown that GPUs are capable of parsing OBJ files upwards of 5x faster than current CPU methods. This is achieved by creating algorithms that take advantage of the strengths of the GPU, while avoiding their weaknesses. By comparing our method with applications (Maya,

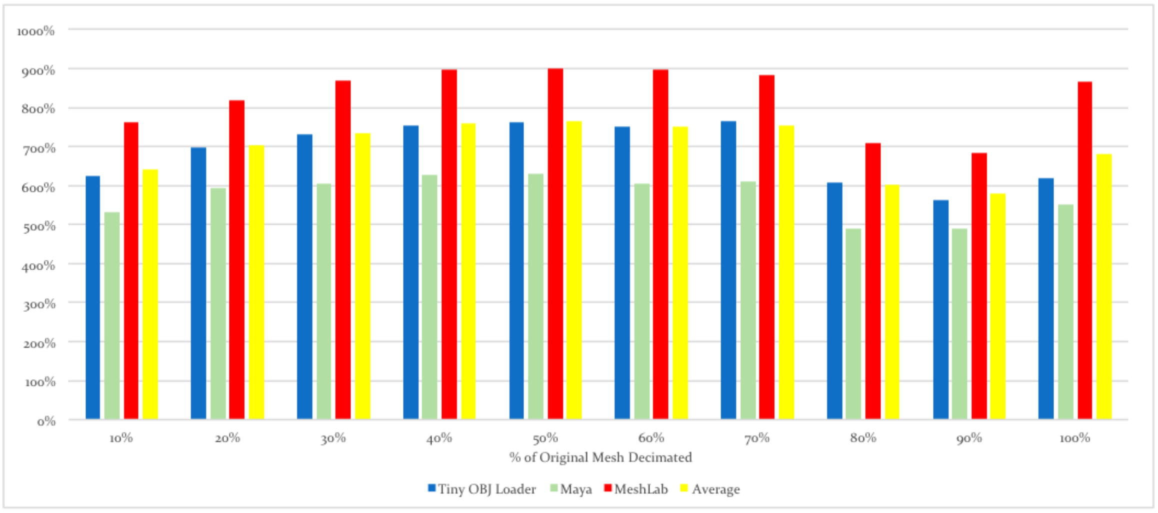


Figure 5: *GTX 970 speedup per mesh comparison.*

MeshLab, as well as Tiny OBJ Loader) we provide strong evidence to this case.

Though we see significant speed ups, there is more work to be done, especially in the read areas of our code. Understanding the effect of Amdahl's law shows us that the read section of our code requires further parallelization for the whole system to benefit. Lack of direct access to the file system makes this very difficult, however, there are several researched options worth investigating. GPUfs [SFKW14] or Direct Memory Access with GPUDirect, both under Linux, are very interesting, as then, GPU thread blocks could be used to both read and parse, cutting down on serial sections, so that as cores increase so would performance. Beyond simply speeding up the current use case, this method could be applied to other different data types. By adding support for hierarchical data at parse time, other 3D file types like FBX, Maya ASCI, STL and PLY. Beyond just 3D data, Comma Separated Values would be easily read by this system as it is currently implemented. Hierarchical data handling JDON and XML data could also be parsed in much the same way as the other 3D data types.

References

[BS10] Bakkum P., Skadron K.: Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General Purpose Computation on Graphics Processing Units* (2010), ACM Press, pp. 94–103. 1

[DWL*12] Du P., Weber R., Luszczek P., Tomov S., Peterson G., Dongarra J.: From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing 38*, 8 (2012), 391–407. 2

[GPK*12] Ghorpade J., Parande J., Kulkarni M., , Bawaskar A.: Gpgpu processing in cuda architecture. *Advanced Computing: An International Journal 3*, 1 (2012), 105–120. 2

[HBKCK14] Hall D., Berg-Kirkpatrick T., Canny J., Klein D.: Sparser, better, faster gpu parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics* (Baltimore, MD, USA, 2014), vol. 1, pp. 208–217. 1

[Hea15] Heath M.: A tale of two laws. *International Journal of High Performance Computing Applications* (2015). 2

[HM08] Hill M., Marty M.: Amdahl's law in the multicore era. *Computer 41*, 7 (2008), 33–38. 2

[Joh11] Johnson M.: Parsing in parallel on multiple cores and gpus. In *Proceedings of the Australasian Language Technology Association Workshop* (2011), pp. 29–37. 1

[KMMS10] Keutzer K., Massingill B. L., Mattson T. G., Sanders B. A.: A design pattern language for engineering (parallel) software. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns* (2010), vol. 13, p. 9. 2

[Mv96] Murray J. D., vanRyper W.: *Encyclopedia of Graphics File Formats*. O'Reilly Media, 1996. 2

[NHP07] Nyland L., Harris M., Prins J.: Fast n-body simulation with cuda. *GPU Gems 3*, 1 (2007), 677–696. 1

[RDV*12] Rinaldi P., Dari E., Vénere M., , Clausse A.: A lattice-boltzmann solver for 3d fluid simulation on gpu. *Simulation Modelling Practice and Theory 25* (2012), 163–171. 1

[SFKW14] Silberstein M., Ford B., Keidar I., Witchel E.: Gpufs: Integrating a file system with gpus. *ACM Transactions on Computer Systems 32*, 1 (2014), 1. 1, 6

[SHUS10] Stone J., Hardy D., Ufimtsev I., Schulten K.: Gpu-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling 29*, 2 (2010), 116–125. 1

[SYH*11] Si X., Yin A., Huang X., Yuan X., Liu X., Wang G.: Parallel optimization of queries in xml dataset using gpu. In *Fourth International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)* (2011), IEEE Computer Society, pp. 190–194. 1