Computational Visual Media
DOI 10.1007/s41095-015-0021-5 Vol. 1, No. 3, September 2015, 229–238

Research Article

Fast OBJ file importing and parsing in CUDA

Aidan L. Possemiers[1] (✉), **Ickjai Lee**[1]

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-time, runtime, 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 outperforms CPU methods with a 6×–8× speedup. When running 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 3.5×, and, when compared to loaded CPU methods, 8×. 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.

Keywords parsing; OBJ; vertex buffer object (VBO); general-purpose programming on the graphics processing unit (GPGPU); compute unified device architecture (CUDA)

1 Introduction

Graphics processing units (GPUs) have seen a lot of interest, outside their original purpose of rendering computer graphics, as they offer considerable computation speedups over their CPU counterparts in particular use cases [1–3]. While research into N-body simulations, global illumination, fluid dynamics, and other exciting simulations have drawn the majority of the attention [1–3]; 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 [4, 5] and integrating the GPU into the file system under Linux [6], but the closest related research has been limited to optimizing and running queries in SQL or on data stored in XML [7, 8]. 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 any 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 ready availability of these GPU embedded devices means that when programming we can now remove the assumption that a GPU might not be available, and that they are only limited to rendering graphics. Parallel optimized search, sort and reduction algorithms can run 30 or more times faster than their linear counterparts (https://developer.nvidia.com/Thrust).

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

1 James Cook University, PO Box 6811, Cairns, QLD 4870, Australia. E-mail: A. L. Possemiers, aidan.possemiers@jcu.edu.au (✉); I. Lee, ickjai.lee@jcu.edu.au.

Manuscript received: 2015-08-25; accepted: 2015-08-29

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 mean that STL or PLY could also be similarly implemented for the GPU.

Our contributions include:

- basic asynchronous reading with parallel element delimiting;
- element indexed proxy structures with parallel element parsing;
- fast vertex buffer object (VBO) indexing through Thrusts parallel removal, sort, and scan as well as custom parallel functions.

2 Preliminaries

GPGPU has been around since the support of floating point numbers and shaders on the GPU in 2001 [9]. NVIDIA's compute unified device architecture (CUDA) removed the layer of abstraction by replacing graphics related concepts like textures and pixels with more familiar concepts like threads, vectors, and arrays, and since its release in 2007 other alternatives such as OpenCL and direct compute have emerged (http://www.nvidia.com/object/cuda_home_new.html). The greatest strength of GPGPU over regular CPU code is the massive parallelization that the hardware allows with particular use cases toting 60× or more speedups over their CPU counter parts [1–3].

While the potential of a 60× or more speedup creates a lot of excitement about GPGPU, the reality is the limitations the GPU imposes often mean, without heavy modification, most linear 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. This is a version of single instruction, multiple data (SIMD) as the GPU contains multiple processing cores that perform the same operation on multiple data all in parallel: hence data parallel [10]. 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 [9, 11], the general consensus is to pre-allocate memory. This limits the use cases; either memory has to be over allocated, assuming the worst case, or the number of return values has to be already known and pre-allocated.

Amdahl's law [12, 13] is another big hurdle for GPGPU and parallel processing in general, which demonstrates 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 speedup can never exceed 100×, no matter how many processors are used [12, 13]. 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 speedup.

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). CUDA works by splitting the code into two sections: host code, that runs on the CPU and system memory; and device code, that runs on whichever GPU the current CUDA context is using. 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 NVIDIA's 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 $(2^{31} - 1)$ blocks in the x direction with compute capability 3.0+ [11]. 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 threadID is used to access relevant information from the GPU's 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 the other 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 [14]. Figure 1 shows how a single triangle might be stored in this file type.

Importing OBJs on the CPU is traditionally very linear. As Algorithm 1 shows, it is 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 Importing

To begin the import, first the text file must be passed to the GPU's 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

```
# this is a comment
v 0.000000 0.000000 0.000000
v 1.000000 1.000000 0.000000
v 0.000000 0.000000 1.000000
vt 0.000000 0.000000
vt 0.000000 1.000000
vt 1.000000 0.000000
vn 1.000000 1.0000000 1.000000
vn 1.000000 0.5000000 1.000000
vn 1.000000 1.0000000 1.000000
f 1/1/2 2/2/3 3/3/1
```

Fig. 1 OBJ file sample.

Algorithm 1: Serial OBJ reading on the CPU
Input: OBJ file.
Output: V vector of 3D coordinates,
UV vector of 2D coordinates,
N vector of normal vectors,
I vector of indices.

```
/*CPU OBJ Read*/
  for each line in OBJ file do
    if line == vertex then
      Parse line as vertex
      append vertex to tempV
    else if line == uv then
      Parse line as uv
      append uv to tempUV
    else if line == normal then
      Parse line as normal
      append uv to tempNormal
    else if line == face then
      Parse line as face
      Use face indices to build packedVertex from tempV,
      tempUV and tempN
      if dictionary contains packedVertex then
        append index to I
      else
        append dictionary.size to I
        add packedVertex to dictionary
      end if
    end if
  end for

  for each packedVertex in dictionary do
    append packedVertex.vertex to V
    append packedVertex.uv to UV
    append packedVertex.normal to N
  end for

  return V, UV, N, I
```

CPU does. 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.

Figure 2 shows our method of speeding up the overall system by reading the data in chunks. As the current chunk is read, 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 CPU idling, 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.

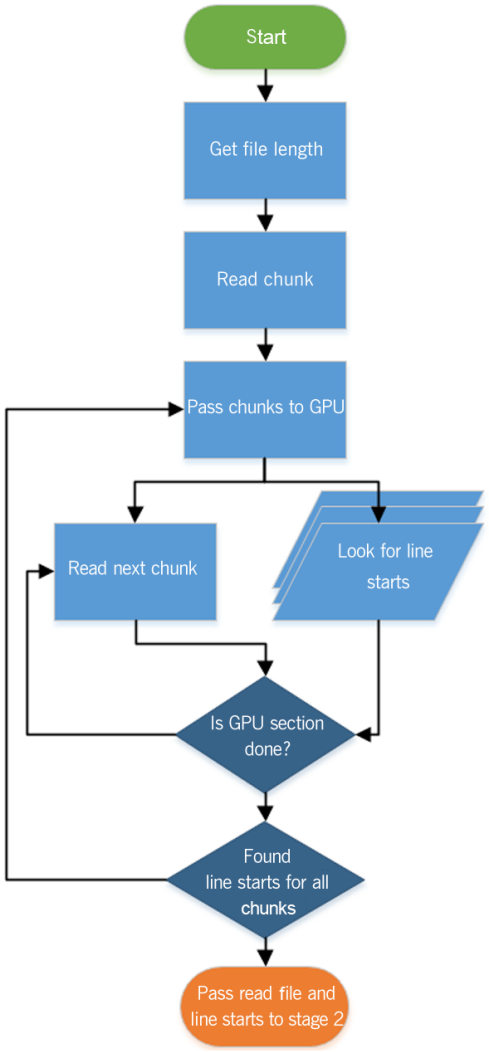

Fig. 2 Import stage.

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 shown in Fig. 3, 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

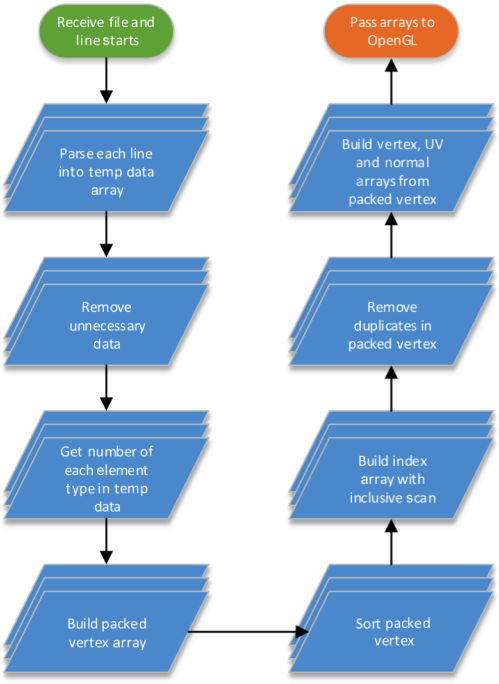

Fig. 3 Reading and parsing stage.

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 have 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 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. In a perfect situation n triangles would only require $n + 2$ 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. To save file space OBJ files index each individual data element removing duplicates; OpenGL, however indexes each unique combination of these data elements. The combination of these elements is the same as those in a PackedVertex minus the extra value: the draw order index.

Algorithm 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 don't need to preserve order.

Step 2, we create an array of unsigned integers the same size as the sorted packed vertices 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 $O(n)$. Though, when the threads write the new ordering, they are not accessing contiguous memory inside the warp, and as such, the process slightly suffers from an almost

Algorithm 2: Parallel VBO indexing
Input: OBJ file.
Output: V vector 3D coordinates,
UV vector 2D coordinates,
N vector normal vectors,
I vector indices.

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

/*Kernel Code*/ //Step 3
for all p do
  if threadID == 0 or P[threadID] != P[threadID − 1]
  then
    tempI[threadID] = 1
  end if
end for
/*End Kernel Code*/

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

/*Kernel Code*/ //Step 5
for all p do
  I[P[threadID].Index] = temp[threadID]
end for
/*End Kernel Code*/

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

/*Kernel Code*/
for all p do
  V[threadID] = P[threadID].vertex
  UV[threadID] = P[threadID].uv
  N[threadID] = P[threadID].normal
end for
/*End Kernel Code*/

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

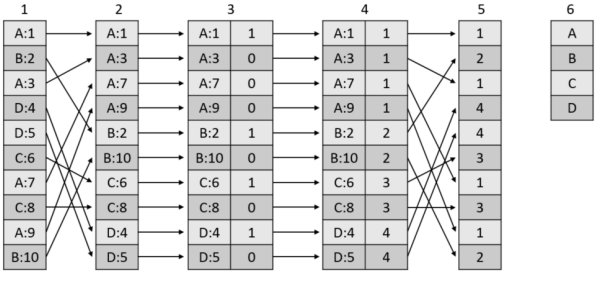

Fig. 4 Parallel VBO indexing.

negligible, time wise, lack of memory cohesion.

Step 6, we show 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 specification 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 are 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 (http://syoyo.github.io/tinyobjloader/) was also

Table 1 Hardware differences

Device	i7 2600K	GTX 970	GTX 750
Cores	4(8)	1664	512
Base (MHz)	3400	1050	1020
Boost (MHz)	3800	1178	1087
Memory (GB)	8	3.5(4)	2
TDP (W)	95	145	55

used as a direct code to code comparison. 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. and is shown in Fig. 5 (http://graphics.stanford.edu/data/3Dscanrep/).

4.1 Import test

The mesh was first taken into Pixologics Zbrush and decimated at intervals of 10%. Table 2 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.

Fig. 5 XYZ RBG Asian dragon.

Table 2 Mesh breakdown

Decimated to	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

4.2 Under load test

In the second experiment, we ran the tests, but only for the un-decimated mesh (100% in Table 2) 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 comparision test

The final experiment compared the GPU method on two different GPUs, looking at the time that the reading section and parsing/indexing section take and comparing these values while running on two 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 5×−8× faster under CUDA on the GPU, than it does sequentially in C++ on the CPU. Figure 6 shows the overall runtime 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. Its import times were almost

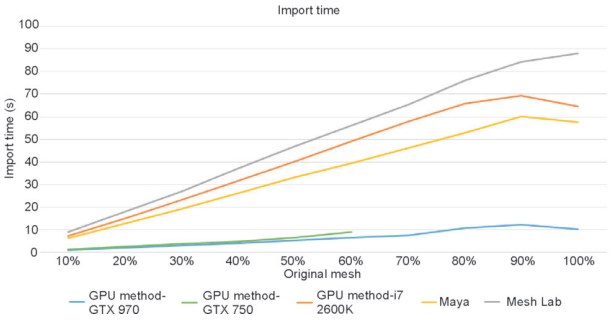

Fig. 6 Import time comparison.

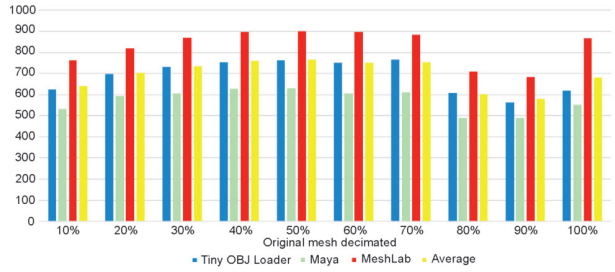

Fig. 7 GTX 970 speedup per mesh comparison.

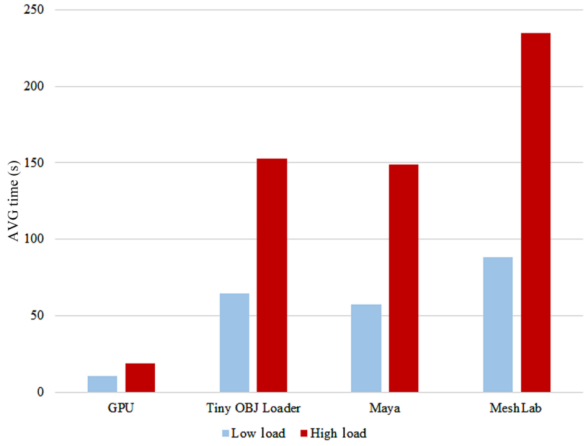

Fig. 8 Load comparison.

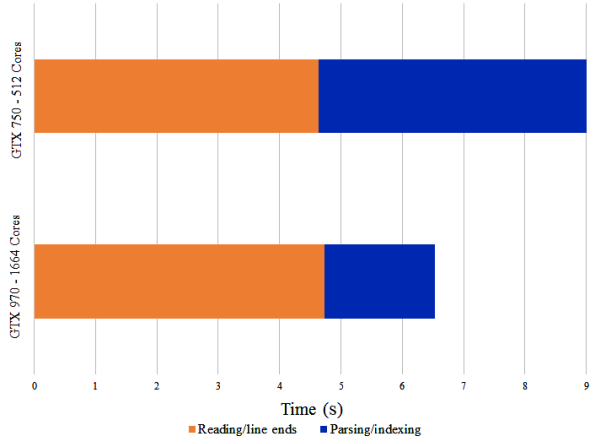

Fig. 9 GPU comparison.

exactly in between Maya and MeshLab, which ran the slowest, and as such, taking into account the Maya optimizations, serveing as the code to code comparison.

The GTX 970 with its 4 GB of memory, easily fits both the large mesh file and the parsed arrays in memory in all cases. However, the GTX 750 with 2 GB 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 5×.

Figure 7 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.

When running the tests again on a loaded system (Fig. 8) the speedup between the loaded GPU and the loaded CPU method was the same as the unloaded speedup. The GPU method, with an approximate 3× speedup, still ran faster on a loaded CPU system than the CPU method running on an unloaded system.

Figure 9 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 3× the number of CUDA cores; however the major performance differs only in the highly parallel parse and index section.

6 Discussion/conclusions

With this research we have successfully proven that GPUs can speed up parsing of OBJ mesh files. However, as cores increase, further speedup is only in the parsing and indexing section of the code, showing the effect of Amdahl's law.

As talked about earlier, Amdahl's law states that the potential parallel speedup, for any fixed amount of processing, is limited by the amount of code run in serial. The reading section of our solution is mostly serial as it requires the CPU to fetch the file data for the GPU before any parallel processing can occur. While we have borrowed some parallel time from later processing, by calculating line ends at the same time as the CPU fetches blocks, we are still bound by the time that the CPU takes to fetch the whole file. Theoretically, the temporary objects themselves could be filled out at the read stage instead of just the line starts, once again borrowing more time. However, as the later face data relies on the data before it, that is as much parallelization as could be done without having all data accessible to the GPU

simultaneously.

Due to simplicity of Amdahl's law, it makes it difficult to calculate direct values for how much improvement could be made. In its original form, it doesn't take into account differences in: clock speed between the GPU and CPU; GPU shared memory; or data transfer rates between the two systems. There have been several attempts to refine the law for the multicore era [12, 13] but none applying it to GPU – CPU applications. That being said, its effects are still felt, for if we were to, hypothetically, increase the number of cores further, we would still see performance increases: these would be diminishing, as the serial sections of code floods the results.

The modification that would lead to the greatest speedup of the parsing and indexing section of code would most likely be some form of mesh preprocessing, either at the read step, or when the file is first exported. Removing comments or render information costs roughly 2 seconds on the 100% mesh, as at that point n is the total number of lines in the file whereas later, when indexing, n is only the total number of triangle a factor of approximately 4.5× less in our file examples. By parsing the proxy objects at read time, comments could be ignored and culled per chunk, therefore eliminating this overhead.

7 Future work

In this paper we have shown that GPUs are capable of parsing OBJ files upwards 5× 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, MeshLab, as well as Tiny OBJ Loader) we provide strong evidence to this case. Though we see significant speedups, 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 [6] 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. Using an RAM disk to store the data for import could be another option, as the GPU could have direct access to the files though NVIDIA's shared memory, or via pinned memory addressing. Future technologies like NVlink could also see more hardware support for GPU access, with potential speed increases there.

Other than just speeding up the current use case, this method could be applied to other different data types. Adding support for hierarchical data at parse time would allow us to process other 3D file types like FBX, Maya ASCI, STL, and PLY. This could be done by modifying this method into an open source development tool like Blender, which already features some acceleration from CUDA.

Beyond just 3D data, comma separated values would be easily read by this system as it is currently implemented. With hierarchical data handling JDON and XML files could also be parsed in much the same way as the other data types.

References

[1] Nyland, L.; Harris, M.; Prins, J. Chapter 31. Fast N-body simulation with CUDA. In: *GPU Gems 3,* 677–696, 2007.

[2] Rinaldi, P. R.; Dari, E. A.; Vénere, M. J.; Clausse, A. A lattice-Boltzmann solver for 3D fluid simulation on GPU. *Simulation Modelling Practice and Theory* Vol. 25, 163–171, 2012.

[3] Stone, J. E.; Hardy, D. J.; Ufimtsev, I. S.; Schulten, K. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling* Vol. 29, No. 2, 116–125, 2010.

[4] 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, Vol. 1, 208–217, 2014.

[5] Johnson, M. Parsing in parallel on multiple cores and GPUs. In: Proceedings of the Australasian Language Technology Association Workshop, 29–37, 2011.

[6] Silberstein, M.; Ford, B.; Keidar, I.; Witchel, E. GPUfs: Integrating a file system with GPUs. *ACM Transactions on Computer Systems* Vol. 32, No. 1, Article No. 1, 2014.

[7] 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, 94–103, 2010.
[8] Si, X.; Yin, A.; Huang, X.; Yuan, X.; Liu, X.; Wang, G. Parallel optimization of queries in XML dataset using GPU. In: Proceedings of 2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming, 190–194, 2011.
[9] 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* Vol. 38, No. 8, 391–407, 2012.
[10] Keutzer, K.; Massingill, B. L.; Mattson, T. G.; Sanders, B. A. A design pattern language for engineering (parallel) software: Merging the PLPP and OPL projects. In: Proceedings of the 2010 Workshop on Parallel Programming Patterns, Article No. 9, 2010.
[11] Ghorpade, J.; Parande, J.; Kulkarni, M.; Bawaskar, A. GPGPU processing in CUDA architecture. *Advanced Computing: An International Journal* Vol. 3, No. 1, 105–120, 2012.
[12] Heath, M. T. A tale of two laws. *International Journal of High Performance Computing Applications* Vol. 29, No. 3, 320–330, 2015.
[13] Hill, M. D.; Marty, M. R. Amdahl's law in the multicore era. *Computer* Vol. 41, No. 7, 33–38, 2008.
[14] Murray, J. D.; vanRyper, W. *Encyclopedia of Graphics File Formats*, 2nd edn. O'Reilly Media, 1996.

Aidan L. Possemiers is currently pursuing his honours in information technology at James Cook University, in Cairns, Australia. He is involved with research and development in the fields of gamification and ecotourism. His other research interests include GPGPU, data visualisation, embeded systems, photogrammetry, modelling, and animation.

Ickjai Lee obtained his Ph.D. degree in 2002 from the School of Electrical Engineering and Computer Science, University of Newcastle, in Australia. After a year as a postdoctoral research fellow at the Business and Technology Laboratory in the University of Newcastle, Australia, he joined the School of IT at James Cook University, Australia. He has been actively involved in working on broad areas of geoinformatics and intelligence informatics. His reserach interests include geospatial data mining, Internet of things, Voronoi/Delaunay tessellations, data structure and modelling, smart cities/homes, GIS, and health informatics.