jamk.fi

Optimizing Radeon VRAM behavior

Lauri Kasanen

Bachelor's Thesis
April 2014

Degree Programme in Software Engineering
School of Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author	Type of publication	Date
Kasanen, Lauri	Bachelor´s Thesis	21.04.2014
	Pages 61	Language English
		Permission for web publication (X)

Title
Optimizing Radeon VRAM behavior

Degree Programme
Software Engineering

Tutor
Rantala, Ari

Assigned by
Mesa community

Abstract

The possibility of applying neural networks for optimizing VRAM behavior was researched in the thesis, and a set of software including a memory simulator was created for the task. Data was gathered from a wide variety of games and applications, producing memory traces suitable for the simulator. Using the simulator to play back memory traces, the efficiency of various approaches could be measured. The simulator was first used in minimizing fragmentation, and later on in training the AI.

The current state of the art of neural networks is shortly reviewed, as well as the common applications for each method. The methods chosen for this study are covered in more detail. The main training methods used were genetic/evolutionary training for the initial solution, and then fine-tuning with a Monte-Carlo solution.

As a result, fragmentation was changed in such a way that eviction was reduced by up to 20%. The change was accepted into future Linux kernels, starting with version 3.15.

The AI achieved acceptable results in optimizing the placement of buffers, improving the performance of most tested applications by 1-2% under memory pressure. The smoothness of each application improved as well, resulting in a more pleasant user experience.

Keywords
Radeon, graphics adapters, artificial intelligence, neural network

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä Kasanen, Lauri	Julkaisun laji Opinnäytetyö	Päivämäärä 21.4.2014
	Sivumäärä 61	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty (X)

Työn nimi
Optimizing Radeon VRAM behavior

Koulutusohjelma
Ohjelmistotekniikka

Työn ohjaaja
Rantala, Ari

Toimeksiantaja
Mesa-yhteisö

Tiivistelmä

Opinnäytetyössä tutkittiin mahdollisuutta parantaa näytönohjainajurin tehokkuutta soveltamalla tekoälyä muistinkäytön hallintaan. Tekoälyn kouluttamista varten luotiin muistisimulaattori sekä muuta ohjelmistoa. Dataa kerättiin laajasta valikoimasta pelejä ja sovelluksia, tuottaen simulaattorille sopivia muistijälkiä. Ajamalla kerätyt muistijäljet simulaattorin läpi voitiin mitata erilaisten lähestymistapojen tehokkuutta. Simulaattoria sovellettiin ensin sirpaloitumisen minimointiin, ja myöhemmin tekoälyn koulutukseen.

Työssä esitetään lyhyesti tekoälytutkimuksen nykytila ja sovellukset. Tutkimukseen valitut tekniikat käydään tarkemmin läpi. Tärkeimmät käytetyt koulutusmetodit ovat geneettinen koulutus, jolla saavutettiin ensimmäinen ratkaisu, sekä Monte-Carlo-metodit, joilla ratkaisua hienosäädettiin.

Tuloksena saatiin muutettua sirpaloitumista siten, että puskureiden edestakainen liikenne väheni jopa 20%. Muutos hyväksyttiin tuleviin Linux-ytimiin, alkaen versiosta 3.15.

Tekoäly saavutti kelvollisen tason puskurien sijoittamisen optimoinnissa, onnistuen parantamaan useimpien testattujen sovellusten suorituskykyä n. 1-2% muistipaineen alla. Myös sovellusten suorituskyvyn tasaisuus parani, tuottaen miellyttävämmän käyttökokemuksen.

Avainsanat (asiasanat)
Radeon, näytönohjaimet, grafiikka, tekoäly, AI

Muut tiedot

Contents

List of Figures

Glossary

AI	Artificial Intelligence.
buffer	A block of memory used for some purpose. For example, a texture.
Catalyst	AMD's proprietary graphics driver.
FPS	Frames per second, a common performance metric.
Frame time	The inverse of FPS, i.e. (milli)seconds per frame.
Linux	An open-source operating system.
LRU	Least Recently Used, a type of cache algorithm.
mipmapping	A technique of accessing pre-scaled versions of a texture.
r600g	The open-source Gallium3D graphics driver responsible for Radeon HD 2000 - HD 6000 models.
RAM	Random Access Memory, a type of computer memory.
squirrel	A small, furry creature.
texture	A buffer consisting of image data.
VRAM	Video RAM, fast on-board memory included on graphics cards.

Acknowledgements

I would like to thank Jerome Glisse for his guidance, Thomas Hellstrom for reviews, Michael Larabel for coverage, and everyone who contributed data to this thesis.

1 Introduction

Graphics cards carry some fast, dedicated memory (VRAM). This memory is used to speed up tasks such as rendering or parallel computations by avoiding the latency in having the graphics card access the system memory.

VRAM is a limited resource, and as such, all workloads may not fit in a given graphics card. In a situation like that, the strategy that decides what parts of the working set to keep in VRAM, and which parts can be kept in regular system RAM, is of utmost importance.

If the application needs to access a specific resource in system RAM, it will have a corresponding hit in its performance. The VRAM placement strategy should try to rank the application's buffers in such a way as to maximize the application's performance.

Application performance is commonly measured in two ways: throughput (FPS, frames per second) and smoothness (how stable the FPS is). For a good user experience, they both must be above some user-dependent threshold.

There are no generally accepted guidelines, as the minimum for an acceptable experience is quite subjective. However, many people draw the line at 60 FPS with a minimum FPS of 55. (Minimum playable FPS 2012)

Both throughput and smoothness must be above the user's minimum requirements, one alone will not suffice. For example, a game running at a healthy throughput of 60 FPS, but dipping to 10 FPS once per second will be extremely irritating, perceived as stalling or stuttering. Vice versa, a smooth FPS of 20 will not stutter, however, the throughput is not high enough for an enjoyable experience, as the user will notice individual images instead of the illusion of movement.

Notice the huge frame time spikes on Catalyst in Figure 1: better performance, but worse smoothness.

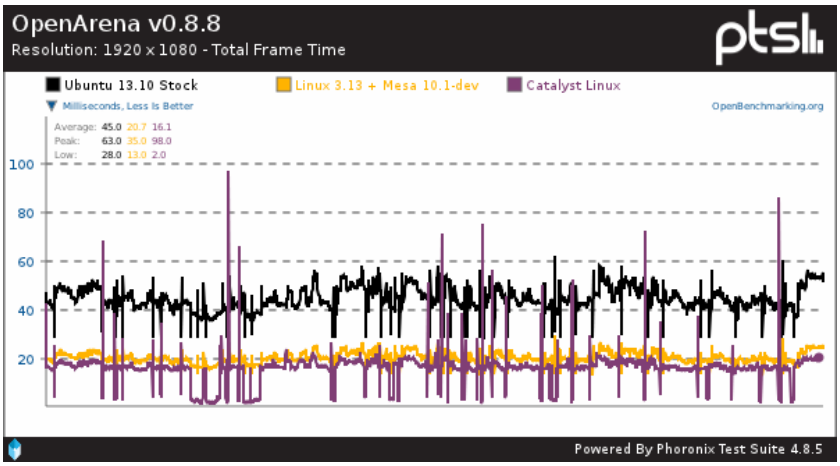

Figure 1. Frame time measurement. Image © Phoronix, printed with permission.

2 Problem

2.1 Problem background

The open source graphics drivers on Linux are generally thought of as lower performance, but higher stability, than the proprietary drivers provided by the graphics card companies. This is a result of the bar for quality in public work; each new version of a proprietary driver generally includes many application-specific changes (Catalyst 13.1 release notes 2013), which would not fly in an open-source driver.

Recent advances in optimization have brought the r600g driver close to competing with the proprietary Catalyst driver (Larabel 2013a). Depending on the benchmark and hardware, it is within 60% - 110% of the Catalyst driver, generally around 80% (Figure 2).

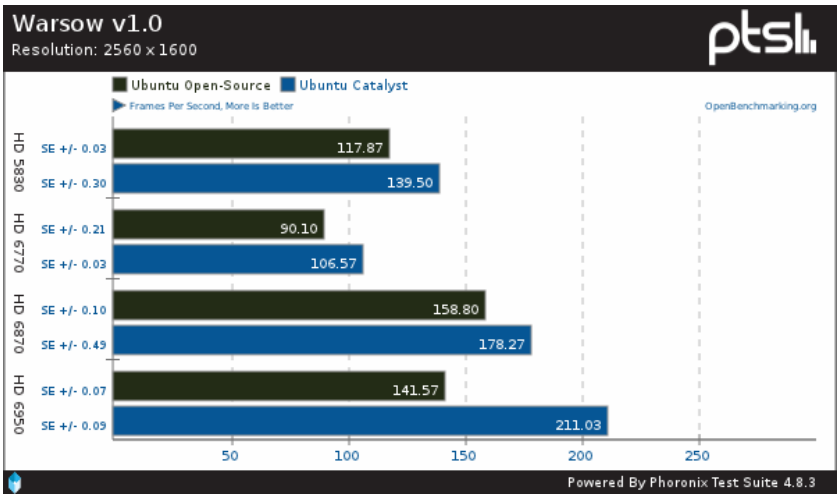

Figure 2. Warsow benchmark at 2560x1600. Image © Phoronix, printed with permission.

One of the remaining areas yet to be optimized is the VRAM placing strategy. In this thesis, research will be done on improving this area of the r600g driver.

There are some practical constraints that the solution will need to abide by. First, there is no central location with a holistic view of everything that happens on the graphics card. Secondly, the run-time performance of the strategy itself must be adequate. It may have to make thousands of decisions per second; having performance go down because the computer is thinking on how to improve performance would be counterproductive.

A perfect solution would know everything that goes on in the system. It could predict exactly what the application will do next, and in what way does it access each buffer. Sadly, either of those is a luxury not available in the real world.

While recording the way the application accesses each buffer is possible in theory, this would incur more overhead than could be gained. Thus, the strategy is limited to considering worst-case possibilities, and it has to make decisions as if the application would use each buffer fully. In practice, many buffers are not fully read or written, but only parts of them are used, for example due to mipmapping.

The current strategy used in r600g is a simple LRU (least recently used) list. While simple, it can lead to quite a bad user experience in non-trivial cases that exceed the VRAM size. A buffer that is needed next could have been swapped out to system RAM, dropping performance. In the worst case, this creates a ping-pong effect where buffers are constantly moved back and forth, dropping performance to single digits FPS. (Larabel 2012; Very low FPS when video memory is full 2013)

Given that the absolute knowledge is not available, the system will have to make do with what is there. For example, the system can keep accurate statistics on the times the application declares it will read from a buffer. Though the system cannot know how much the application will read, it does know the application will read from the buffer. Using this information, it can make a worst-case estimation that the application will read the buffer in full.

Such a worst-case estimation will directly improve the minimum FPS; in other words, the smoothness. This has an indirect benefit to performance, however optimal performance will not be reached by these means.

The target of this study is to improve on the current strategy, particularly in the cases that are currently pathological. The improved system may not gain significantly in throughput, but it should gain measurably in smoothness.

2.2 Problem definition

Knowing what information the system has available, and with the above goals in mind, the problem could be defined more clearly.

The component that decides how to place buffers is separate from the component that has access to accurate statistics. Moving information between these components, the kernel and the userspace, is expensive; therefore the information moved must be minimized.

According to the privilege separation, userspace should not be able to directly decide a placement in memory space, as this could cause security holes. Moving the entire statistics data over would be too expensive. How about ranking the buffers, giving each buffer an importance score?

This limits the information needed to pass to the kernel to one integer per buffer. In addition, the kernel is free to disregard this hint, keeping in line with the separation. Knowing the relative importance of each buffer, the kernel should be able to make much better buffer placing decisions than the current LRU strategy.

While giving each buffer a score could certainly be done by the usual programming techniques, linear/weight calculations and a set of if-conditions, it is believed that the relationship of the statistics to the buffer's importance is both non-linear and hard to model manually.

Given this assumption, it is likely that such a manual method would not do well in many cases, and it would be constantly tweaked to accommodate newly discovered pathological cases. There is no existing model for a buffer's importance, and no single right answer to the question "How important is this buffer?"

The mainstream solution to solving non-linear, unknown models like that (also known as modeling or regression problems) is to use AI solutions such as neural networks.

3 Artificial Intelligence

3.1 Theory

There are many approaches to artificial intelligence. What is common to all of them though, is the ability to make decisions the computer was not told explicitly how to make. They differ in their areas of usability, theoretic foundations, whether they are based on real biological phenomena, and other ways.

One such approach is the neural network. Invented in the 60s, it mimics the biological brain cells. Such networks are able to generalize, to learn either independently or with guidance, and tend to achieve quite decent results. Neural networks have been applied to problems such as Backgammon (Tesauro 1994), business data mining (Bigus 1996), and text compression (Mahoney 1996) with success.

Neural networks come in many varieties. The multi-layer perceptron (MLP) is the most common one, used for classification, modeling, and time-series prediction. The radial basis function network (RBF) shares the same uses. The adaptive resonance theory network and Kohonen map are used for clustering. Recurrent networks are used for extremely complex modeling problems. (Bigus 1996, p. 77)

As the problem here is a modeling one, this narrows the choice to either MLP, RBF or recurrent.

Recurrent networks are generally hard to train, and their runtime performance is not deterministic: they may take ten or hundred times longer to make a decision compared to another. This is due to them spinning until the result has converged to a stable value, instead of only running through once like the other models. This rules that model out.

The choice between MLP and RBF is somewhat arbitrary. Liu and Gader (2000) found that RBF ignores outliers better, while MLP is said to perform better. MLP is also covered more in literature. The architecture of a multi-layer perceptron is shown in Figure 3.

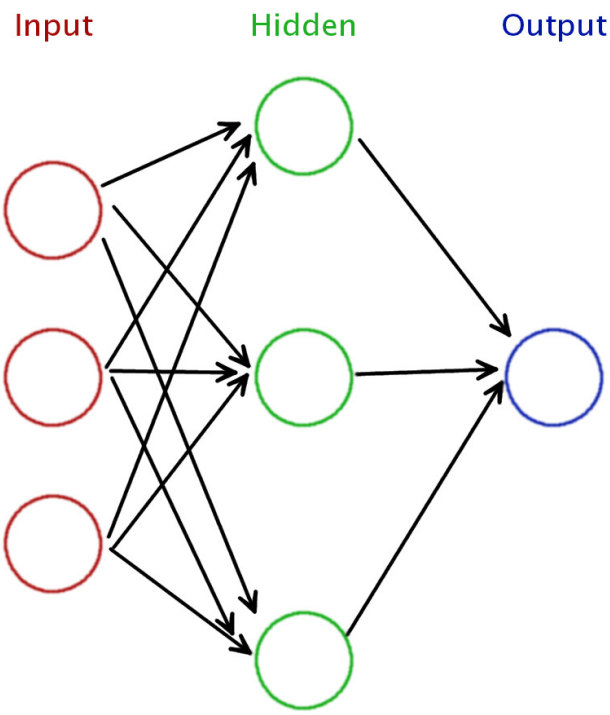

Figure 3. Multi-layer perceptron.

There are many ways to train a MLP neural network. The algorithms are usually divided into four types: supervised, unsupervised, competitive, and reinforcement learning (Siddique & Adeli 2013).

Supervised learning is used when the test data is clearly labeled. For example, if the task was to detect if a picture contains a squirrel, sets of pictures of both squirrels and non-squirrels would be fed in, each labeled by a human on whether it contains a squirrel. Then the network's guesses are compared to the labels and corrected until the network can correctly determine whether a picture contains a small, furry creature or not.

In unsupervised learning, the network is not told anything about the data. It is used mainly in clustering problems, where the clusters are not defined beforehand. For example, the network may be fed customer data, and asked to segment customers into four classes. Studying the decisions it made can be very useful in finding new or undervalued customer segments.

Competitive learning is used in classification problems mainly. Only the neuron that "wins" the round, in other words whose guess was closest to correct, gets to be tweaked. This results in each neuron specializing in a specific type of input.

Reinforcement learning is used in stateful problems, where each action may not be graded alone, and only the full path of actions may be graded. For example, the Backgammon network of Tesauro (1994) used this method.

Alternative methods for tweaking the network, instead of training it, include genetic and evolutionary methods as well as Monte-Carlo methods.

3.2 Choice and rationale

For the network type, the common multi-layer perceptron architecture was chosen. It is supported by a wide literature base, and its deterministic performance characteristics are necessary for this kind of use.

The number of hidden layers depends on the type of modeling being done. For most problems, a single hidden layer is enough. Each hidden layer forms a hyperplane in pattern space, such that two hidden layers are able to define a region, and three hidden layers are able to define a closed area (Siddique & Adeli 2013, p. 111).

The complexity of this hyperplane depends on the used activation function. The simple step functions create a flat plane, and networks using those would require two or three hidden layers to capture non-linear relationships. S-curve functions are able to create a much more complex plane, such that a single hidden layer with a more advanced activation function is able to approximate non-linear relationships. A single layer was chosen, as S-curve functions will be used.

With a single hidden layer, the big question is how to choose the amount of hidden nodes. There is no generally accepted answer to this question, and all the literature offers is guesses and rules of thumb. One such widely quoted rule is to use a number between the amount of input nodes and the amount of output nodes.

The number of hidden nodes can be easily pruned later if the network decides some are unnecessary, however, they cannot be added without re-training. Given this restriction, the number of hidden nodes was set at the number of input nodes; that is, nine.

For training, in the VRAM strategy case the seemingly only option would be reinforcement learning. Supervised learning cannot be used, as a human cannot give any buffer an importance score; unsupervised and competitive learning do not apply to modeling problems.

However, reinforcement learning is not a good fit for modeling problems (Wiering, Hasselt, Pietersma & Schomaker 2011). Wiering et al found that while reinforcement learning can be applied to such problems, and the result performs on a par with a network trained by supervised learning, the training was slow, and there is no guarantee that the network will not get stuck in local minima.

Given that none of the generally used training methods are applicable, the alternative methods of genetic and Monte-Carlo evolution were chosen. The two methods are covered in more detail in Section 6, *Training the network*.

4 Data gathering

All further steps required data. In the first weeks, the needed data points were planned out, and the data gathering was implemented as patches to Mesa. Since it was clear from the start that wide coverage would be needed, the public was asked for assistance (Kasanen 2014; Dawe 2014).

In addition to data gathered by the researcher, the public sent a wide variety of traces, enhancing the coverage much further than would have been possible otherwise.

To get suitable data for the purposes of this research, the following data points were selected as inputs:

- number of reads
- number of writes
- time since last read
- time since last write
- buffer size
- number of processor operations
- time since last processor operation
- whether the buffer should be considered high priority (MSAA, depth)
- VRAM size.

Timing information was set at millisecond accuracy. A time measurement was chosen instead of the frame number, because it ought to allow for a better user experience. Should frames take long, inter-frame swapping should be minimized. Should frames be fast (< 10 ms / frame), timing is good as well, since the user experience works on longer timescales.

A game might only draw shadows every other frame, or less often; this causes those frames to take longer than the frames without such extra work. Yet, the user will notice if every Nth frame is too slow. It remains to be seen whether this level of accuracy is good; other choices beside the frame number include thresholds determined by common user studies (10 ms, 30 ms, 60 ms, 100 ms…) or non-linear scaling.

To enable those inputs to be replayed, the memory traces listed each operation on a buffer along with timing information. (See Figure 4.)

```
cpu op buffer 7 at 3 ms
create buffer 8 at 3 ms (8 bytes)
cpu op buffer 8 at 3 ms
create buffer 9 at 3 ms (48 bytes)
cpu op buffer 9 at 3 ms
create buffer 10 at 3 ms (48 bytes)
cpu op buffer 10 at 3 ms
create buffer 11 at 3 ms (24 bytes)
cpu op buffer 11 at 3 ms
create buffer 12 at 3 ms (32 bytes)
cpu op buffer 12 at 3 ms
create buffer 13 at 3 ms (8 bytes)
cpu op buffer 13 at 3 ms
create buffer 14 at 3 ms (16384 bytes)
create buffer 15 at 3 ms (65536 bytes)
create buffer 16 at 3 ms (16384 bytes)
write buffer 16 at 3 ms
create buffer 17 at 3 ms (4096 bytes, high priority)
destroy buffer 17 at 3 ms
destroy buffer 17 at 3 ms
destroy buffer 14 at 3 ms
cpu op buffer 7 at 11 ms
```

Figure 4. Sample from a memory trace.

As the traces took a considerable amount of space in their uncompressed text form, a custom binary format was developed. A binary format also allows the traces to be read back much faster, an important point for speedy training.

Some helper applications were also developed to make it nicer to work with. Figure 4 above is from one such helper: a reader for the format with color highlighting. For more details on the format and on the applications, please see the Software section.

5 Fragmentation

5.1 Introduction

Fragmentation is a common problem in all memory management. As buffers get allocated, moved around, and deleted, the memory space becomes increasingly fragmented. It limits the maximum size of a new allocation, and so buffers bigger than this also cannot be moved to VRAM. [1]

An example of the effects is reported by Larabel (2013b). Big buffer allocations were failing due to fragmentation, causing the application to misrender and/or crash.

Figure 5. Fragmentation

Fragmentation (Figure 5) is an inevitable result of continued use. It can be mitigated by smart allocation strategies, and it can be repaired after the fact by moving the used buffers together (constraints allowing).

In normal system RAM, both strategies are viable. Cleaning up the memory area, also known as compaction (Corbet 2010) can be fairly low-impact to performance. It only involves freezing the process, and changing some page table entries, which is a relatively fast operation. The downside is that a TLB (translation look-aside buffer) cache flush is needed so that the cache does not give out the old, wrong addresses.

Under the graphics context, compaction is still doable, however, ideally it should not be done while running a heavy workload. The delay caused by a VRAM memory compaction operation may be measured in milliseconds, which may cause the frame to take too long, and would be seen as unacceptable stutter to the user.

As such, VRAM compaction should be limited to times when it would not be noticeable, for example application start-up and exit.

---

[1] The very latest generation, HD 7000, can use non-continuous memory areas with small overhead, so this issue is not as pressing there.

5.2 Simulation

In order to measure fragmentation in different situations, a simulator was developed. It replays the collected memory traces while simulating the VRAM placement using the existing LRU strategy. The results will not apply directly to other placing strategies, but they will be indicative of general trends regardless of the placing strategy.

The simulator took snapshots of the VRAM state once every ten memory operations, and counted the amount of holes (fragmentation). It also printed a marker every time an eviction was triggered.

Two different allocation strategies were tested. The default allocator allocates buffers from the start of VRAM. The proposed min-max allocator allocates buffers from two ends of the VRAM space, based on the assumptions that smaller buffers are recycled more often than large ones, and that recycling of each type would then only create fragmentation of the same type. The min-max allocator is visualized in Figure 6.

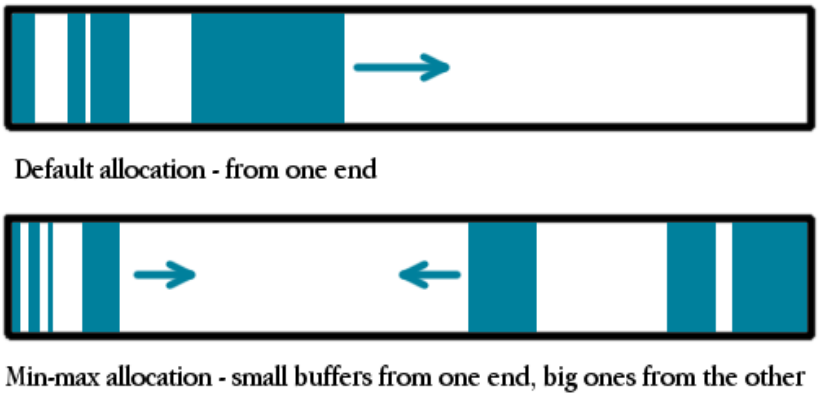

Figure 6. Allocation strategies.

Nine common VRAM sizes were tested: 64 MB, 128 MB, 256 MB, 384 MB, 512 MB, 1024 MB, 1536 MB, 2048 MB, and 4096 MB. In cases where the trace could not run on a configuration, that trace was skipped. For example, Planetary Annihilation allocated a buffer of 78 MB in size; it is obviously beyond the capabilities of a 64 MB VRAM graphics card.

Several threshold values were tested for the min-max allocator in order to find a rough optimum. The simulation took approximately 2.5 hours per run, and each run generated about 16 GB of data. As the amount of data was far too great to process on an ordinary office suite, a custom graphing tool was developed.

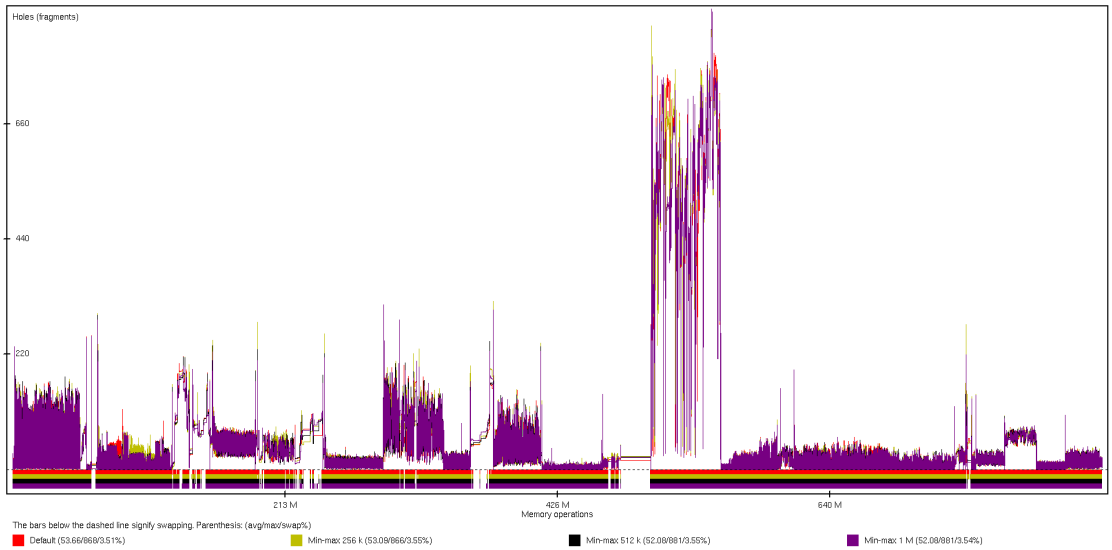

Figure 7. 64 MB VRAM

Starting off with the 64 MB VRAM run (Figure 7), it can be clearly seen that the workload is too heavy for this VRAM size. The heavy variance in the number of fragments, visible as dense vertical movement on the line graph, points out that there is considerable trashing going on.

There is not much difference in the swapping between the tested strategies. The horizontal swapping bars are very similar, and the exact percentage of swapping events varied between 3.51% and 3.55%.

Still, it is a good data point to have, in order to see how the strategies behave under heavy pressure.

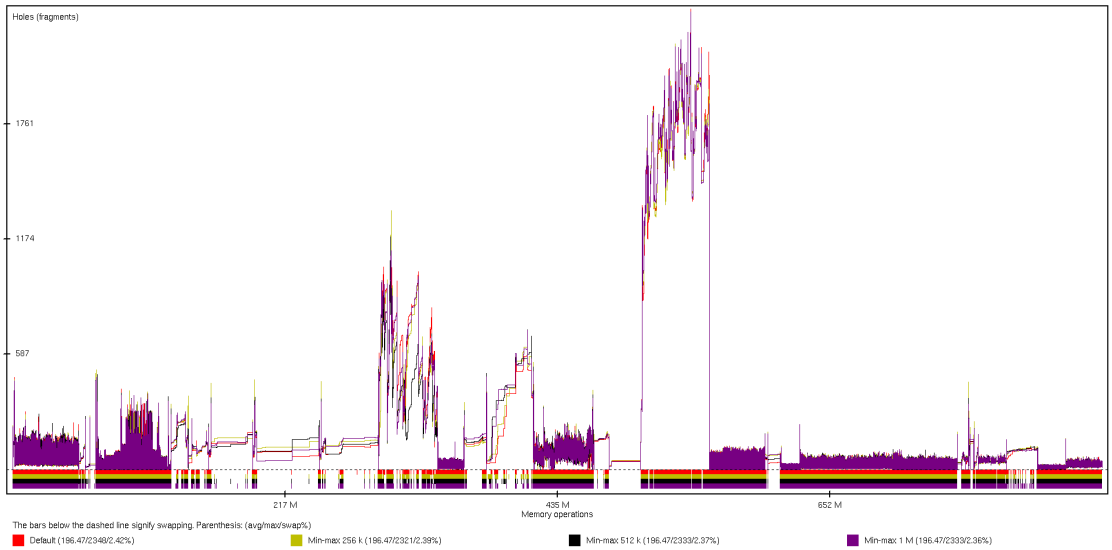

Figure 8. 128 MB VRAM

With 128 MB (Figure 8), all of the traces could be run through. The trashing is greatly reduced, and about half of the workload now shows smooth progression in the fragmentation lines.

Fragmentation is approximately equal in all cases, but the swapping (eviction) is lower in all of the min-max runs compared to the default LRU run. LRU had swapping in 2.42% of the time, whereas the lowest min-max run had 2.36%.

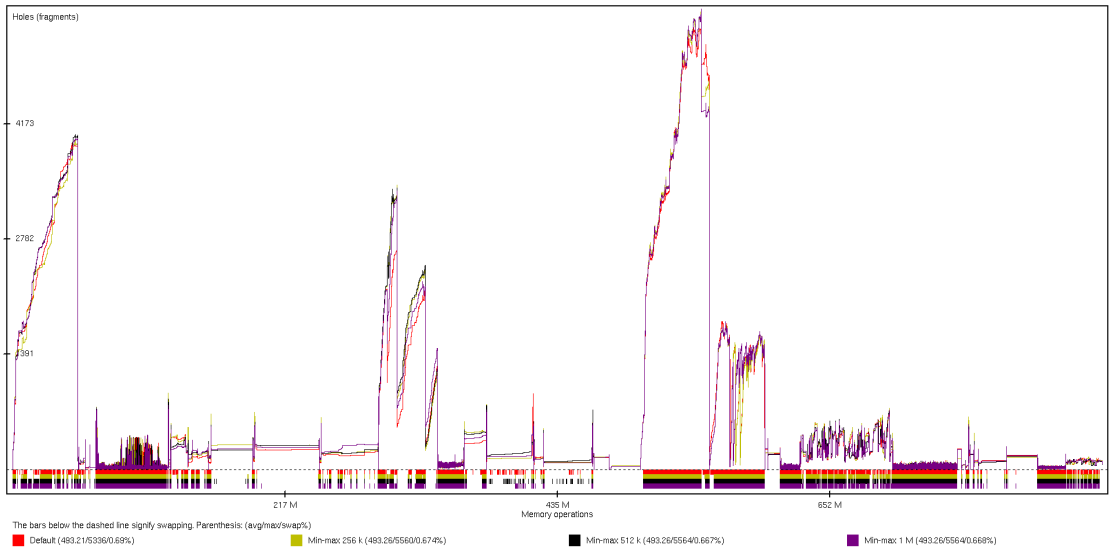

Figure 9. 256 MB VRAM

The patterns become visible in the 256 MB run (Figure 9). Almost all trashing is now gone. Peak fragmentation is surprisingly higher in min-max (5560 vs. 5336 holes), however, swapping continues to be lower: 0.69% in LRU, 0.667% in the lowest min-max run. The average fragmentation was the same between all runs.

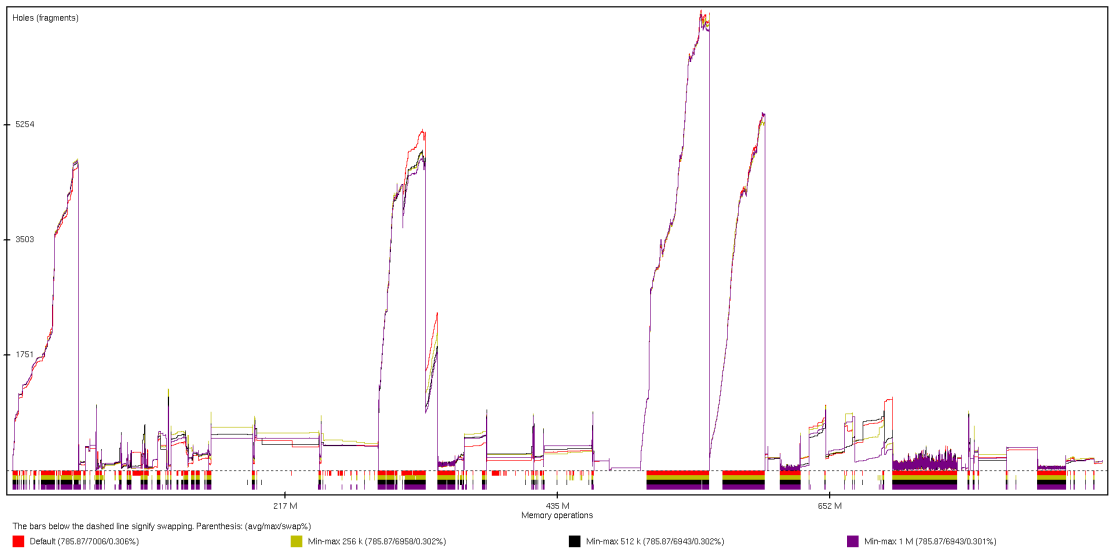

Figure 10. 384 MB VRAM

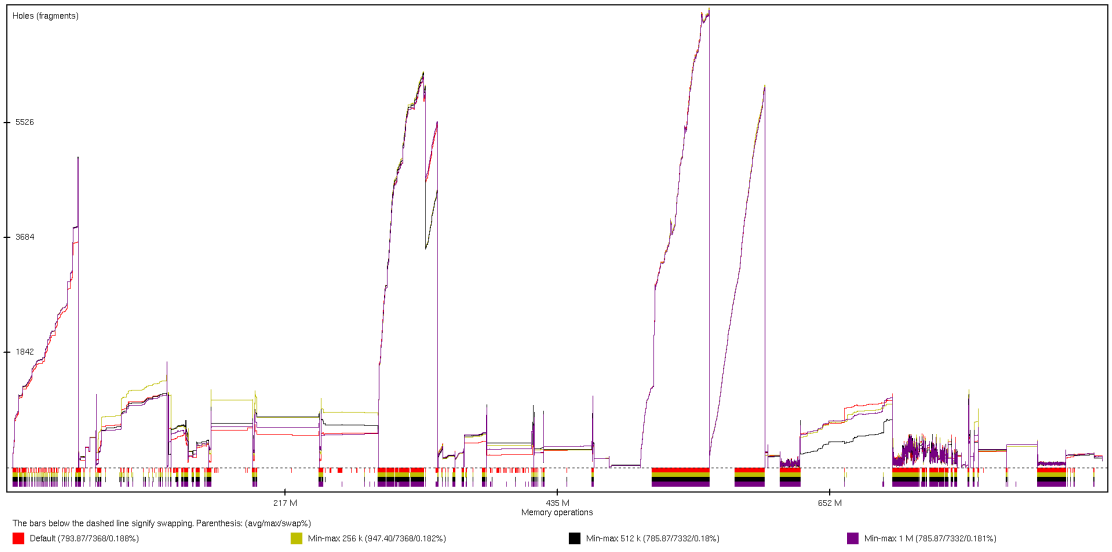

Figure 11. 512 MB VRAM

The 384 and 512 MB runs continue the same pattern: swapping is lower in the min-max allocator when compared to the default. (Figures 10 and 11)

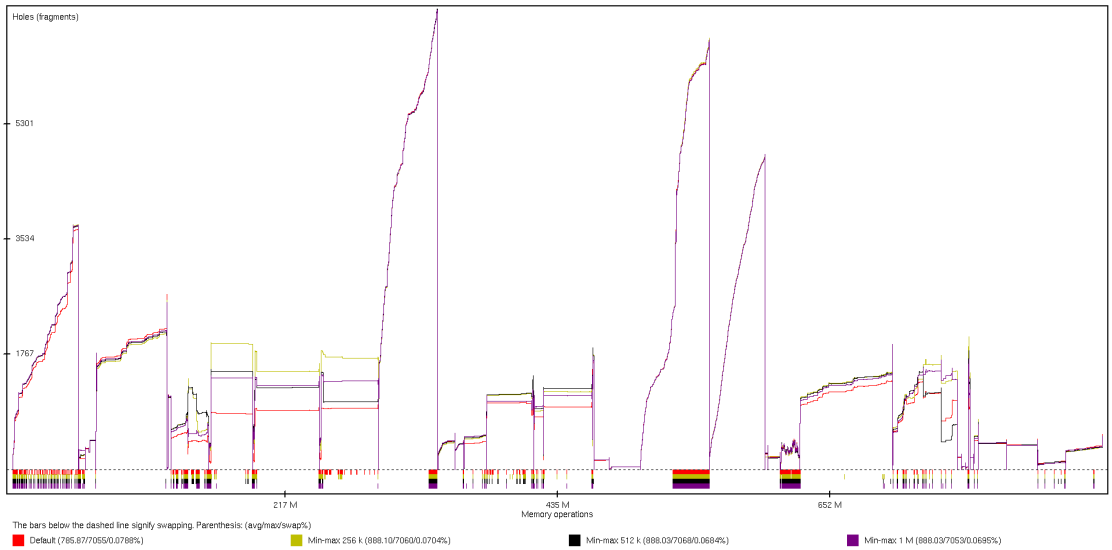

Figure 12. 1024 MB VRAM

Starting with the 1 GB run (Figure 12), the average fragmentation of min-max starts to climb above the default's: 785.85 in LRU, 888.10 in the highest min-max run. Swapping continues to be better in min-max, and now it is easily visible in the swapping bars as well. There are present bars in the LRU and 256 KB min-max areas (red and olive) that are absent in the higher-threshold min-max strategies.

The swapping difference continues to be significant. LRU had swapping in 0.0788% of the time, whereas the lowest min-max run had 0.0684%.

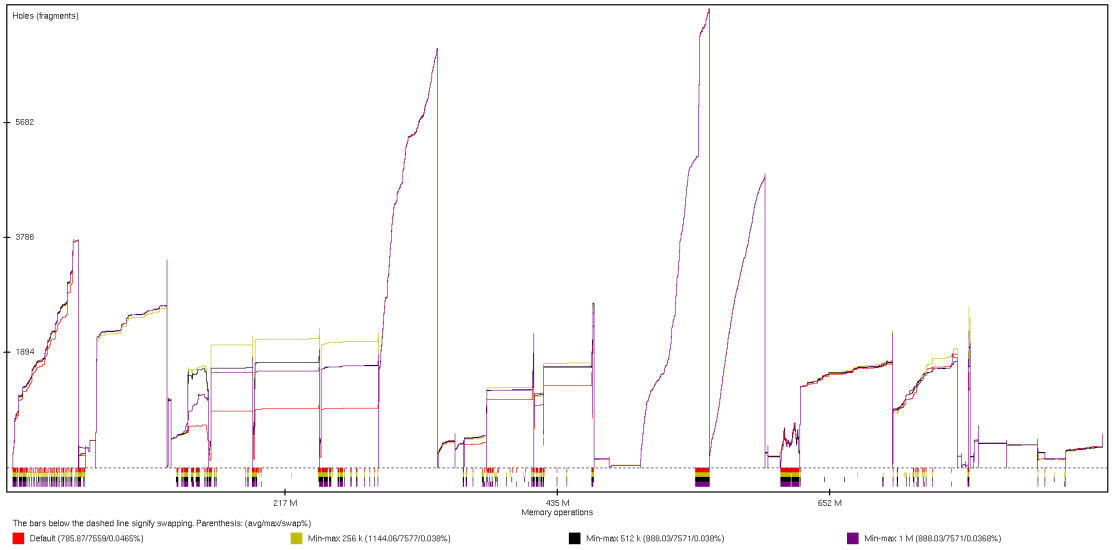

Figure 13. 1536 MB VRAM

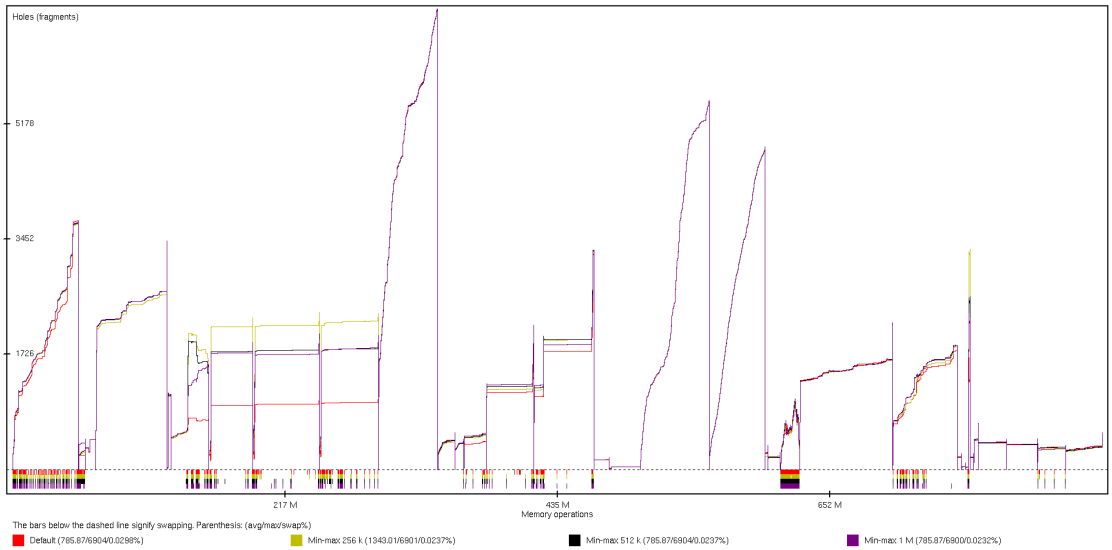

Figure 14. 2048 MB VRAM

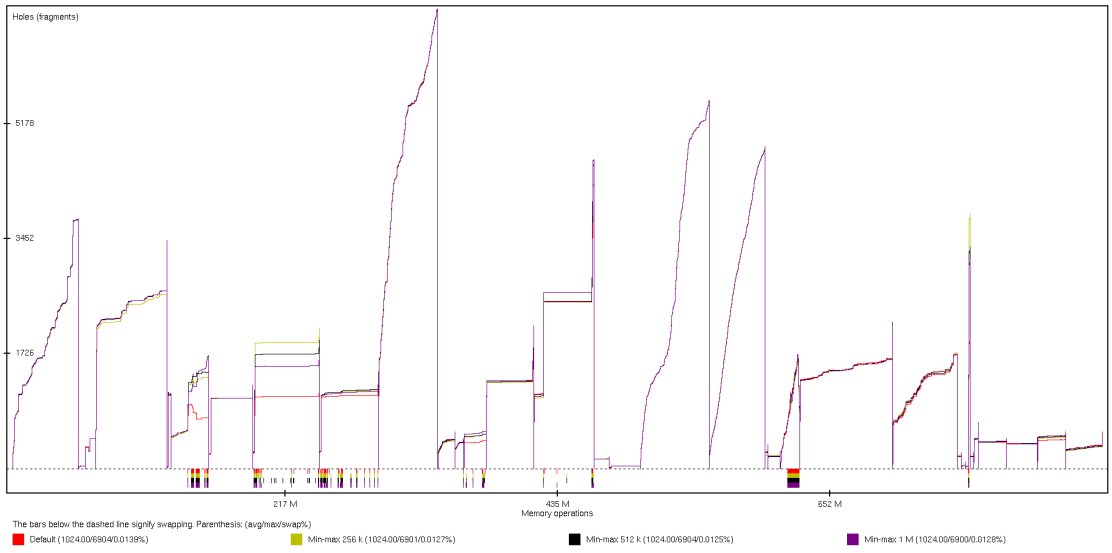

Figure 15. 4096 MB VRAM

The higher amounts of VRAM (Figures 13 to 15) continue in the same vein.

The above results follow that the min-max strategy turned out to actually increase fragmentation. Despite the higher absolute amount of holes, they actually decreased swapping in all runs except the 64 MB one. It is assumed that this is due to a better quality of fragmentation; that is, the holes created are more suitable for new allocations.

Gathering the swapping statistics together, the optimal threshold value can be determined (Figure 16).

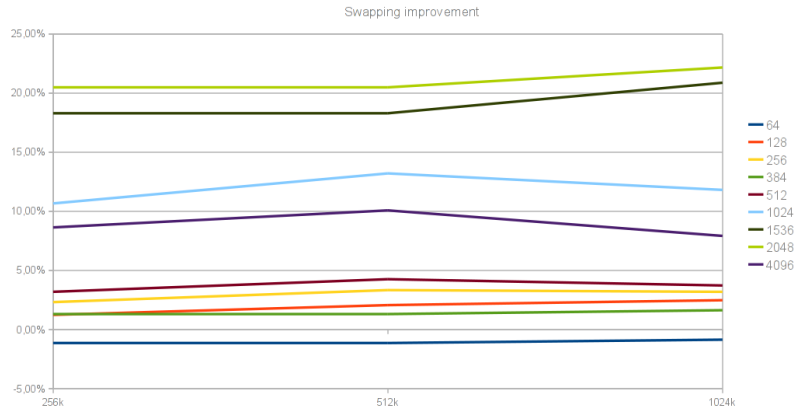

Figure 16. Swapping improvement over the default strategy.

As the workload was far too heavy for a 64 MB VRAM graphics card, causing high swapping rates, it is believed that that result can be ignored. The traces were generally recorded in 1366x768 resolution or higher, and such high resolutions are not supported by most graphics cards with 64 MB VRAM.

In all other runs, the min-max allocation strategy improved swapping over the default. For the 128 MB, 256 MB, 384 MB, and 512 MB runs the improvement was in single-digit percentages.

For 1024 MB and 4096 MB, the min-max allocation resulted in about 10% less swapping. For the last ones, 1536 and 2048 MB, the highest results were measured: around 20%.

As far as this test data goes, the optimal threshold for min-max allocation is 512 kb. While for some VRAM sizes it did worse than the 1 MB threshold, it also outdid the higher threshold in some cases. In no case did the 512 kb threshold lose to the 256 kb threshold, however.

6 Training the network

6.1 Activation function

The activation function is the transformation done inside each neuron, operating on the sum of all weighted inputs (plus bias). A variety of functions have been used over time: starting from simple step functions, continuing via exponential functions, to S-shaped functions. S-shaped functions are considered to be closest to how real neurons behave.

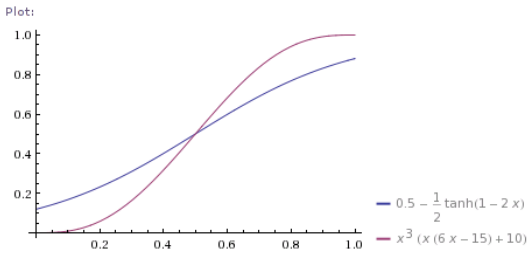

Figure 17. Hyperbolic tangent and smootherstep, scaled to use the same input/output space.

Testing a variety of these functions, Karlik and Olgac (2011) found that S-shaped functions had superior performance, reaching the highest accuracy of all tested functions.

So the question here is which type of function would give the best run-time performance; in other words, fastest to calculate. Three options were tested: the default S-shaped activation function, hyperbolic tangent; and an adaptation from the graphics world, Perlin's smootherstep function, both in floating point, and when converted to use fixed-point mathematics.

As can be seen in Figure 17 above, the hyperbolic tangent is less steep than the smootherstep function. This should not cause any issues in the decision-making.

Each function was ran 10^9 times. Surprisingly, the fixed point implementation was not the fastest of all. A single **tanhf()** call took approximately 4.8 ns (+- 0.2%). A single fixed-point **smootherstep()** call took ~4.1 ns. The fastest of all, floating-point **smootherstep()**, took only ~2.8 ns per call.

The slow performance of the fixed-point function is attributed to it consisting mainly of multiplication. Fixed-point multiplication requires both a multiplication and a division, making it an expensive operation even when the division is implemented as a shift.

As the floating-point smootherstep function beat the customary hyperbolic tangent by 31%, and there is no hard requirement against the use of floating-point mathematics, it was selected as the activation function.

6.2 Cost model

The basic outlines for the cost model can be had from the most common speeds of currently used memory types (GDDR5 for the VRAM, DDR3 for the system RAM).

As the major analyst houses keep this information behind paywalls (IDC for example would charge 5 000 $ for the latest two-page report), and the DRAM makers do not list this information in their financial reports, to get a rough view one had to resort to checking the inventory levels of a web shop.

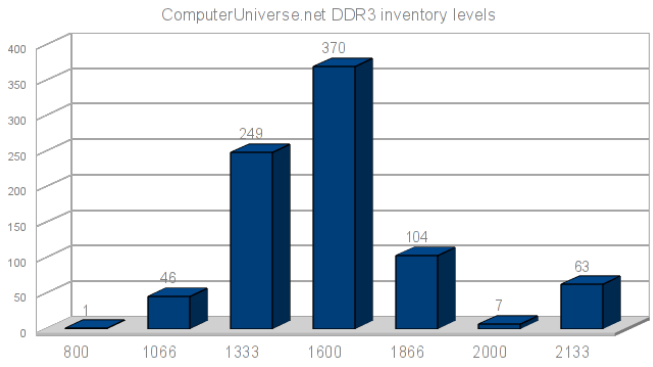

Figure 18. DDR3 inventory levels: the number of SKUs per each speed class.

From the inventory levels in Figure 18 it can be seen that 1600 MHz is the most popular type of DDR3 memory being sold. The bandwidth of such memory is 12.8 GB/s.

For the average GDDR5 speed, a mid-high-end card from both Nvidia and AMD's latest generations was chosen. Nvidia GTX 770 ships with a memory bandwidth of 224 GB/s, whereas AMD Radeon 280 has 240 GB/s. Taking the average ends up at 232 GB/s.

6.2.1 Other considerations

For a buffer in either memory, the cost of a read or write can thus be calculated as the buffer size divided by the memory bandwidth; however, a multitude of other considerations must be taken into account.

First of all, a GPU write to system RAM (cacheable memory) will incur a performance hit of about 66%. A buffer move, beyond the overlapping read in one memory type and the write in the other also costs some PCI-E latency. If the buffer is needed immediately after the move, the GPU engines will stall to wait for it, potentially delaying useful work. [2]

6.3 Monte-Carlo training

Monte-Carlo methods work based on randomness. They are used in cases where an exhaustive search is impossible due to the computation required. Due to their nature, they will often find a "good enough" solution, however, may also fail to find a solution altogether.

In the simplest form, a Monte-Carlo method means making a random change, and measuring whether the results improved. Restricting the change in magnitude and the amount of variables changed can be used to guide the method.

6.4 Evolutionary/genetic training

Evolutionary or genetic methods follow the behavior seen in nature. They can conduct a global search over the whole search space with resources far below an exhaustive search, and often succeed in finding the global optimum (Siddique & Adeli 2013).

The methods work as follows: the solution to the problem is converted to a genome form. A population of critters (animals, bacteria, etc.) is created by filling the genome of each randomly. This population is then allowed to have sex, intermingle, and mutate. After a few hundred or thousand generations, however long it takes for the population to converge close enough to a single solution, the "alpha male" is selected as the final solution.

---

[2] This is a limitation of the current driver. The latest generation of graphics cards supports multiple engines, and they could work on independent pieces should the driver support be there. In that case, only one engine would stall.

As the most fit individuals will have the most offspring, undesirable elements will die out of the population, and each individual will move closer to the global optimum. In this sense, these methods resemble the most brutal ideas of eugenics and Darwinism.

Various parameters can be tuned to guide the evolution. The method of sex, the probability of mutations, the chosen genetic representation, the size of the population, and so on all affect the converging speed and ability of the population.

In the training application here, the following setup was chosen. The genome was encoded as 118 bytes, each byte representing a floating-point value from -1 to 1 on a linear scale, giving it an accuracy of ~0.0078. The best half of the population was allowed to survive (the principle of elitism), and was also used as the source for eligible mates. Each individual was selected for mating based on its position when ordered by score, such that the fittest individual got to mate more than the second fittest, and so on.

The population size was kept constant, with the worse half being replaced with fresh children in each generation. Sex was implemented on a random per-gene basis, with each gene coming whole from one parent. The mutation probability was set at 0.1% per generation.

7 Software

A set of custom software was developed to aid in the research. This section covers each application shortly. Some helper libraries were used, however, the logic in each is self-made. The source code can be found at http://github.com/clbr/hotbos, under the AGPLv3 free software license.

All tools were written in C unless otherwise mentioned, and the Zlib compression library is used for reading/writing the memory traces in the binary format.

In the software of this section, there are about four thousand lines of C/C++, and slightly over hundred lines in shell scripts. This figure does not include the runtime code in the Linux kernel, Mesa, and associated libraries, which come to about 1.5k lines in total.

7.1 Activation function benchmark

In order to measure the CPU overhead of the three selected activation functions, this simple benchmark runs each 10^9 times while measuring the time taken. The timing used the **gettimeofday**() function, which has an accuracy of one microsecond.

As the total runtime is on the order of ten seconds, the accuracy is more than enough.

7.2 Text-to-binary format converter

The data traces recorded by the Mesa patch come in a simple but verbose text format. In order to reduce the disk space demands and to ease their handling in the network trainer, they were converted into a custom compressed binary format.

The format converter uses the Zlib library for compression. Zlib was chosen because it gives a good balance between decompression speed and compression ratio - these files will need to be repeatedly decompressed in the training phase, as they will not fit into RAM in their decompressed form all at once.

The common competing compression algorithms, XZ and Bzip2, produce higher compression ratios, however, at the expense of slower decompression and more memory usage. The compression speed was considered fairly irrelevant for these purposes.

The in-memory binary format takes advantage of delta compression, bit packing, and variable-sized indexing. This is then further compressed by Zlib for disk storage.

With knowledge of the data, the binary format can be much smaller than what any generic compressor would be able to do. For example, there are no created buffers over 2 GB in size, which allows the high-priority bit to be stored in the buffer size field.

Binary format specification:

```
/*
   All data is little-endian. No BE support.

   One entry takes two to four bytes, as follows:

        struct {
                u8 time: 5;
                u8 id: 3;
                u8/u16/u24 buffer;
        }

   Create entries are followed by four bytes:

        struct {
                u8 high_prio: 1;
                u32 size: 31;
        }
*/
```

The binary format filled the set goals quite nicely. It resulted in a compression ratio better than that of XZ applied on the text form, by a variable amount (1.5 to 12x). The compression ratio compared to uncompressed text varied from 150x to 1300x. It allows fast reading for the training and fragmentation benchmark applications.

7.3 Memory trace reader

To be able to easily study the traces in the binary format, a simple reader was developed. It prints the contents to the screen in a scrollable form, with optional color-coding for faster reading. (See Figure 19).

```
cpu op buffer 7 at 3 ms
create buffer 8 at 3 ms (8 bytes)
cpu op buffer 8 at 3 ms
create buffer 9 at 3 ms (48 bytes)
cpu op buffer 9 at 3 ms
create buffer 10 at 3 ms (48 bytes)
cpu op buffer 10 at 3 ms
create buffer 11 at 3 ms (24 bytes)
cpu op buffer 11 at 3 ms
create buffer 12 at 3 ms (32 bytes)
cpu op buffer 12 at 3 ms
create buffer 13 at 3 ms (8 bytes)
cpu op buffer 13 at 3 ms
create buffer 14 at 3 ms (16384 bytes)
create buffer 15 at 3 ms (65536 bytes)
create buffer 16 at 3 ms (16384 bytes)
write buffer 16 at 3 ms
create buffer 17 at 3 ms (4096 bytes, high priority)
destroy buffer 17 at 3 ms
destroy buffer 17 at 3 ms
destroy buffer 14 at 3 ms
cpu op buffer 7 at 11 ms
```

Figure 19. Sample from a memory trace.

7.4 Fragmentation benchmark

The fragmentation benchmark program runs all collected memory traces through the memory simulation code, using either LRU or min-max logic, measuring swapping and the number of fragments.

The output is in text form, and was further processed by some shell scripts before handing it to the fragmentation grapher.

Initially the memory simulation code was shared with the trainer, however, as the neural network capabilities were being added, they were branched so that each has its own copy. As such, the simulation code used by the benchmark was frozen to the state before the AI addition.

7.5 Fragmentation grapher

As the volume of data produced by the fragmentation benchmark was far too great for an office suite, even after scripted processing, a custom graphing tool was developed. It was made in C++, utilizing the FLTK toolkit for rendering.

The grapher renders a combined graph, with a line graph for the number of holes, and a horizontal bar graph for swapping. It takes approximately a minute to render one such graph, owing to the great amount of data.

In retrospect, the swapping bar graph is not a perfect form for showing swapping differences. Each pixel covers close to half a million memory operations, and if there was swapping during any of those, the pixel is drawn. This makes it hard to see any close differences.

7.6 Network trainer

The main program in this research, the AI trainer, was initially developed in C, however, later moved to C++ for easy access to the **multimap** data structure. The trainer is multi-threaded using the OpenMP library, which allows it to scale almost linearly to several dozen cores.

Beyond the well-done multi-threading, the data structure in the memory simulator bears mentioning. It combines several doubly-linked lists and a fixed array in such a way, that most operations can be done in O(1) complexity. Introducing this data structure sped up the simulation five-fold compared to simple linked lists.

Several runtime modes are supported:

- benchmark, measure the current AI state vs. LRU
- three different Monte-Carlo modes
- genetic mode.

It runs all collected memory traces through the memory simulation code, keeping track of the cost of each memory operation. The total cost of all traces is then used as the score for the tested critter, or in the Monte-Carlo modes, the score of the round. At the end of training (if the network converged in genetic mode, or the user requested exit), the results are printed similarly to the benchmark mode. See Figure 20 for a sample.

The current network constants are stored in a header file, **magic.h**, which can be dropped in to the runtime implementation for easy updates.

```
   64: Score went from 11373443721800562 to 13835903349739071 - -21.7% worse
  128: Score went from 8457631614472705 to 8897426058742106 - -5.2% worse
  256: Score went from 6823682679644306 to 6671306343161154 - 2.23% improvement
  384: Score went from 6366782903587257 to 6231975162922597 - 2.12% improvement
  512: Score went from 6103944126818047 to 6062687234429972 - 0.676% improvement
 1024: Score went from 5889442175382210 to 5895237176535220 - -0.0984% worse
 1536: Score went from 6091687353267279 to 6094802697922121 - -0.0511% worse
 2048: Score went from 6788258883170870 to 6763013725034131 - 0.372% improvement
 4096: Score went from 6768448221629419 to 6778450801742876 - -0.148% worse

Total: Score went from 64663321679772655 to 67230802550229248 - -3.97% worse
```

Figure 20. Sample output from benchmark mode.

8 Results

8.1 Min-max allocator

The initial research vector, reducing fragmentation by two-ended allocation, was successful. In no case did it do worse, and at best it could reduce eviction by up to 20%.

Initially implemented as an opaque decision inside the memory manager, it was suggested to change it to a placement flag instead, so that drivers could request top-down placement for other reasons in addition to the buffer's size.

For example, the page tables are never accessed by the CPU. As the CPU is limited to the first 256 MB of the VRAM (it cannot access more due to PCI-E limitations), the page tables were taking up valuable space. Under the new system, they may be allocated in the higher parts of VRAM, leaving more space for desired CPU-accessible content, such as texture updates.

In the new form, the change was accepted into future Linux kernels, starting with version 3.15.

8.2 AI

The main direction of this research, the neural network used to give buffers a score, was successful as well.

On average, the found solution gives approximately 1% improvement over LRU, as measured by the simulator. In specific cases improvements of up to 66% were measured. For most cases the performance is approximately equal to LRU; for 4% of the cases there are improvements; and for about 2% of the cases, the AI does measurably worse.

Taking a look at the simulated results over 2% in either direction (Figures 21 to 23), it can be seen that the changes concentrate on the lower VRAM sizes. It is also evident that the number of big losses is smaller than the number of big wins.

The simulator measures the total cost of a memory trace. In the following charts, the cost of the LRU run was divided by the cost of the AI run, so that 1 means no change, 1.2 means 20% improvement, and so on.

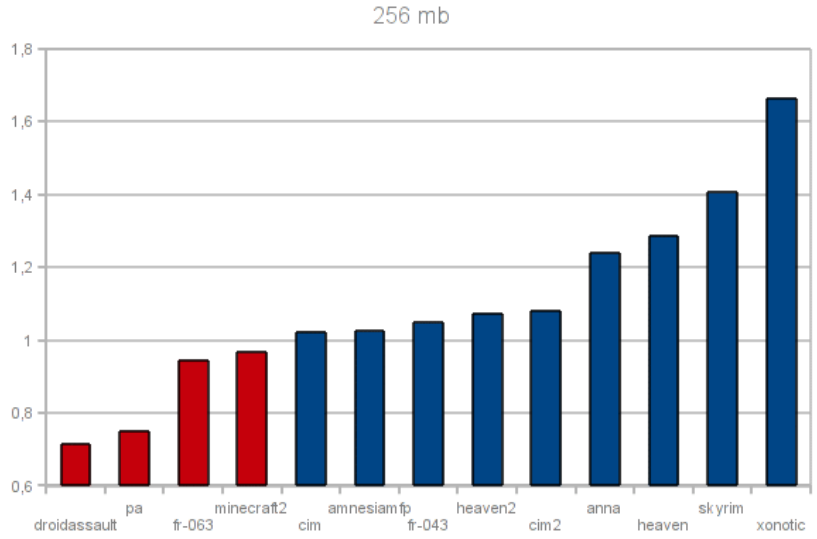

Figure 21. Major results in 256 MB.

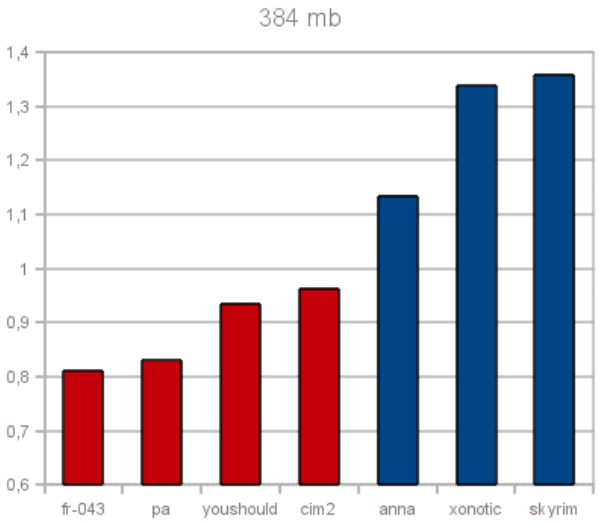

Figure 22. Major results in 384 MB.

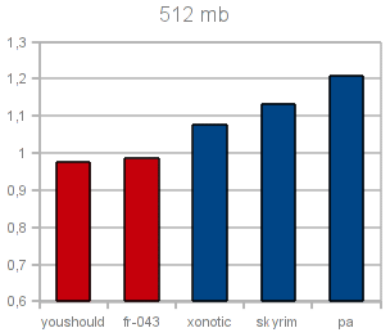

Figure 23. Major results in 512 MB.

Moving to testing with real hardware, it could be seen that the CPU overhead of calculating the score was negligible. Passing the score to the kernel initially took 3% of the CPU time, however, that could be optimized away by embedding the score in an existing call (the cs ioctl).

Likewise, the use of a priority queue in place of the LRU linked list did not measurably increase CPU overhead. The change turned a few operations from O(1) to O(log n), but the amount of buffers in VRAM at once is fairly low, in the single thousands.

The hardware specifications can be seen in Figure 24.

Figure 24. Test hardware

The Radeon driver allows one to limit VRAM to a value below what the card is actually capable of, which is great for testing such as this. In this test run, only 256 MB was tested, as the available tests do not require enough VRAM to cause memory pressure under higher VRAM amounts.

The tests here only changed Mesa, using the same kernel. Under the baseline target, named **256mb-master**, the kernel emulated the scoring in such a way that it essentially becomes LRU. No overhead was measured from the emulation compared to real LRU.

The benchmarking software used, Phoronix Test Suite, runs each test a minimum of three times, more if there is variance. This means that even fairly small differences, starting from around 0.5%, are statistically significant.

The tested applications consist of a set of freely available games.

Figure 25. Screenshots from tested applications.

Nexuiz is a first-person shooter game published by Alientrap, with fairly good-looking visuals, running on the DarkPlaces engine.

OpenArena, World of Padman, Smoking Guns, Tremulous, Urban Terror, and Warsow are shooters of various styles, based on the ioquake3 engine. Even though they share an engine, their visuals are rather different, as are their runtime demands.

VDrift is a driving simulator with a custom engine. It is fairly CPU-heavy.

Xonotic is a fork of Nexuiz with many improvements. Screenshots of Nexuiz, VDrift, and Xonotic in that order can be seen in Figure 25.

Xonotic is visually the heaviest application tested. It also allows several different quality presets to be tested, ranging from "Low" to "Ultimate" in four steps.

scoring

Phoronix Test Suite	256mb-master	256mb-score
Nexuiz	42.22	42.86
OpenArena	26.47	27.10
World of Padman	129.00	132.47
Smokin Guns	40.70	40.63
Tremulous	33.20	33.13
Urban Terror	53.13	47.73
VDrift	100.63	102.88
Warsow	44.63	45.33
Xonotic	124.23	126.65
Xonotic	55.60	56.65
Xonotic	47.75	48.57
Xonotic	36.19	36.60

PHORONIX-TEST-SUITE.COM

Figure 26. Test scores

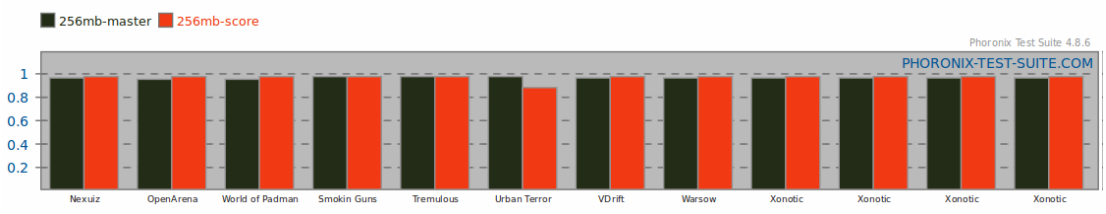

Figure 27. Scores arranged visually

In the results above (Figures 26 and 27), for most tests improvements of 1-2% can be seen. Xonotic in particular enjoys a constant improvement over all tested quality presets.

There were three regressions: Smoking Guns 0.2%, Tremulous 0.3%, and Urban Terror 10.2%. The first two can be considered as noise, given the very small difference.

The regression in Urban Terror is interesting, as that test was equal under the simulation. Whether there is a corresponding decrease in smoothness needs to be checked from the frame time.

8.2.1 Frame times

Of the tests run, two support frame time recording: OpenArena and Urban Terror. This allows one to gauge the smoothness in addition to raw throughput.

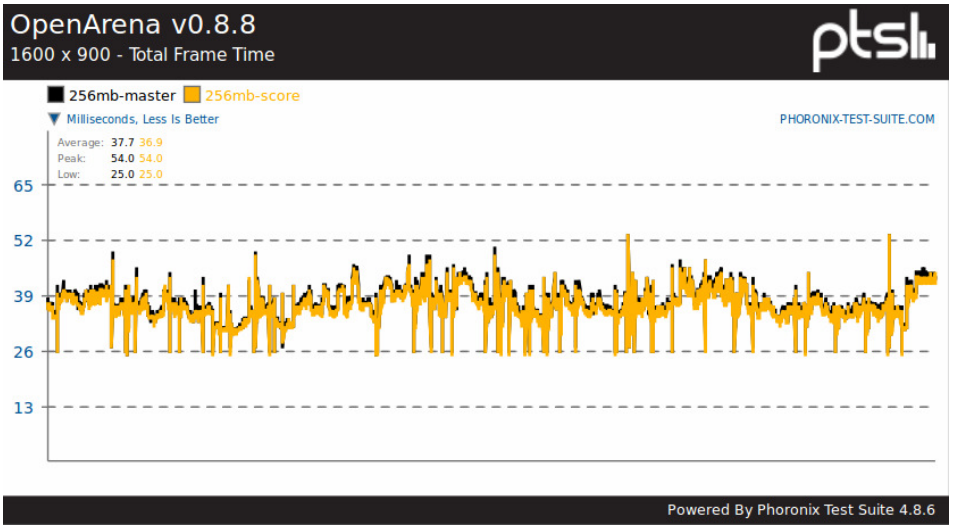

Figure 28. OpenArena frame time

In OpenArena, the average frame time was 0.8 ms lower, and most peaks are lower as well. The highest peak coincides with the baseline, however, it seems to be fairly rare - there are only two such peaks. Most peaks are visibly lower when compared to the baseline. (See Figure 28.)

For OpenArena, the goal of increasing smoothness was reached.

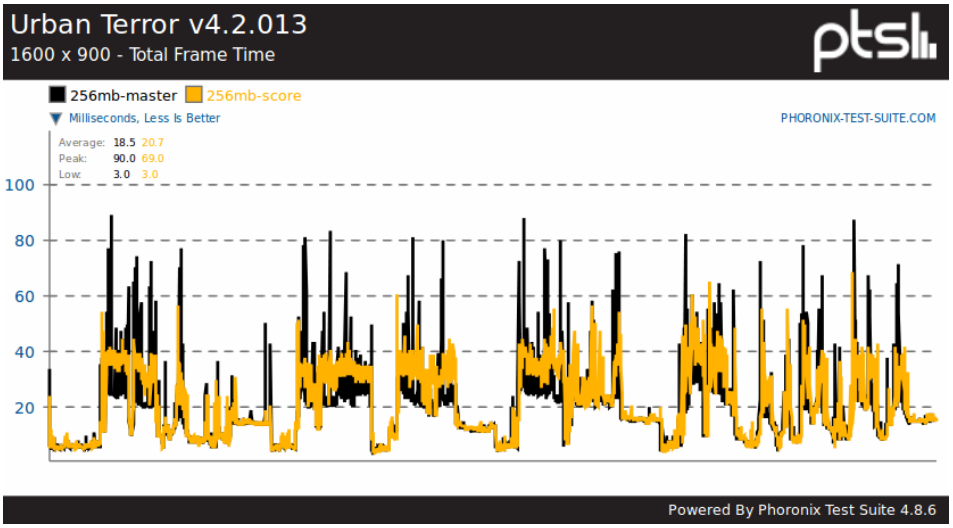

Figure 29. Urban Terror frame time

Urban Terror had regressed 10.2% in FPS, and here the frame time average is larger as well; however, the peaks are greatly lower, and there are less of them. The highest peak was 69 ms vs 90 ms. The smoothness here improved greatly. (See Figure 29.)

Given the improvement in smoothness was far greater than the decrease in throughput, even this case can be termed a success.

9 Discussion

During the initial training attempts, it became clear that a solution with this amount of nodes could not perform well simultaneously at the lowest end (64 MB and 128 MB VRAM sizes), and at the more common sizes. As the common graphics cards for sale are approaching 2 GB, and the wide installed base concentrates around 512 MB, the two lowest VRAM sizes were dropped from consideration.

In the training phase, it also became quite clear that the hardware available was inadequate: 6 cores and 8 GB of RAM were limiting the training speed. A setup with 64 cores and 128 GB of RAM is estimated as a sweet spot, but alas, one has to make do.

The training took almost three times longer than anticipated (8 weeks, when 3 were planned). Of course, given the problem space, 256^{118} combinations, copious computing time was to be expected.

In exploring unknown areas such as this, it is always a risk that no good solution exists. As the training took more time, that possibility caused worries, and aborting the experiment was considered. Luckily, a satisfactory solution was eventually found.

Examining the discovered solution, it was surprising to see that it discarded entirely six of the tried statistics. It found relevance only in the number of writes, number of CPU operations, and the VRAM size. Pruning the unused nodes cut the network size to 39% of the initial version.

Further, the connection of buffer writes deduced by the AI was extremely surprising: a small number of writes meant a *reduction* in score. As the number of writes increased, the score started to increase again (Figure 30). It is exactly this kind of new, unexpected connections that neural networks excel at, and which humans may never find.

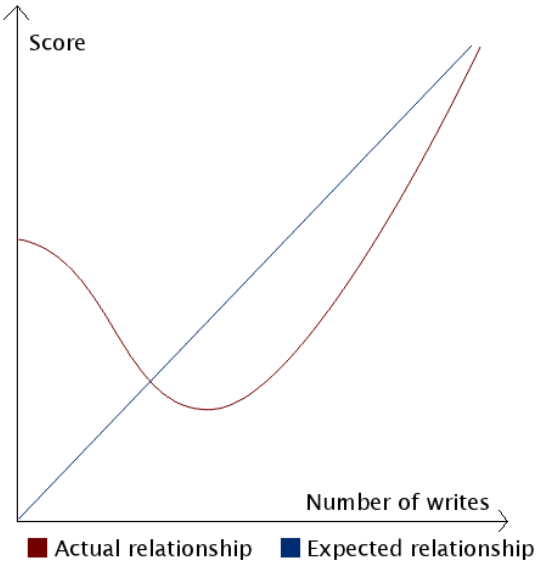

Figure 30. Expected vs. actual relation of writes to the buffer score.

All in all, the results fell slightly short of expectations. Given the difference to Catalyst, an improvement of about 5% was hoped for.

10 Future work and conclusion

With the enormous problem space, it cannot be said that the solution reached here is the global optimum; thus, three paths forward can be seen.

First, the parameters can be adequate, and merely more computing time is needed to find a better solution.

Second, it is possible the input parameters are adequate, but the processing power of the network is not. If so, it would need more hidden nodes, which would also mean slower training. This is hinted towards by the inability of the current network to do well at both 64 and 128 MB VRAM and the higher amounts.

Finally, it is possible that the input parameters are not the best possible. It may be that some different input can be easily gathered, and that input provides great correlation to a buffer's importance.

It was proved that LRU is not the most optimal solution, and that a neural network can exceed it. This opens up chances for other kinds of competition as well, outside that of artificial intelligence.

References

Bigus, J. 1996. Data Mining with Neural Networks. 1st ed. Indiana: McGraw-Hill.

Catalyst 13.1 release notes. 2013. AMD Knowledge Base. Accessed on Jan 15 2013. http://support.amd.com/en-us/kb-articles/Pages/AMDCatalystSoftwareSuiteVersion131.aspx

Corbet, J. 2010. Memory compaction. Accessed on Jan 20 2013. http://lwn.net/Articles/368869/

Dawe, L. 2014. Help Make Open Source AMD Graphics Drivers Better. Gaming on Linux. Accessed on 20 Jan 2013. http://www.gamingonlinux.com/articles/help-make-open-source-amd-graphics-drivers-better.2938

Karlic, B., Olgac. A. V. 2011. Performance analysis of various activation functions in generalized MLP architectures of neural networks. International journal of Artificial Intelligence and Expert Systems, volume 1, issue 4.

Kasanen, L. 2014. Radeon VRAM Optimizations Coming, But Help Is Needed. Phoronix. Accessed on Jan 20 2013. http://www.phoronix.com/scan.php?page=news_item&px=MTU2Nzk

Larabel, M. 2012. Ubuntu 12.10: Open-Source Radeon vs. AMD Catalyst Performance. Phoronix. Accessed on Jan 15 2013. http://www.phoronix.com/scan.php?page=article&item=ubuntu_1210_amdstock&num=3

Larabel, M. 2013a. AMD's Radeon Gallium3D Starts Posing A Threat To Catalyst. Phoronix. Accessed on Jan 15 2013. http://www.phoronix.com/scan.php?page=article&item=amd_catalyst_gallium80

Larabel, M. 2013b. CS Memory Accounting For Radeon Gallium3D. Phoronix. Accessed on Jan 20 2013. http://www.phoronix.com/scan.php?page=news_item&px=MTI4OTM

Liu, J., Gader, P. D. 2000. Outlier Rejection with MLPs and Variants of RBF Networks. International Conference on Pattern Recognition pp. 2680-2683.

Mahoney, M. 1996. Fast text compression with neural networks. Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference.

Minimum playable FPS. 2012. Whirlpool forums. Accessed on Jan 15 2013. http://forums.whirlpool.net.au/archive/1890684

Siddique, N., Adeli, H. 2013. Synergies of fuzzy logic, neural networks and evolutionary computing. 1st ed. UK: John Wiley & Sons.

Tesauro, G. 1994. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. Neural Computation 6, 2 (March 1994)

Very low FPS when video memory is full (GART & ram <-> vram swapping). 2013. FreeDesktop.org Bugzilla. Accessed on Jan 15 2013. https://bugs.freedesktop.org/show_bug.cgi?id=66632

Wiering, M., Hasselt, H., Pietersma A.-D., Schomaker, L. 2011. Reinforcement Learning Algorithms for solving Classification Problems. Adaptive Dynamic Programming And Reinforcement Learning, 2011 IEEE Symposium

Appendix A. Data statistics

In total, about 60 GB of data was collected. In a delta-compressed memory format, it takes about 16 GB; in the custom storage format, 830 MB.

Details and statistics on each memory trace:

0ad1.bin:
7146 buffers, runtime 626717 ms (~10.4 minutes)
7146 creates, 99612946 cpu ops, 221797452 reads, 2833214 writes, 6382 destroys
11.4023 creates/s, 158944 cpu ops/s, 353904 reads/s, 4520.72 writes/s, 10.1832 destroys/s

1995_1.bin: 1366x768
5573 buffers, runtime 275870 ms (~4.6 minutes)
5573 creates, 1441547 cpu ops, 3115245 reads, 461136 writes, 5371 destroys
20.2015 creates/s, 5225.46 cpu ops/s, 11292.4 reads/s, 1671.57 writes/s, 19.4693 destroys/s

1finger1.bin: 1366x768
3173 buffers, runtime 111358 ms (~1.9 minutes)
3173 creates, 246217 cpu ops, 621039 reads, 96671 writes, 2746 destroys
28.4937 creates/s, 2211.04 cpu ops/s, 5576.96 reads/s, 868.11 writes/s, 24.6592 destroys/s

471110_1.bin: 640x480
16010 buffers, runtime 220284 ms (~3.7 minutes)
16010 creates, 1677829 cpu ops, 10761557 reads, 109635 writes, 15957 destroys
72.6789 creates/s, 7616.66 cpu ops/s, 48853.1 reads/s, 497.698 writes/s, 72.4383 destroys/s

aaa1.bin: AAAaaaAAAaaa for the Awesome 1280x800 lowest
9252 buffers, runtime 250385 ms (~4.2 minutes)
9252 creates, 914870 cpu ops, 2119803 reads, 53523 writes, 5808 destroys
36.9511 creates/s, 3653.85 cpu ops/s, 8466.17 reads/s, 213.763 writes/s, 23.1963 destroys/s

altitude1.bin: 1280x800 highest
9301 buffers, runtime 355298 ms (~5.9 minutes)
9301 creates, 4940980 cpu ops, 9562028 reads, 45355 writes, 2559 destroys
26.178 creates/s, 13906.6 cpu ops/s, 26912.7 reads/s, 127.653 writes/s, 7.2024 destroys/s

amnesia1.bin: Amnesia: The dark descent
43108 buffers, runtime 1164945 ms (~19.4 minutes)
43108 creates, 6864013 cpu ops, 24936540 reads, 2250880 writes, 42609 destroys
37.0045 creates/s, 5892.16 cpu ops/s, 21405.9 reads/s, 1932.19 writes/s, 36.5761 destroys/s

amnesiamfp1.bin: Amnesia: A machine for pigs
26210 buffers, runtime 284461 ms (~4.7 minutes)
26210 creates, 7333150 cpu ops, 26498259 reads, 1874116 writes, 25080 destroys
92.1392 creates/s, 25779.1 cpu ops/s, 93152.5 reads/s, 6588.31 writes/s, 88.1667 destroys/s

anna1.bin: Anna extended version
20520 buffers, runtime 529469 ms (~8.8 minutes)
20520 creates, 13425313 cpu ops, 33238877 reads, 1598824 writes, 17093 destroys
38.7558 creates/s, 25356.2 cpu ops/s, 62777.8 reads/s, 3019.67 writes/s, 32.2833 destroys/s

anomaly2_1.bin:
18871 buffers, runtime 1283439 ms (~21.4 minutes)
18871 creates, 8312709 cpu ops, 30039021 reads, 689442 writes, 9566 destroys
14.7035 creates/s, 6476.9 cpu ops/s, 23405.1 reads/s, 537.183 writes/s, 7.45341 destroys/s

anomalywe1.bin: Anomaly Warzone Earth 1280x800 medium
15748 buffers, runtime 745212 ms (~12.4 minutes)
15748 creates, 8119419 cpu ops, 26148915 reads, 609975 writes, 8612 destroys
21.1322 creates/s, 10895.4 cpu ops/s, 35089.2 reads/s, 818.525 writes/s, 11.5564 destroys/s

aquaria1.bin:
56260 buffers, runtime 1670661 ms (~27.8 minutes)
56260 creates, 77000054 cpu ops, 52366501 reads, 1327605 writes, 56199 destroys
33.6753 creates/s, 46089.6 cpu ops/s, 31344.8 reads/s, 794.659 writes/s, 33.6388 destroys/s

assassin2_1.bin: No AA, shadows low, others max
96464 buffers, runtime 313322 ms (~5.2 minutes)
96464 creates, 16572977 cpu ops, 46558238 reads, 14797233 writes, 76940 destroys
307.875 creates/s, 52894.4 cpu ops/s, 148595 reads/s, 47226.9 writes/s, 245.562 destroys/s

avadon1.bin:
2099 buffers, runtime 1987402 ms (~33.1 minutes)
2099 creates, 92823145 cpu ops, 167751909 reads, 844353 writes, 2087 destroys
1.05615 creates/s, 46705.8 cpu ops/s, 84407.7 reads/s, 424.853 writes/s, 1.05012 destroys/s

awesomenauts1.bin: 1680x1050
9954 buffers, runtime 475623 ms (~7.9 minutes)
9954 creates, 16184830 cpu ops, 28928799 reads, 354491 writes, 7238 destroys
20.9283 creates/s, 34028.7 cpu ops/s, 60823 reads/s, 745.319 writes/s, 15.2179 destroys/s

badhotel1.bin:
701 buffers, runtime 806671 ms (~13.4 minutes)
701 creates, 335227 cpu ops, 537287 reads, 238459 writes, 639 destroys
0.869004 creates/s, 415.568 cpu ops/s, 666.055 reads/s, 295.609 writes/s, 0.792145 destroys/s

bastion1.bin: 1680x1050
2697 buffers, runtime 414524 ms (~6.9 minutes)
2697 creates, 5000417 cpu ops, 10032086 reads, 447846 writes, 3508 destroys
6.50626 creates/s, 12063 cpu ops/s, 24201.5 reads/s, 1080.39 writes/s, 8.46272 destroys/s

beathazard1.bin:
450 buffers, runtime 214003 ms (~3.6 minutes)
450 creates, 626289 cpu ops, 1277221 reads, 76510 writes, 272 destroys
2.10277 creates/s, 2926.54 cpu ops/s, 5968.24 reads/s, 357.518 writes/s, 1.27101 destroys/s

blackplague1.bin: Penumbra Black Plague
90760 buffers, runtime 3159040 ms (~52.7 minutes)
90760 creates, 10880143 cpu ops, 52930650 reads, 2315828 writes, 86884 destroys
28.7302 creates/s, 3444.13 cpu ops/s, 16755.3 reads/s, 733.08 writes/s, 27.5033 destroys/s

brokenage1.bin:
9793 buffers, runtime 454253 ms (~7.6 minutes)
9793 creates, 1390232 cpu ops, 5461675 reads, 224039 writes, 9376 destroys
21.5585 creates/s, 3060.48 cpu ops/s, 12023.4 reads/s, 493.203 writes/s, 20.6405 destroys/s

capsized1.bin:
5912 buffers, runtime 243469 ms (~4.1 minutes)
5912 creates, 538826 cpu ops, 1188568 reads, 204875 writes, 5861 destroys
24.2824 creates/s, 2213.12 cpu ops/s, 4881.8 reads/s, 841.483 writes/s, 24.0729 destroys/s

cim2_1.bin: Cities in Motion 2, AA on
34880 buffers, runtime 235837 ms (~3.9 minutes)
34880 creates, 20610385 cpu ops, 69527604 reads, 374507 writes, 30566 destroys
147.899 creates/s, 87392.5 cpu ops/s, 294812 reads/s, 1587.99 writes/s, 129.606 destroys/s

cim_1.bin: Cities in Motion 1, AA on
23977 buffers, runtime 54989 ms (~0.9 minutes)
23977 creates, 1845952 cpu ops, 4244715 reads, 48960 writes, 14172 destroys
436.033 creates/s, 33569.5 cpu ops/s, 77192.1 reads/s, 890.36 writes/s, 257.724 destroys/s

cogs1.bin: 1280x800 low
962 buffers, runtime 114915 ms (~1.9 minutes)
962 creates, 10094327 cpu ops, 23207662 reads, 141754 writes, 903 destroys
8.3714 creates/s, 87841.7 cpu ops/s, 201955 reads/s, 1233.56 writes/s, 7.85798 destroys/s

costumequest1.bin:
18610 buffers, runtime 991851 ms (~16.5 minutes)
18610 creates, 5778511 cpu ops, 27024325 reads, 710384 writes, 8828 destroys
18.7629 creates/s, 5825.99 cpu ops/s, 27246.4 reads/s, 716.22 writes/s, 8.90053 destroys/s

css1.bin: Counter-Strike Source 1680x1050
101289 buffers, runtime 423090 ms (~7.1 minutes)
101289 creates, 18682695 cpu ops, 58776135 reads, 3935324 writes, 94445 destroys
239.403 creates/s, 44157.7 cpu ops/s, 138921 reads/s, 9301.39 writes/s, 223.227 destroys/s

css2.bin: 1280x800
20109 buffers, runtime 202185 ms (~3.4 minutes)
20109 creates, 3483404 cpu ops, 11939843 reads, 381430 writes, 19922 destroys
99.4584 creates/s, 17228.8 cpu ops/s, 59054 reads/s, 1886.54 writes/s, 98.5335 destroys/s

darwinia1.bin:
23379 buffers, runtime 557596 ms (~9.3 minutes)
23379 creates, 12620525 cpu ops, 19909313 reads, 398361 writes, 23088 destroys
41.9282 creates/s, 22633.8 cpu ops/s, 35705.6 reads/s, 714.426 writes/s, 41.4063 destroys/s

dayofdefeat1.bin: 1280x800
12249 buffers, runtime 326931 ms (~5.4 minutes)
12249 creates, 14209267 cpu ops, 25994278 reads, 362432 writes, 10919 destroys
37.4666 creates/s, 43462.6 cpu ops/s, 79510 reads/s, 1108.59 writes/s, 33.3985 destroys/s

defcon1.bin: 1280x800
12473 buffers, runtime 606759 ms (~10.1 minutes)
12473 creates, 14876353 cpu ops, 23738555 reads, 419791 writes, 634 destroys
20.5568 creates/s, 24517.7 cpu ops/s, 39123.5 reads/s, 691.858 writes/s, 1.0449 destroys/s

dota2_1.bin: 1366x768 all max
42608 buffers, runtime 488330 ms (~8.1 minutes)
42608 creates, 25168716 cpu ops, 75314458 reads, 3407014 writes, 39140 destroys
87.2525 creates/s, 51540.4 cpu ops/s, 154229 reads/s, 6976.87 writes/s, 80.1507 destroys/s

dota2_2.bin: 1366x768 low shadows, others max
46164 buffers, runtime 606763 ms (~10.1 minutes)
46164 creates, 33690629 cpu ops, 95263522 reads, 4689068 writes, 42764 destroys
76.0824 creates/s, 55525.2 cpu ops/s, 157003 reads/s, 7728.01 writes/s, 70.4789 destroys/s

dota2_3.bin: 1920x1080 low shadows, others max
44718 buffers, runtime 645179 ms (~10.8 minutes)
44718 creates, 25107968 cpu ops, 72107259 reads, 3529311 writes, 41382 destroys
69.311 creates/s, 38916.3 cpu ops/s, 111763 reads/s, 5470.28 writes/s, 64.1403 destroys/s

dreamchild1.bin: 800x600
2183 buffers, runtime 248116 ms (~4.1 minutes)
2183 creates, 10208900 cpu ops, 14346802 reads, 85547 writes, 1769 destroys
8.7983 creates/s, 41145.7 cpu ops/s, 57823 reads/s, 344.786 writes/s, 7.12973 destroys/s

droidassault1.bin:
35759 buffers, runtime 966990 ms (~16.1 minutes)
35759 creates, 1180961 cpu ops, 7004758 reads, 652838 writes, 28588 destroys
36.9797 creates/s, 1221.28 cpu ops/s, 7243.88 reads/s, 675.124 writes/s, 29.5639 destroys/s

dub1.bin: 1366x768
11037 buffers, runtime 346937 ms (~5.8 minutes)
11037 creates, 777423 cpu ops, 1553133 reads, 84530 writes, 10998 destroys
31.8127 creates/s, 2240.82 cpu ops/s, 4476.7 reads/s, 243.647 writes/s, 31.7003 destroys/s

dungeondefenders1.bin:
89381 buffers, runtime 1587944 ms (~26.5 minutes)
89381 creates, 24980362 cpu ops, 76450150 reads, 4823395 writes, 88448 destroys
56.2874 creates/s, 15731.3 cpu ops/s, 48144.2 reads/s, 3037.52 writes/s, 55.6998 destroys/s

dynamitejack1.bin:
22738 buffers, runtime 976838 ms (~16.3 minutes)
22738 creates, 4876882 cpu ops, 6319728 reads, 139538 writes, 11540 destroys
23.2771 creates/s, 4992.52 cpu ops/s, 6469.58 reads/s, 142.847 writes/s, 11.8136 destroys/s

etqw1.bin: 1680*1050 all max
163691 buffers, runtime 228847 ms (~3.8 minutes)
163691 creates, 19586219 cpu ops, 56622992 reads, 4922848 writes, 156739 destroys
715.286 creates/s, 85586.5 cpu ops/s, 247427 reads/s, 21511.5 writes/s, 684.907 destroys/s

fez1.bin: 1366x768
41111 buffers, runtime 923150 ms (~15.4 minutes)
41111 creates, 12297160 cpu ops, 29500988 reads, 529001 writes, 4106 destroys
44.5334 creates/s, 13320.9 cpu ops/s, 31956.9 reads/s, 573.039 writes/s, 4.44781 destroys/s

fortrix2_1.bin: 1280x800
23813 buffers, runtime 1274525 ms (~21.2 minutes)
23813 creates, 1649199 cpu ops, 5641416 reads, 397754 writes, 9661 destroys
18.6838 creates/s, 1293.97 cpu ops/s, 4426.27 reads/s, 312.079 writes/s, 7.58005 destroys/s

fr-025_1.bin: 1366x768
56462 buffers, runtime 228169 ms (~3.8 minutes)
56462 creates, 4887837 cpu ops, 12310201 reads, 918191 writes, 55488 destroys
247.457 creates/s, 21422 cpu ops/s, 53952.1 reads/s, 4024.17 writes/s, 243.188 destroys/s

fr-043_1.bin: 1366x768
2493 buffers, runtime 35411 ms (~0.6 minutes)
2493 creates, 25707 cpu ops, 80828 reads, 9588 writes, 1750 destroys
70.4019 creates/s, 725.961 cpu ops/s, 2282.57 reads/s, 270.763 writes/s, 49.4197 destroys/s

fr-062_1.bin: 1366x768
3513 buffers, runtime 89679 ms (~1.5 minutes)
3513 creates, 57337 cpu ops, 188696 reads, 39096 writes, 3370 destroys
39.1731 creates/s, 639.358 cpu ops/s, 2104.13 reads/s, 435.955 writes/s, 37.5785 destroys/s

fr-063_1.bin: 1366x768 CSMT disabled
6778 buffers, runtime 185501 ms (~3.1 minutes)
6778 creates, 242007 cpu ops, 1176481 reads, 71363 writes, 5489 destroys
36.5389 creates/s, 1304.61 cpu ops/s, 6342.18 reads/s, 384.704 writes/s, 29.5901 destroys/s

frozensynapse1.bin: 1280x800
15505 buffers, runtime 574674 ms (~9.6 minutes)
15505 creates, 18428602 cpu ops, 28437521 reads, 202251 writes, 10811 destroys
26.9805 creates/s, 32067.9 cpu ops/s, 49484.6 reads/s, 351.94 writes/s, 18.8124 destroys/s

galconfusion1.bin: 1280x800
19184 buffers, runtime 789742 ms (~13.2 minutes)
19184 creates, 9486372 cpu ops, 12788079 reads, 121168 writes, 9362 destroys
24.2915 creates/s, 12012 cpu ops/s, 16192.7 reads/s, 153.427 writes/s, 11.8545 destroys/s

gateways1.bin:
4768 buffers, runtime 325553 ms (~5.4 minutes)
4768 creates, 480743 cpu ops, 1419793 reads, 188684 writes, 820 destroys
14.6458 creates/s, 1476.7 cpu ops/s, 4361.17 reads/s, 579.58 writes/s, 2.51879 destroys/s

glxgears1.bin: default res 300x300
32 buffers, runtime 5323 ms (~0.1 minutes)
32 creates, 138356 cpu ops, 360784 reads, 36081 writes, 3 destroys
6.01165 creates/s, 25992.1 cpu ops/s, 67778.3 reads/s, 6778.32 writes/s, 0.563592 destroys/s

glxgears2.bin: 1024x1024
32 buffers, runtime 6176 ms (~0.1 minutes)
32 creates, 8571 cpu ops, 22264 reads, 2229 writes, 4 destroys
5.18135 creates/s, 1387.79 cpu ops/s, 3604.92 reads/s, 360.913 writes/s, 0.647668 destroys/s

heaven1.bin: heaven 3.0 1920x1080, AF16x, shaders high, no AA
18977 buffers, runtime 81739 ms (~1.4 minutes)
18977 creates, 5184725 cpu ops, 19045672 reads, 207026 writes, 18691 destroys
232.166 creates/s, 63430.2 cpu ops/s, 233006 reads/s, 2532.77 writes/s, 228.667 destroys/s

heaven2.bin: heaven 3.0 1920x1080, AF16x, shaders high, no AA
15409 buffers, runtime 62901 ms (~1.0 minutes)
15409 creates, 3506809 cpu ops, 10173322 reads, 227019 writes, 15102 destroys
244.972 creates/s, 55751.2 cpu ops/s, 161735 reads/s, 3609.15 writes/s, 240.092 destroys/s

hl1.bin:
17380 buffers, runtime 250279 ms (~4.2 minutes)
17380 creates, 4328751 cpu ops, 9068139 reads, 141386 writes, 13978 destroys
69.4425 creates/s, 17295.7 cpu ops/s, 36232.1 reads/s, 564.914 writes/s, 55.8497 destroys/s

hl2ep2_1.bin: 1680x1050
45391 buffers, runtime 372576 ms (~6.2 minutes)
45391 creates, 8847952 cpu ops, 35169020 reads, 2379285 writes, 44783 destroys
121.83 creates/s, 23748 cpu ops/s, 94394.2 reads/s, 6386.04 writes/s, 120.198 destroys/s

kerbal1.bin: 0.18.3 demo
42432 buffers, runtime 509566 ms (~8.5 minutes)
42432 creates, 21653280 cpu ops, 35377597 reads, 2011964 writes, 27669 destroys
83.2709 creates/s, 42493.6 cpu ops/s, 69426.9 reads/s, 3948.39 writes/s, 54.2991 destroys/s

killingfloor1.bin: 1280x800, lowest
18359 buffers, runtime 956582 ms (~15.9 minutes)
18359 creates, 9136927 cpu ops, 16486720 reads, 151201 writes, 9499 destroys
19.1923 creates/s, 9551.64 cpu ops/s, 17235 reads/s, 158.064 writes/s, 9.93015 destroys/s

left4dead2_1.bin: 2.1.3.5 Dec 2013, 1920x1080, AA off, AF off, others med-low
43771 buffers, runtime 262573 ms (~4.4 minutes)
43771 creates, 11191310 cpu ops, 40006242 reads, 1639841 writes, 42437 destroys
166.7 creates/s, 42621.7 cpu ops/s, 152362 reads/s, 6245.28 writes/s, 161.62 destroys/s

left4dead2_2.bin: 1680x1050
107979 buffers, runtime 430462 ms (~7.2 minutes)
107979 creates, 17506153 cpu ops, 57786594 reads, 3734898 writes, 118183 destroys
250.844 creates/s, 40668.3 cpu ops/s, 134243 reads/s, 8676.49 writes/s, 274.549 destroys/s

lightsmark1.bin: 2008 1600x900
11011 buffers, runtime 41105 ms (~0.7 minutes)
11011 creates, 12205771 cpu ops, 19842333 reads, 457080 writes, 10381 destroys
267.875 creates/s, 296941 cpu ops/s, 482723 reads/s, 11119.8 writes/s, 252.548 destroys/s

lugaru1.bin: 1680x1050
736 buffers, runtime 326224 ms (~5.4 minutes)
736 creates, 22074717 cpu ops, 44010547 reads, 147428 writes, 655 destroys
2.25612 creates/s, 67667.4 cpu ops/s, 134909 reads/s, 451.923 writes/s, 2.00782 destroys/s

minecraft1.bin: 1.7.4
233112 buffers, runtime 378591 ms (~6.3 minutes)
233112 creates, 39786967 cpu ops, 73145225 reads, 16851274 writes, 216618 destroys
615.736 creates/s, 105092 cpu ops/s, 193204 reads/s, 44510.5 writes/s, 572.169 destroys/s

minecraft2.bin: Sonic Ether GLSL deferred mod
121314 buffers, runtime 200004 ms (~3.3 minutes)
121314 creates, 34958394 cpu ops, 47989789 reads, 10320624 writes, 109569 destroys
606.558 creates/s, 174788 cpu ops/s, 239944 reads/s, 51602.1 writes/s, 547.834 destroys/s

muoto1.bin: 1366x768 2xMSAA
9796 buffers, runtime 559567 ms (~9.3 minutes)
9796 creates, 2677823 cpu ops, 3727288 reads, 85564 writes, 9717 destroys
17.5064 creates/s, 4785.53 cpu ops/s, 6661.02 reads/s, 152.911 writes/s, 17.3652 destroys/s

nederland1.bin: 1366x768
3166 buffers, runtime 27554 ms (~0.5 minutes)
3166 creates, 37799 cpu ops, 82153 reads, 14441 writes, 3032 destroys
114.902 creates/s, 1371.82 cpu ops/s, 2981.53 reads/s, 524.098 writes/s, 110.038 destroys/s

nexuiz1.bin: 1680x1050
3881 buffers, runtime 30212 ms (~0.5 minutes)
3881 creates, 1670369 cpu ops, 2809243 reads, 45932 writes, 3618 destroys
128.459 creates/s, 55288.3 cpu ops/s, 92984.3 reads/s, 1520.32 writes/s, 119.754 destroys/s

openarena1.bin: 1680x1050
28167 buffers, runtime 96698 ms (~1.6 minutes)
28167 creates, 7189303 cpu ops, 11562523 reads, 180637 writes, 28120 destroys
291.288 creates/s, 74348 cpu ops/s, 119574 reads/s, 1868.05 writes/s, 290.802 destroys/s

osmos1.bin:
22573 buffers, runtime 945099 ms (~15.8 minutes)
22573 creates, 2913147 cpu ops, 5194245 reads, 97815 writes, 12165 destroys
23.8843 creates/s, 3082.37 cpu ops/s, 5495.98 reads/s, 103.497 writes/s, 12.8717 destroys/s

pa1.bin: Planetary Annihilation v58772, 1920x1200, AA off, others max
45899 buffers, runtime 356410 ms (~5.9 minutes)
45899 creates, 14865342 cpu ops, 42849015 reads, 1067966 writes, 44909 destroys
128.781 creates/s, 41708.5 cpu ops/s, 120224 reads/s, 2996.45 writes/s, 126.004 destroys/s

party_heart1.bin: 1366x768
1135 buffers, runtime 22844 ms (~0.4 minutes)
1135 creates, 348676 cpu ops, 1094953 reads, 29085 writes, 356 destroys
49.6848 creates/s, 15263.4 cpu ops/s, 47931.8 reads/s, 1273.2 writes/s, 15.584 destroys/s

penumbra1.bin: Penumbra Overture
43391 buffers, runtime 1245889 ms (~20.8 minutes)
43391 creates, 7030974 cpu ops, 24562594 reads, 1093264 writes, 41585 destroys
34.8273 creates/s, 5643.33 cpu ops/s, 19714.9 reads/s, 877.496 writes/s, 33.3777 destroys/s

portal1.bin:
52220 buffers, runtime 1174949 ms (~19.6 minutes)
52220 creates, 4438903 cpu ops, 16722290 reads, 1522403 writes, 48040 destroys
44.4444 creates/s, 3777.95 cpu ops/s, 14232.3 reads/s, 1295.72 writes/s, 40.8868 destroys/s

runner2_1.bin:
2889 buffers, runtime 332467 ms (~5.5 minutes)
2889 creates, 6228468 cpu ops, 19476499 reads, 484490 writes, 2041 destroys
8.68958 creates/s, 18734.1 cpu ops/s, 58581.8 reads/s, 1457.26 writes/s, 6.13896 destroys/s

sam3_1.bin:
31793 buffers, runtime 392958 ms (~6.5 minutes)
31793 creates, 25691533 cpu ops, 74516950 reads, 9283906 writes, 36180 destroys
80.9069 creates/s, 65379.8 cpu ops/s, 189631 reads/s, 23625.7 writes/s, 92.0709 destroys/s

shatter1.bin:
10955 buffers, runtime 609124 ms (~10.2 minutes)
10955 creates, 42113301 cpu ops, 131868976 reads, 1356390 writes, 9982 destroys
17.9848 creates/s, 69137.5 cpu ops/s, 216490 reads/s, 2226.79 writes/s, 16.3875 destroys/s

skyrim1.bin: Wine git, AA off, others max
26782 buffers, runtime 252779 ms (~4.2 minutes)
26782 creates, 19579474 cpu ops, 56841792 reads, 1523215 writes, 19522 destroys
105.95 creates/s, 77456.9 cpu ops/s, 224868 reads/s, 6025.88 writes/s, 77.2295 destroys/s

smokingguns1.bin: 1680x1050
3304 buffers, runtime 54131 ms (~0.9 minutes)
3304 creates, 3254020 cpu ops, 4781877 reads, 39733 writes, 3261 destroys
61.0371 creates/s, 60113.8 cpu ops/s, 88339 reads/s, 734.016 writes/s, 60.2427 destroys/s

spectraball1.bin:
24396 buffers, runtime 1254921 ms (~20.9 minutes)
24396 creates, 8317176 cpu ops, 21536280 reads, 1448571 writes, 12254 destroys
19.4403 creates/s, 6627.65 cpu ops/s, 17161.5 reads/s, 1154.31 writes/s, 9.76477 destroys/s

spin1.bin: 1366x768 AA disabled
6263 buffers, runtime 332997 ms (~5.5 minutes)
6263 creates, 3945728 cpu ops, 5784178 reads, 156554 writes, 6137 destroys
18.808 creates/s, 11849.1 cpu ops/s, 17370.1 reads/s, 470.136 writes/s, 18.4296 destroys/s

splice1.bin:
14214 buffers, runtime 549177 ms (~9.2 minutes)
14214 creates, 3200172 cpu ops, 5314888 reads, 802671 writes, 5301 destroys
25.8824 creates/s, 5827.21 cpu ops/s, 9677.91 reads/s, 1461.59 writes/s, 9.65263 destroys/s

stargazer1.bin: 1366x768
10431 buffers, runtime 340095 ms (~5.7 minutes)
10431 creates, 941067 cpu ops, 4019647 reads, 452470 writes, 9772 destroys
30.6708 creates/s, 2767.07 cpu ops/s, 11819.2 reads/s, 1330.42 writes/s, 28.7331 destroys/s

supertuxkart1.bin: STK cand git, mid settings, Hacienda level
1042 buffers, runtime 70860 ms (~1.2 minutes)
1042 creates, 1718798 cpu ops, 3054361 reads, 230699 writes, 839 destroys
14.7051 creates/s, 24256.3 cpu ops/s, 43104.2 reads/s, 3255.7 writes/s, 11.8402 destroys/s

supertuxkart2.bin: STK cand git, mid settings, Jungle level
995 buffers, runtime 62564 ms (~1.0 minutes)
995 creates, 990683 cpu ops, 1859964 reads, 198964 writes, 803 destroys
15.9037 creates/s, 15834.7 cpu ops/s, 29729 reads/s, 3180.17 writes/s, 12.8349 destroys/s

surgeonsim1.bin: Surgeon Simulator 2013
4779 buffers, runtime 482913 ms (~8.0 minutes)
4779 creates, 17014187 cpu ops, 42367908 reads, 1733657 writes, 3958 destroys
9.89619 creates/s, 35232.4 cpu ops/s, 87734 reads/s, 3590 writes/s, 8.19609 destroys/s

tf2_1.bin: 1680x1050
163790 buffers, runtime 282378 ms (~4.7 minutes)
163790 creates, 2616594 cpu ops, 6579247 reads, 370589 writes, 163786 destroys
580.038 creates/s, 9266.28 cpu ops/s, 23299.4 reads/s, 1312.39 writes/s, 580.024 destroys/s

tf2_2.bin: pl_barnblitz map
83916 buffers, runtime at 739482 ms (~12.3 minutes)
83916 creates, 26092639 cpu ops, 93014190 reads, 4167907 writes, 73393 destroys
113.479 creates/s, 35285 cpu ops/s, 125783 reads/s, 170.096 writes/s, 99.2492 destroys/s

thomas1.bin: And Thomas was alone
31812 buffers, runtime 979275 ms (~16.3 minutes)
31812 creates, 6528456 cpu ops, 7966090 reads, 427896 writes, 31667 destroys
32.4853 creates/s, 6666.62 cpu ops/s, 8134.68 reads/s, 436.952 writes/s, 32.3372 destroys/s

tremulous1.bin: 1680x1050
3386 buffers, runtime 64854 ms (~1.1 minutes)
3386 creates, 2029521 cpu ops, 2875981 reads, 22162 writes, 3349 destroys
52.2096 creates/s, 31293.7 cpu ops/s, 44345.5 reads/s, 341.721 writes/s, 51.6391 destroys/s

trine2_1.bin: 1680x1050
25716 buffers, runtime 228570 ms (~3.8 minutes)
25716 creates, 1327581 cpu ops, 5952135 reads, 77524 writes, 23402 destroys
112.508 creates/s, 5808.2 cpu ops/s, 26040.8 reads/s, 339.17 writes/s, 102.384 destroys/s

urbanterror1.bin: 1680x1050
6073 buffers, runtime 201869 ms (~3.4 minutes)
6073 creates, 25001394 cpu ops, 29069898 reads, 116012 writes, 6023 destroys
30.0839 creates/s, 123850 cpu ops/s, 144004 reads/s, 574.69 writes/s, 29.8362 destroys/s

worldofgoo1.bin:
24529 buffers, runtime 1125350 ms (~18.8 minutes)
24529 creates, 16091201 cpu ops, 29775067 reads, 218387 writes, 24497 destroys
21.7968 creates/s, 14298.8 cpu ops/s, 26458.5 reads/s, 194.061 writes/s, 21.7683 destroys/s

worldofpadman1.bin: 1680x1050
6370 buffers, runtime 174534 ms (~2.9 minutes)
6370 creates, 17420486 cpu ops, 25720048 reads, 128989 writes, 6322 destroys
36.4972 creates/s, 99811.4 cpu ops/s, 147364 reads/s, 739.048 writes/s, 36.2222 destroys/s

wormsreloaded1.bin:
2308 buffers, runtime 258014 ms (~4.3 minutes)
2308 creates, 20300997 cpu ops, 24656646 reads, 143092 writes, 2266 destroys
8.94525 creates/s, 78681.8 cpu ops/s, 95563.2 reads/s, 554.59 writes/s, 8.78247 destroys/s

xonotic1.bin: git, Solarium map, 1680x1050, all max, no TC
8794 buffers, runtime 237665 ms (~4.0 minutes)
8794 creates, 12744266 cpu ops, 29794420 reads, 316599 writes, 15910 destroys
37.0017 creates/s, 53622.8 cpu ops/s, 125363 reads/s, 1332.12 writes/s, 66.943 destroys/s

xonotic2.bin: 1680x1050
6075 buffers, runtime 487698 ms (~8.1 minutes)
6075 creates, 17374997 cpu ops, 32298481 reads, 721045 writes, 5628 destroys
12.4565 creates/s, 35626.5 cpu ops/s, 66226.4 reads/s, 1478.47 writes/s, 11.5399 destroys/s

yetitmoves1.bin: 1280x800 high
16577 buffers, runtime 785419 ms (~13.1 minutes)
16577 creates, 4504140 cpu ops, 11575205 reads, 95205 writes, 13878 destroys
21.1059 creates/s, 5734.7 cpu ops/s, 14737.6 reads/s, 121.216 writes/s, 17.6695 destroys/s

youshould1.bin: 1366x768
5606 buffers, runtime 392804 ms (~6.5 minutes)
5606 creates, 751242 cpu ops, 1676496 reads, 137225 writes, 5081 destroys
14.2717 creates/s, 1912.51 cpu ops/s, 4268.02 reads/s, 349.347 writes/s, 12.9352 destroys/s