UNIVERSITY OF THESSALY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SOFTWARE FOR ANALYSIS AND VISUALIZATION OF MEMORY ACCESS PATTERNS

An Overview

PAVLOS AIMONIOTIS

Supervisor :
Nikolaos Bellas

Contents

List of Figures

Chapter 1

ABSTRACT

Detecting and optimizing memory access patterns, the patterns with which a system or a program reads or writes in memory, can lead to a major improvement in system performance. These patterns differ in the level of locality of reference, the tendency of a processor to access the same set of memory locations repetitively over a short period of time, and drastically affect cache performance.

Computer memory is usually described as ”random access”, but traversals by software will still exhibit patterns that can be exploited for efficiency. Memory access patterns can give significant information to understand the behaviour of any program, tool or function. Identifying access patterns and using this information to organize I/O operations and apply optimal data structures to make the code cache-friendly can provide a significant speedup to the execution of the program.

In this report a software for the analysis and visualization of memory access patterns is presented. The tool uses Valgrind toolset, specifically Gleipnir tool, for memory profiling and memory tracing. The output is being processed and data and information are being collected. These data and information are used to provide visualizations through python libraries, where we get to visualize the complete execution of the program, as well as two-dimensional and three-dimensional heap allocated arrays. Visualizations contains addresses, variable names, occurrences indexes, time heat map as well as cell positions, depending on the visualization the user chooses to do. All the data and information are also stored in files, to be further examined. A fully functional TCL shell [4] is responsible for the smooth operation of program.

Chapter 2

INTRODUCTION

2.1 Motivation

In recent years, a major system performance factor is memory. Heap allocations are extremely slow, complex and may also result in allocations scattered all over memory. As a result Dynamic RAMs are extremely slow. Also, the memory has a predefined size that does not allow unlimited number of programs to run concurrently. Due to "Memory Wall"[5], we need to find alternative ways to improve performance. "Memory Wall" refers to the growing speed gap between CPU and DRAM [6] [1], as technology have stayed far behind on making DRAMs faster, Figure 2.1 shows the gap between CPU and DRAM. One way, to improve performance, is by detecting memory access patterns, as said in [7], [8], [9].

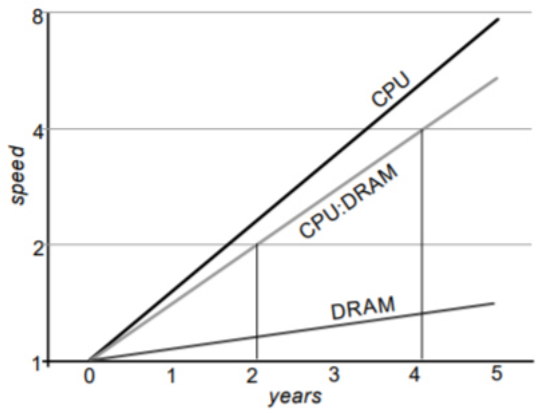


Figure 2.1: Figure 1 from [1]: The growing CPU-DRAM speed gap, expressed as relative speed over time (log scale). The vertical lines represent intervals at which the speed gap has doubled.

2.2 Objectives

Memory access patterns can show us how the memory is being allocated and accessed according to time and understand how the program touches memory. It is often said in High-Performance Computing that the cost of a cache miss is the largest performance penalty for an algorithm. The difference between an L1 cache-hit, and a full miss resulting in main-memory access, is about 50ns to 100ns. If algorithms randomly walk around, then we are less likely to benefit from the hardware support that hides this latency.

What we can do instead of randomly predicting, is to create data and information. Data and information will help to re-design algorithms and data structures. These information are the main goal of this research project. To improve memory performance we need new techniques, algorithms and tools.

2.3 Contribution

The contribution of this work is a tool that analyses and visualises memory access patterns. We will call this tool mapvisual (Memory Access Pattern Visualization) for the rest of the report. The implementation flow is described briefly on the following lines.

First, (1) we trace the file through Gleipnir, a valgrind plug-in, which I optimized specifically for mapvisual. Second, (2) we parse the gleipnir output file. Then,

(3) we go through a data filtering process. Lastly, (4) we perform algorithms for specific visualization techniques.

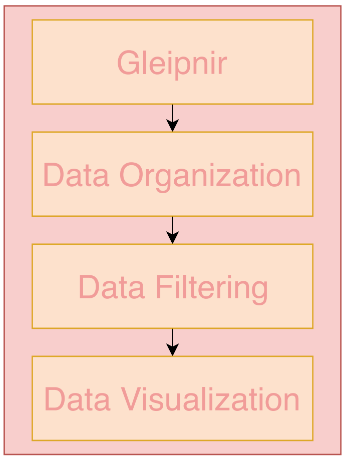


Figure 2.2: Implementation flow

We found many related work, but almost none of it was user-friendly, or even workable. Many of them was outdated, many was relating to intel pins of outdated versions and none was visualizing the way that mapvisual does.

The crucial contribution of mapvisual, is array visualization. Mapvisual does not only visualizes complete execution of memory access patterns. It also can visualize any given array of bytes, either two-dimension or three-dimension. This concept enters us on a new dimension of visualization, as we enter the three-dimensional world, and walking around the array as exactly we do visualize it in our brain.

2.4 Structure

Chapter 2 is about the background. Everything you need to know briefly, to understand the whole process of creating the tool and later on the documentation and on how it works.

Chapter 3 is about the tool, what was the process and how it is created.

Chapter 4 contains experimental results of benchmarks. Data, information and figures, are all been shown there.

Chapter 5 is the documentation of the tool, everything explained to the user, so that he will be able to install it and work with it.

Chapter 6 contais the experimental results. Info, data and visualizations.

Chapter 7 is the conclusion.

Chapter 8 has the references.

Chapter 3

BACKGROUND

3.1 Memory Access Pattern

One of the most important things in memory access patterns is locality of reference, also known as the principle of locality. It refers to the tendency of a processor to access the same set of memory locations repeatedly over a period of time. There are two basic types.

Temporal locality refers to the tend of processor to use the same data and access same memory location again and again. Think of a summary for example. If you keep adding to a summary variable for a million times, and you access summary every and each moment then you can store this variable to a faster memory storage, to avoid latency of subsequent references.

Spatial locality means that if a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In this case it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access for subsequent reference. Think of a loop that accesses an array sequentially for example. As you may have noticed, temporal locality is a special case of spatial locality.

Locality a type of predictable behaviour. Almost any system nowadays uses techniques of locality of reference, such as prefetching for memory, caching and branch predictors.

Plenty memory access patterns are known and appear very often on executions. Most of them have taken their name from a specific characteristic they have or from their "image", or better, of how those patterns look. Some of them are: sequential, linear, scatter and etc. I am pretty sure, until now you already understood how those patterns look like.

Sequential access pattern refers to reads and writes on increment or decremented straightforward addresses.

Linear access pattern refers to accesses that happen through a linear way. That means that there is a stable linear pattern that keeps going throughout accesses.

Scatter access pattern refers to accesses that happen sequential when reading but randomly when writing. The opposite, random reads but sequential writes, refers to gather.

3.2 Valgrind toolset

Valgrind is a dynamic binary instrumentation (DBI) framework that occupies a unique part of the DBI framework design space. [10] It is designed for building heavyweight Dynamic Binary Analysis (DBA) tools. It is used for memory debugging, memory leak detection and profiling. Valgrind sits between a layer of where the program runs and operating system. Before executing the instructions, the Valgrind core passes the program to a suite of tools chosen by the user when user runs it. It helps analyze any kind of software written on C, C++, Java, Fortran or assembly. Though, it is used mainly on C and C++, as most memory errors tend to happen on those two languages. Valgrind, became one of the most, if not the most, loved tool for software engineers. Many people contributed, and it now stands with a great toolset doing a variety of different operations.

The most important of them are the following:

Memcheck detects memory-management problems. When a program is running under Memcheck's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted.

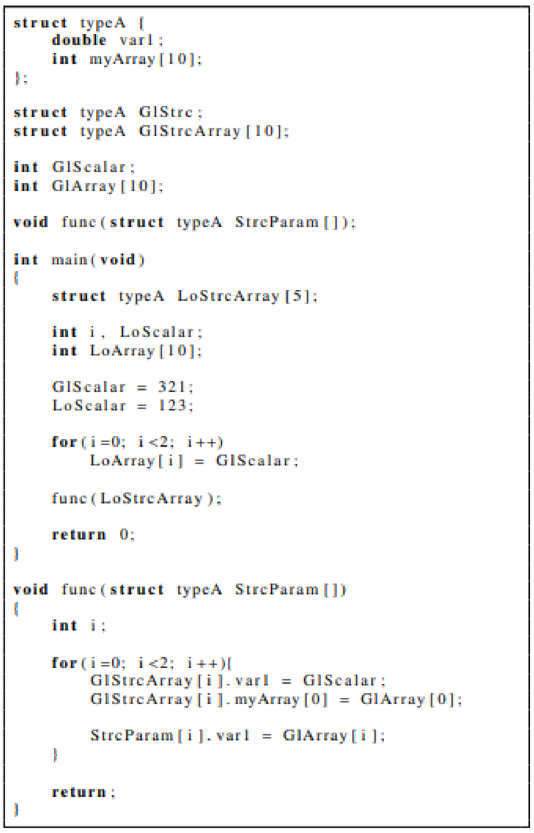
Cachegrind is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches in your CPU and so can accurately pinpoint the sources of cache misses in your code.

Massif is a heap profiler. It performs detailed heap profiling by taking regular snapshots of a program's heap.

DHAT is a tool for examining how programs use their heap allocations.

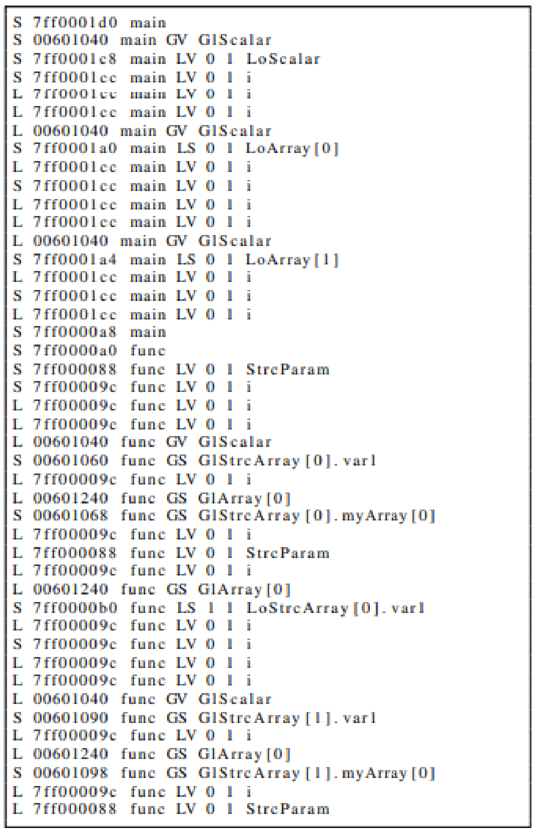
Those plugins are so important that come along with the source code of valgrind. There are many tools that are not in the official package of valgrind, but are still super useful. One of those is Gleipnir [11], which does memory profiling and tracing.

Gleipnir is a memory analysis tool that maps an application's source code variables to generated data traces. It is built as an intermediate between Valgrind's Lackey and Callgrind. The trace file generated by Lackey is purely a data and instruction trace with loads and stores corresponding to each data address. Gleipnir extends this information with the thread id, originating function, scope, data structure and the corresponding element, or a single scalar variable name. [2]



```c
struct typeA {
    double var1;
    int myArray[10];
};

struct typeA GlStrc;
struct typeA GlStrcArray[10];

int GlScalar;
int GlArray[10];

void func(struct typeA StrcParam[]);

int main(void)
{
    struct typeA LoStrcArray[5];

    int i, LoScalar;
    int LoArray[10];

    GlScalar = 321;
    LoScalar = 123;

    for(i=0; i<2; i++)
        LoArray[i] = GlScalar;

    func(LoStrcArray);

    return 0;
}

void func(struct typeA StrcParam[])
{
    int i;

    for(i=0; i<2; i++){
        GlStrcArray[i].var1 = GlScalar;
        GlStrcArray[i].myArray[0] = GlArray[0];

        StrcParam[i].var1 = GlArray[i];
    }

    return;
}
```

Listing 1: Example Source Code



```
S 7ff0001d0 main
S 00601040 main GV GlScalar
S 7ff0001c8 main LV 0 1 LoScalar
S 7ff0001cc main LV 0 1 i
L 7ff0001cc main LV 0 1 i
L 7ff0001cc main LV 0 1 i
L 00601040 main GV GlScalar
S 7ff0001a0 main LS 0 1 LoArray[0]
L 7ff0001cc main LV 0 1 i
S 7ff0001cc main LV 0 1 i
L 7ff0001cc main LV 0 1 i
L 7ff0001cc main LV 0 1 i
L 00601040 main GV GlScalar
S 7ff0001a4 main LS 0 1 LoArray[1]
L 7ff0001cc main LV 0 1 i
S 7ff0001cc main LV 0 1 i
L 7ff0001cc main LV 0 1 i
S 7ff0000a8 main
S 7ff0000a0 func
S 7ff000088 func LV 0 1 StrcParam
S 7ff00009c func LV 0 1 i
L 7ff00009c func LV 0 1 i
L 7ff00009c func LV 0 1 i
L 00601040 func GV GlScalar
S 00601060 func GS GlStrcArray[0].var1
L 7ff00009c func LV 0 1 i
L 00601240 func GS GlArray[0]
S 00601068 func GS GlStrcArray[0].myArray[0]
L 7ff00009c func LV 0 1 i
L 7ff000088 func LV 0 1 StrcParam
L 7ff00009c func LV 0 1 i
L 00601240 func GS GlArray[0]
S 7ff0000b0 func LS 1 1 LoStrcArray[0].var1
L 7ff00009c func LV 0 1 i
S 7ff00009c func LV 0 1 i
L 7ff00009c func LV 0 1 i
L 7ff00009c func LV 0 1 i
L 00601040 func GV GlScalar
S 00601090 func GS GlStrcArray[1].var1
L 7ff00009c func LV 0 1 i
L 00601240 func GS GlArray[0]
S 00601098 func GS GlStrcArray[1].myArray[0]
L 7ff00009c func LV 0 1 i
L 7ff000088 func LV 0 1 StrcParam
```

Listing 2: Gleipnir Generated Trace-file

Figure 3.1: Figure from [2]: Listing 1 and Listing 2 show an example of trace information collected by Gleipnir. Listing 1 shows a very simple program which contains a few data structures and a single function. Listing 2 shows a segment of the trace file generated by Gleipnir for program code in Listing 1. The format of the output generated by Gleipnir is straightforward.

Gleipnir traces and produces an output line for every and each access in memory. This tracing can lead to a significant run time overhead. We managed to optimize Gleipnir tool, specifically for our purposes, and it will be analysed later on. A description of a sample gleipnir output file is shown in 3.1.

Operation: L (load) / S (store) / M (modify) / I (instruction) / X (keyword) depending on the type of access. Keyword accesses are special lines inserted in the trace file by Gleipnir to describe certain events during tracing.

Address: Memory address (in the simulated address space of Valgrind).

Memory_size: Size of memory accessed.

Thread_Id: Id of the thread that caused the access.

Scope: G (global) / S (stack) / H (heap) depending on the location of the element.

Function: Name of the function call that caused the access.

Variable_Info: G (global) / L (local) and V (variable) / S(structure)

Chapter 4

TOOL ANALYSIS

In this chapter we are going to discuss how the tool came to life. We will start with memory tracing, on how Gleipnir works and the optimization which manages to reduce output files more than half. We will continue on how this information is adapted from mapvisual. How does mapvisual computes information, how does it reconstructs arrays and how all these data are going through python scripts which lead to visualizations. To create and code the whole tool, was a process of many complex and detailed things, lines were written and changed after months, there are things that are even still complicated for me. It is obvious that we can not go through all the process in this chapter, but the important part is going to be shown as detailed as possible

4.1 Memory tracing

4.1.1 Gleipnir optimization

Memory tracing is a crucial point for the performance of the tool. There many memory profiler and memory tracing tools, but nothing was suitable for the information that mapvisual wanted. We wanted a tool to analyse every access, have as output accessed addresses, identify if a load or store happened and if the variable belongs on stack or heap. It was important to know the starting address point for every allocation storage and the variable names, either stack or heap allocated.

Everything described above was found on Gleipnir tool, which under the

```
--read-var-info=yes
--read-debug=yes
```

flags, was exactly what we were looking for. Remember that Gleipnir is built as an intermediate between Valgrind's Lackey and Callgrind. If you have already worked with Lackey or Callgrind, you will already understand what does it mean.

On the other hand Gleipnir, had major problems. Firstly, Gleipnir was extremly slow and under the flags that we discussed above, –read-var-info=yes –read-debug=yes, it became incredibly slower on specific expirements. Speed is not the only problem we faced. Gleipnir also produces very large files. It seems that big applications can create output files equal to Terabytes, or maybe more. We will not know until we find such application. If we wanted to rely on Gleipnir valgrind plugin tool, we had to act and optimize it.

Gleipnir is a well-developed tool, and Valgrind a very complicated tool. Developing a tool through Valgrind API can get really complicated. Especially, when you come in as a third person on a tool, which is thousands and thousands of lines, and you try to optimize it, while it has stopped its development because it was supposed to have reached its limits. As you already understand, optimizing Gleipnir as a tool, was not the smartest choice, at the end it would be a great success and would have great research interest, but the chances were against me, and it was really possible to make a hole into the water.

At this point, there were three possible ways to choose. Either try to find another memory tracing tool or we continue with bottlenecks. The third way, was something that we would easily call as the best choice. If Gleipnir can not get optimized as a valgrind tool, at least we can optimize it for our own purposes.

As you will get to know how everything works in Documentation chapter later on, there is something you need to know at this point. In order for Gleipnir to memory trace files, user needs to include Gleipnir's library on the source code and add pins as start and stop point on the part of the code that wants to be analysed. It is something that every user needs to do, and can not become an automated process. That means that user will have interaction with the tool.

So now let two functions with same local variables lets name them *func1-¿var* and *func2-¿var*. If it was to run Gleipnir, you would know that var is different because the first one would refer to *func1* and the second one to *func2*. But as a user, if you wanted to analyse *func1*, why not to restrict in *func1* and get in your pins calls of *func2*? Sometimes, it is unavoidable and it is obvious, but as a tool, it should not build around corner cases.

You may wonder why all this information came at this point. We come back to the point that Gleipnir was about to be optimized for our own purposes. So this is what happened. Everything that gets analysed except stack or heap, has been removed. We do not need to know about global. The next step was to remove shared objects, it was many bytes per file, and not useful to the tool. It has successfully been removed. But what made the biggest impact is as you can understand now, the removal of function names and arguments. Sometimes it may be complicated for the user to understand why he has to go through this, so lets make this clear.

Slambench [3], an open-source project for SLAM (Simultaneous localization and mapping), when analysed with the original Gleipnir for function raycast() called on raycastKernel() for 3 frames, created access lines as follows.

```
L 1ffeffec48 8 1 S NONE::raycastKernel(__device_builtin__float3*,
    __device_builtin__float3*, __device_builtin__uint2, Volume,
    sMatrix4, float, float, float, float) LS view.data[3].z
```

Removing shared object name and function name with arguments from this line, makes it as simple as:

```
L 1ffeffec48 8 1 S LS view.data[3].z
```

On this experiment, the total lines that were accessed was 3.225.668. The total file size of Gleipnir's output was reduced from 557,6 Megabytes to 93,1 Megabytes. We managed to achieve a reduction of 83,2%! The reduction is not stable and it depends on many parameters, like function name size, variable names size, arguments and many other things that would make a line bigger or smaller. A general rule is that we are talking for reductions $\geq$ 25%, with bigger applications tending to be around 75%.

4.1.2 Data structure

Even when optimized, Gleipnir remains a big file to parse and store. So when parsed there is another post-process that takes place and redirects information into the correct structures. This post-process could be described as a compression process.

The data, extracted from the trace file, for every valid memory access will be stored in record instances. The unique record instances will be added to the Lookup table. For every access, a pair of operation type and Lookup table index will be added to the Access List. Doing like this, we manage to store all info on easily access-able struct. The detailed flowchart of the data filtering and organization along with the details on the used structure types are shown in 4.1.

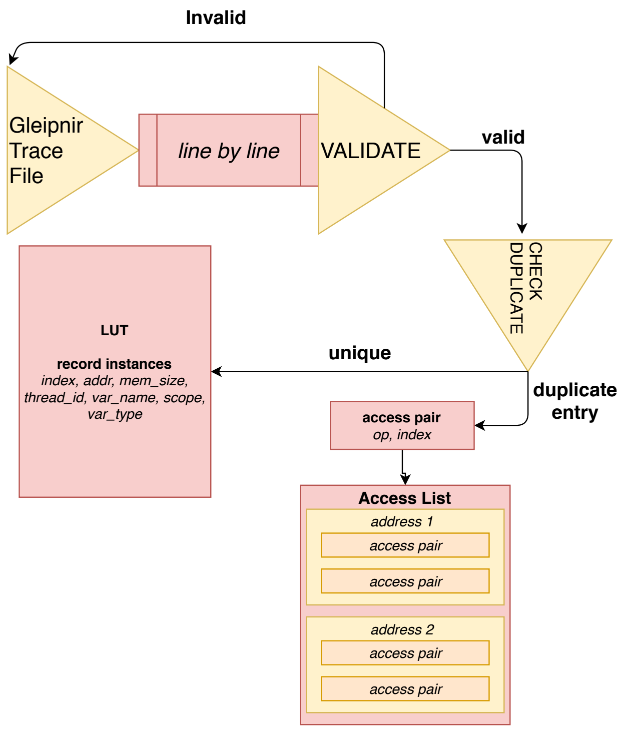


Figure 4.1: Data organization

The following is part of the start of every Gleipnir optimized file.

```
X START 0:32372 at 0
X THREAD_CREATE 0:1
X MMAP 004a29000 2105344
X MMAP 004027000 8192
X MMAP 004c2b000 2125824
```

```
X MMAP 004029000 159744
X MMAP 004e32000 3928064
X MMAP 004050000 12288
*X 1 MALLOC 0051f1030 40 tangledaccess_c_12 0
*X 1 MALLOC 0051f1080 40 tangledaccess_c_13 0
*X 1 MALLOC 0051f10d0 160 tangledaccess_c_14 0
*X 1 MALLOC 0051f1190 160 tangledaccess_c_15 0
*X 1 MALLOC 0051f1250 160 tangledaccess_c_16 0
S 1ffefff468 8 1 S
L 1ffefff468 8 1 S
S 1ffefff49c 4 1 S LV i
L 1ffefff49c 4 1 S LV i
L 1ffefff49c 4 1 S LV i
L 1ffefff490 8 1 S LV arr
L 1ffefff49c 4 1 S LV i
$S 0051f1030 4 1 H H-0 tangledaccess_c_12.0
L 1ffefff49c 4 1 S LV i
L 1ffefff488 8 1 S LV pin
L 1ffefff49c 4 1 S LV i
$S 0051f1080 4 1 H H-0 tangledaccess_c_13.0
```

As you can see information on asterisks (*) are really important and a separaty saved aswell. To make the whole tool work is a process of many complex and detailed things and its obvious that we can not go through all of them in here. But I think this is an important part and it deserves some lines.

As you can see for heap allocated variables, when accessing them ($) we get an address (0051f1080) and a file with a line (tangledaccess_c_13). On asterisks (*) we have the starting point of all heap allocated variables and their total bytes. Those two values can help us reconstruct every array of the program and identify memory access patterns. If you read back in 2.3, this is one of the most important contributions of this work. We will use special functions and paths to find file in our system, in this case tangledaccess.c, then open it and find line 12, use string separation algorithms, detach the variable from all the other code line, then convert 0051f1030 address on a cell of the array allocated in tangledaccess.c on line 12. The whole process will be explained later on, but at the moment we need to stick that we need to save the asterisks (*) on a file like this.

```
0051f1030 40
0051f1080 40
0051f10d0 160
```

```
0051f1190 160
0051f1250 160
```

Also we need to have in mind, while we are parsing line by line, we have to convert the file and line on the actually variable, it is faster than storing it and coming back later. In our case, for example, tangledacces.c the 13th line contains the following code line

```
pin = (int *) malloc ( N * sizeof(int) );
```

which means that our function should detect the line, separate the string and return *pin* as variable name. Also, in this moment I want to mention that you can pin's type is *int* and total bytes are 40, just from the above list. We can easily determine that this is an array of integers with N = 10.

4.2 Data filtering

Storing the data we discussed in 4.1.2 appropriately, makes the things easy to start collecting data. A good structure filled with information is always a good point to start.

The first goal is to create information about the total structure we stored. We know that we have a Look Up Table (LUT) with unique accesses and for every duplicate of those we stored it on an access list, as it was the same access, but probably with a different operation and occurrence index.

For every entry we start computing the memory usage, to have a great overall image of memory usage before visualizing. Hopefully, we have stored all kind of information about loads, stores and modifies, and we can start access by access, creating arrays and storing information depending on indexes and positions. No need to get in details of coding, as code is open-source and everyone can have access on it, and it will just get things complicated, as I did in 4.1.2, where a figure can solve the problem, instead of typing lines and lines of code.

If we manage to find the correct position, index and pair, then it will be no problem start calculating memory usage.

```
switch(pair_array[i].operation) {
    case 'S':
    mem_usage_s[pos]+=lut[pos]->m_size;
    break;
    case 'L':
```

```
    mem_usage_l[pos]+=lut[pos]->m_size;
    break;
    case 'M':
    mem_usage_l[pos]+=lut[pos]->m_size;
    mem_usage_s[pos]+=lut[pos]->m_size;
    break;
}
```

The first information we can produce until now is the following.

```
Index - Appearance Count - Address - Memory Usage Store - Memory
    Usage Load - Variable Name
```

Another trick we can do, is while going through our data which we stored as we were are parsing the tracing file, is to create the order of access. Parsing memory tracing file and storing data on our LUT and access list was sequential, which means every access we do in loop is sequential and stands for the real accessibility of the program.

```
order_of_access[i] = pair_array[i].index
```

and we just created an array with the order of accessed memory locations! So the following conclusion comes. A new file with the following can be created.

```
Order of Access - Address - Operation - Variable Name
```

Everything starts building up! This file may not be that clear all by it self, but for sure it will be our core on creating visualizations.

```
57 0051f108c S pin
69 0051f1040 L arr
63 0051f1268 S array
76 0051f1040 S arr
73 0051f1090 S pin
85 0051f1044 L arr
79 0051f1270 S array
92 0051f1044 S arr
89 0051f1094 S pin
101 0051f1048 L arr
95 0051f1278 S array
108 0051f1048 S arr
105 0051f1098 S pin
```

```
117 0051f104c L arr
111 0051f1280 S array
124 0051f104c S arr
121 0051f109c S pin
133 0051f1050 L arr
127 0051f1288 S array
140 0051f1050 S arr
137 0051f10a0 S pin
149 0051f1054 L arr
143 0051f1290 S array
159 0051f1298 S array
167 0051f1250 L array
173 0051f10d0 S dif
173 0051f10d0 L dif
179 0051f1190 S sky
172 0051f1250 S array
185 0051f1258 L array
```

As mentioned before, those addresses mean nothing at the moment, but having our allocated info saved from before, our starting point and total bytes of each pointer ,as shown on 4.1.2, we will come to the point of converting them.

4.3 Visualization

At this point, we have extraced all necessary data. If someone does not like visualizations or understands best by text, he could just stop here. Files produced are enough for to understand the pattern.

There are three types of visualizations supported. All plots are three-dimensions and created using plotly python library. Remember plotly is still a library, which creates an HTML file and you can see it on your browser, walk around it. If you create plots with millions of data, the HTML file can get big, and your browser may not be in a place to open it. It is up to you how you want to use it.

Complete memory access pattern is the main visualization. The axis contain information about variable names or array name, address and occurrence index*. The coloring is a heat map in time. The darker the earlier accessed, the brighter the later. Of course there is a mouse-over, which analyses every dot in the plot.

Two-dimension array visualization refers to 2d arrays. It reconstructs the array. The axis contain information about position and occurrence index*. Position refers to the real position of an element in the array, usually called (i, j). The coloring is a heat map in time. The darker the earlier accessed, the brighter the later. Of course there is a mouse-over, which analyses every dot in the plot.

Three-dimension array visualization refers to 2d arrays. It reconstructs the array. The axis contain information about position. Position refers to the real position of an element in the array, usually called (i, j, k). The coloring is a heat map in time. The darker the earlier accessed, the brighter the later. Of course there is a mouse-over, which analyses every dot in the plot.

**occurence index refers on time of accessed. A variable or an element, may have been accessed more than 1 time, while others may have been accessed just one. We could not neglect this information as it is extremely useful*

To visualize, we will strictly use the file we created of order data in 4.2 and allocated information in 4.1.2.

4.3.1 Complete memory access pattern

As said above, this is the main visualization and the most common. It is almost the first graph that comes in mind when talking about memory access patterns, and exactly what we discussed in 3.1.

If something is visited sequentially then you have to see sequential patterns, if it is strided, strided, if it is a scatter, pattern will be a scatter. It is pretty much, straightforward. Variable names and their patterns, addresses and occurence index and dots for every cell.

A python script runs through the data and does at this point nothing super special. The script is responsible to parse the files we created and store data on its own storage. Then, we sort a pack of access (refer to access time, i.e. if it is the 10th line in order file, we take that as the 10th element accessed), addresses and names, strictly by address. By doing this we have created an information like the following.

```
address1 3 arr1
address1 5 arr1
address1 7 arr1
address1 9 arr1
address1 11 arr1
address2 4 arr2
address2 6 arr2
address2 8 arr2
address2 10 arr2
address3 1 pos
```

```
address4 2 bool
```

**this is a sample example*

We can now easily count the accesses on specific addresses. We just created the occurrence index we discussed in the previous section!

We gathered all the required information to print about the complete memory access pattern.

```
Index - Address - Variable - Occurrence index
```

The plotly is responsible to take all these data, and in some lines of code, it creates plots.

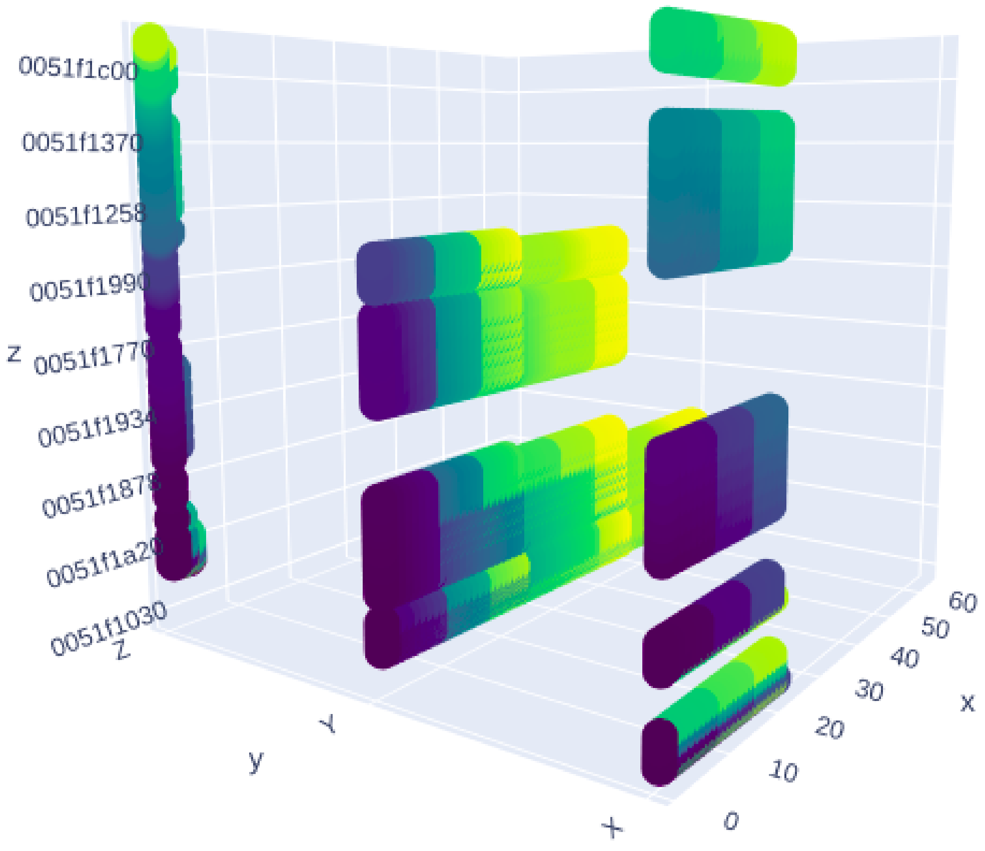


Figure 4.2: Complete MAP on blocked matrix multiplication

On 4.2 we get to see our first visualization. It is matrix multiplication based on blocks. This is a heap allocated analysis. On *Y axis* are variable names, on *Z axis* addresses and on *X axis* occurrence index. As you can see *X* array is multiplied by *Y* array and the result is stored in *Z* array.

In this example the crucial part of the code is small, so it would be better to show it here. Remember, this is a blocked matrix multiplication, not a row matrix multiplication, not a column matrix multiplication. May be complicated on the first eye, because blocked matrix multiplication is complicated.

```
#elif BLOCKED
  for (int ii = 0; ii < N; ii+=B)
    for (int jj = 0; jj < N; jj+=B)
      for (int i = ii; i < ii+B; i++) {
       for (int j = jj; j < jj+B; j++) {
          tmp = 0;
          for (int k = 0; k < N ; k++)
              tmp += *(X+i*N+k) * *(Y+k*N+j);
          *(Z+i*N+j) = tmp;
      }
     }
#endif
```

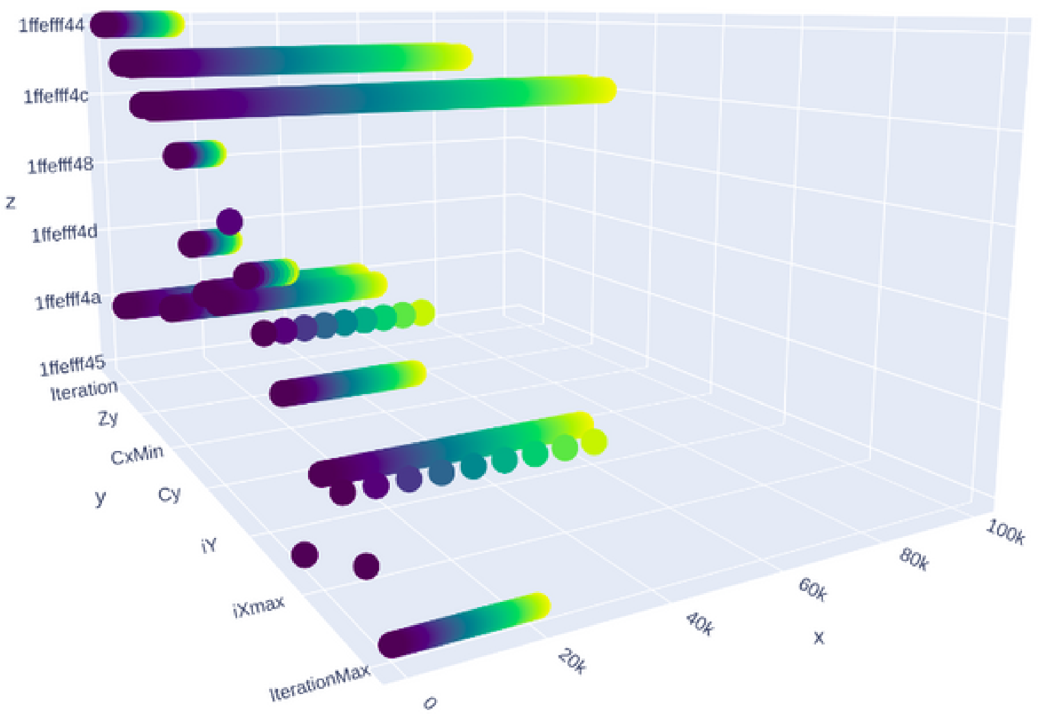


Figure 4.3: Complete MAP on part of mandelbrot set

On 4.3 we get to see another visualization of mapvisual. This visualization is part of mandelbrot set, stack allocated analysis. On *Y axis* are variable names, on *Z axis* addresses and on *X axis* occurrence index. The perspective of the picture is not the best one, but as it is on 3d world it will always lack of perspective. In the tool, this is an HTML file, it opens through browser and you can walk around it, zoom or unzoom it and get into cells. But as a picture it is malfunctioning. Anyway, lets get back into to it. As you can see most variables are used from the very begging, darker colors. The "bigger" lines, mean more accessed variables.

4.3.2 Two-dimension array memory access pattern

The goal of two-dimension array memory access pattern visualization is to reconstruct the array on three-dimensional world. At this point the user should be able to provide some information. Obviously, the name of the array that he was to visualize, but also the total columns and type of bytes. We will get into this on Documentation section. For know, lets stick with the python script.

Plotly is a visualization, plot library. Obviously the starting point is at bottom left (0, 0). There are two choices here. Either we move our starting point to top left but without seeing the (0, 0) in axis, or we keep bottom left (0, 0) and we assume that this is the starting point of our array, and array is shown vertically reversed. We stayed with that, so from now on, the bottom left is the start of the array. Bottom line is the first row, top line is the last row.

Firstly, the script needs to parser the order data file and allocated info (we discussed about them on 4.2 and 4.1.2). For order data file, we check if the name of the array is the same as the name that user gave as input array. If show we store it. The way to check if the array that user inserted is simple. Allocated information file contains the first address of the array. Order data file contains all the addresses of the array name the user gave as input. If the address that appears in allocated info appears as well in the data we just parsed, the user putted a valid name of array. Otherwise, the user gave an invalid array name.

To reconstruct the array we need to convert every address into rows and columns, or better for us programmers on (i, j). While we are sorting them according to address, we are going through a process. We subtract the current address on hexadecimal from the starting address.

```
difference = int(current_address, 16) - int(start_address, 16)
```

We now have the exact difference, or better the position of our current array according to start point. Remember that the user has inserted as argument the type of bytes, assume that for an int we have 4 bytes, for a double 8 bytes and so on.

If we divide the difference with the type of bytes we take the position of element.

```
element = difference / type_of_byte
```

Getting the position of the element and knowing how many columns we have requires two lines of code to convert it to (i, j).

```
i = element // int(cols)
j = element % int(cols)
```

We just converted an address of an array into the correct position of (i, j)! Remember we did this, while sorting the addresses, so we now have the following.

```
Index - Address - Variable - Occurrence index - Row - Col
```

On figure 4.2 you show the complete memory access pattern of a blocked matrix multiplication. As you can see you have three arrays, but even though you show the pattern, it would be better if you would get into they Y array for example and how this is happening legit on the parallelogram that an array creates!

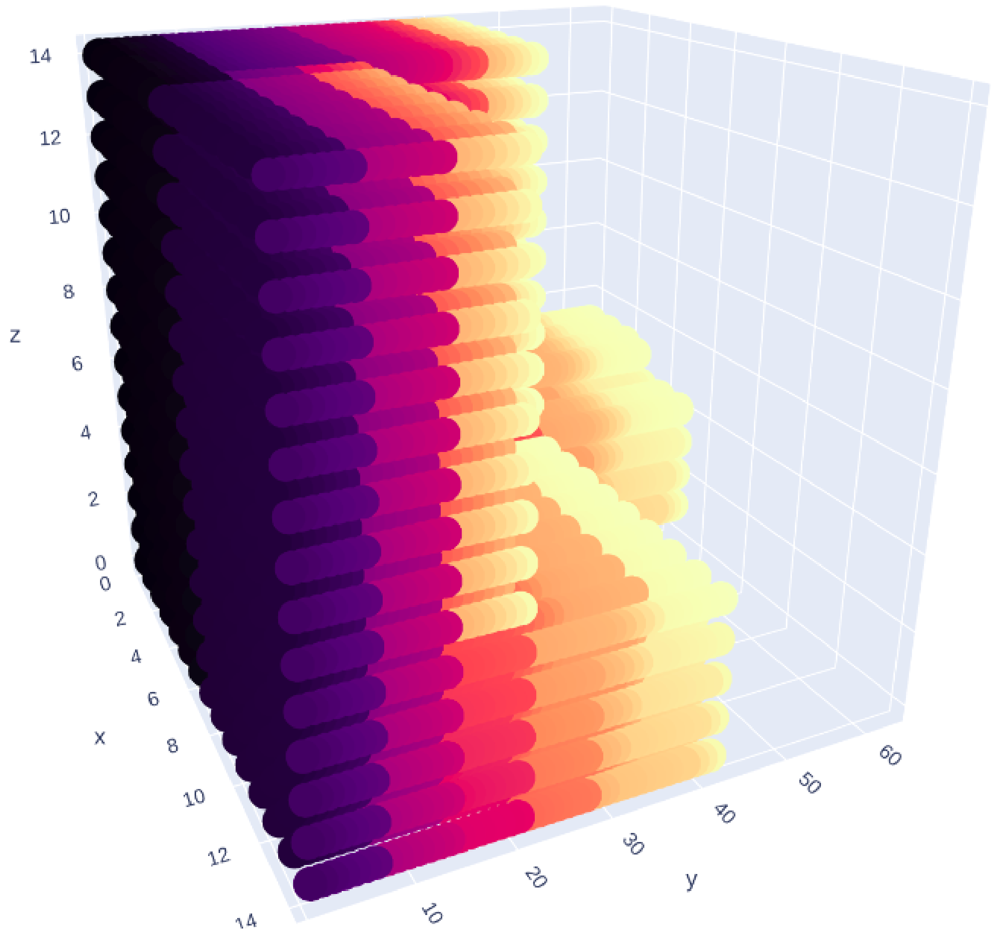


Figure 4.4: Two-dimension Y array memory access pattern of blocked matrix multiplication

On 4.4 is exactly that! As you can see we are now into the Y array and see every cell how it has been accessed, both in time but also in space. Rows are placed on Z axis (obviously they would be placed vertically) and columns on X axis. As described before, axis Y shows the occurrence indexes, or how many times each cell has been visited. The heat map stands for time.

4.3.3 Three-dimension array memory access pattern

The goal of three-dimension array memory access pattern visualization is to reconstruct the array on three-dimensional world. At this point the user should be able to provide some information. Obviously, the name of the array that he was to visualize, but also the total columns, the depth of the array and type of bytes. We will get into this on Documentation section. For know, lets stick with the python script.

Plotly is a visualization, plot library. Obviously the starting point is at bottom left (0, 0). There are two choices here. Either we move our starting point to top left but without seeing the (0, 0) in axis, or we keep bottom left (0, 0) and we assume that this is the starting point of our array, and array is shown vertically reversed. We stayed with that, so from now on, the bottom left is the start of the array. Bottom line is the first row, top line is the last row.

The process is similar to the two-dimension array visualization showed in 4.3.2. Firstly, the script needs to parser the order data file and allocated info (we discussed about them on 4.2 and 4.1.2). For order data file, we check if the name of the array is the same as the name that user gave as input array. If show we store it. The way to check if the array that user inserted is simple. Allocated information file contains the first address of the array. Order data file contains all the addresses of the array name the user gave as input. If the address that appears in allocated info appears as well in the data we just parsed, the user putted a valid name of array. Otherwise, the user gave an invalid array name.

To reconstruct the array we need to convert every address into rows, columns and depth, or better for us programmers on (i, j, k). While we are sorting them according to address, we are going through a process. Remember that a three-dimension array can be described as an array of arrays. So we what we are doing is almost what we did 4.3.2 but after the first time, we get some info to do another analysis. Every time we divide or module, we get a step closer. It will be clear shown in mathematics.

We subtract the current address on hexadecimal from the starting address.

```
difference = int(current_address, 16) - int(start_address, 16)
```

We now have the exact difference, or better the position of our current array according to start point. Remember that the user has inserted as argument the type of bytes, assume that for an int we have 4 bytes, for a double 8 bytes and so on.

```
element = difference / type_of_byte
```

We now need to get first into detecting the rows. Rows do not care about columns or depth. They know that they exist every *column* ∗ *depth* times.

```
i = element // (column * depth)
```

Next step is get back to the "other" array, remember that we said that we now have an array of an array.

```
ans = element - (i * total)
```

And now that we stick into a single row we can determine the two-dimension array which consists of (j ,k), just as before in 4.3.2 when we took (i, j).

```
j = ans // int(depth)
k = ans % int(depth)
```

We just converted an address of an array into the correct position of (i, j, k)! Remember we did this, while sorting the addresses, so we now have the following.

```
Index - Address - Variable - Occurrence index - Row - Col - Depth
```

On the plot, Z axis represents rows, X axis represents columns and Y axis represents depth.

On 4.5 we see a complete memory access pattern of a single three-dimension array which is accessed sequentially, on the following order, depth, columns, rows. This is a simple example to understand the usefulness of this visualization. It is just a a sequential accessed array, so for us to understand, but imagine if it had any difficult or complicated pattern.

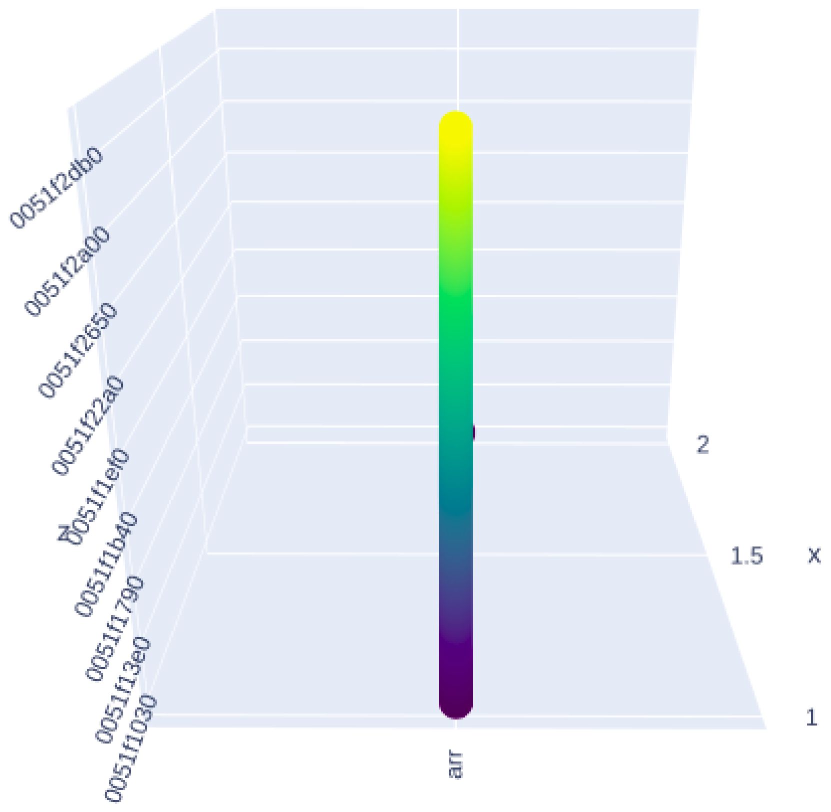


Figure 4.5: Complete memory access pattern of a 3D array

On the other side, 4.6, shows the same array reconstructed. You can see the it is obviously gets brighter when going through depth at a slow pace, as it always visits depth first (Y axis), then it is almost starts getting a better pace of brightness on columns (X axis), as it is more rare to run through columns than depth, and finally the brightness goes on a faster pace on rows (Z axis), as it is the outer loop and the most rare. A look on the code will make it clear.

```
for (i=0; i < ROWS; i++){
    for (j=0; j < COLS; j++){
      for (k=0; k < DEPTH; k++){
        *(arr+i*(COLS*DEPTH)+j*DEPTH+k) = i;
      }
    }
  }
```

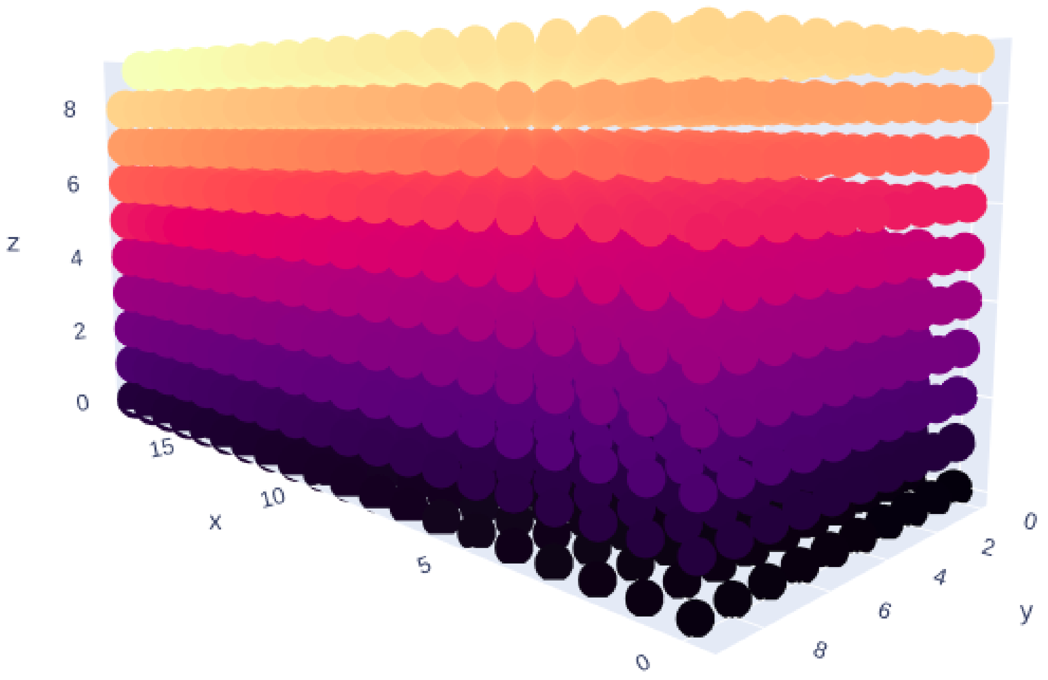


Figure 4.6: Three-dimension arr visualization of 4.5

4.4 Shell

Above all, a TCL shell runs. The shell is responsible for the smooth functionality. It is supported with history and tab completion to make the experience easier for the user. TCL shell commands have also been added for valgrind plugin tools to run on your executable, may be useful sometimes.

There are not many things we can see in this chapter about the shell that runs the tool. Nothing special happens there, almost everything that it is done on those hundreds of code, is just to get ready the shell to call the functions.

The shell contains plenty of commands, in order the flow to be clear and easy to understand, and also to be able to put files from previous executions and go straightforward on the exact commands. There is not a point to get into explanation as everything will be explainted on Documentation chapter about the TCL Shell, as it is a way for the tool to talk with the user.

Chapter 5

DOCUMENTATION

In the following pages a Documentation of the tool will be presented. I hope it is clear and fully detailed.

5.1 Installation

Source code

You can find the source code on `https://github.com/pavlosaim/mapvisual`

You can find the memory tracing gleipnir optimized source code on `https://github.com/pavlosaim/memory-tracing`

Libraries

mapvisual runs a plug in library, you will probably need to install them if you do not already have them in your system. This library is used to visualize the data. It is called plotly and more information can be found on https://plot.ly

```
pip install plotly==4.3.0
```

Valgrind with Gleipnir

Install a new valgrind to have access. Clone it from the original repository

```
svn co svn://svn.valgrind.org/valgrind/trunk valgrind
```

We have to install Gleipnir valgrind tool manually. Clone the optimized for mapvisual version, as discussed in here, into the valgrind folder. Then rename the memory-tracing folder to gleipnir.

```
cd valgrind;

git https://github.com/pavlosaim/memory-tracing.git;

mv memory-tracing/ gleipnir/
```

Before making Valgrind with Gleipnir, on gleipnir folder.

Apply debug patch and makefile patches.

```
./glconf.sh -d
```

This will patch debuginfo.c and pub_tool_debuginfo.h for gleipnir's use, and update Valgrind's makefiles and conf files.

Malloc replace

replace - will NOT patch Valgrind, but configure Gleipnir to use gl_malloc_replacement.c (i.e. Allocation will call VG_(cli_malloc)()

```
./glconf -m replace
```

Applying the patch will not break other tools. (fingers crossed)

IMPORTANT: If you want to switch between the configurations (wrap vs replace), you have to distclean and start from top.

Back on valgrind main folder. Edit Makefile.am and add "gleipnir" on TOOLS.

Proceed with valgrind installation process.

```
autogen.sh

./configure --prefix = current_valgrind_dir

sudo make; sudo make install;
```

Gleipnir should now be installed with valgrind!

mapvisual

Clone mapvisual from github repository, anywhere in your system.

```
git clone https://github.com/pavlosaim/mapvisual.git
```

Compile it and you mapvis executable should appear.

```
make all;
./mapvis
```

If everything went find, the tool should be able to run and on your screen the following should be shown.

```
Memory Access Pattern Visualization is currently running!
Type <help> for running flow and command explanation.
```

5.2 Tcl shell

Once you you execute the program, you will enter the TCL shell. TCL shell is responsible for everything from now on. There are plenty of commands, and will all be shown in the next lines.

load_exe

```
load_exe <path>
```

load_exe is the tcl command responsible to set the path to your executable you want to run and analyse. It requires that you already compiled your code with Gleipnir libraries and pin inside.

set_src_path

```
set_src_path <path>
```

set_src_path is the tcl command responsible to set the path to your source code, the one you compiled to get the executable on load_exe. It is absolutely necessary to be executed if you are doing heap analysis. Remember on 4.1.2 we needed to convert a file with a line to a variable name. This path is used to detect the file!

set_valgrind_path

```
set_valgrind_path <path>
```

set_valgrind_path is the tcl command responsible to set the path to your valgrind, the one you installed gleipnir. Probably a one time done on script line.

set_arguments

```
set_arguments <args>
```

set_arguments is the tcl command responsible to set to the executable that you loaded through load_exe. There is no way to check by the tool if you had to put arguments or not. This is job of the source file. So be careful that if you forget them and source file has no checks, you may struggle with unexpected errors.

set_mem_alloc

```
set_mem_alloc <HoS>
HoS = Heap or Stack
'H' for Heap, 'S' for stack
```

set_mem_alloc is the tcl command responsible to set the flag for heap or stack analysis. Two possible arguments, 'H' for heap or 'S' for stack. Defaut has been set to 'S', but be careful.

run_gleipnir

```
run_gleipnir
```

If you set everything correct, run_gleinir should do you the memory tracing. No arguments required.

memory_trace_analysis

```
memory_trace_analysis
```

memory_trace_analysis analysis gleipnir out, so run_gleipnir should run obviously first. No arguments needed.

filtering_process

```
filtering_process
```

filtering_process analysis memory_trace_analysis output, so memory_trace_analysis should run obviously first. No arguments needed.

memory_stats

```
memory_stats
```

memory_stats shows memory usage info, it should be executed after filtering_process.

complete_memory_access_pattern

```
complete_memory_access_pattern <save(optional)>
```

complete_memory_access_pattern visualizes exactly the complete memory access pattern as described in 4.3.1. Use save argument if you want to save the visualization data and information. The output will be stored in figures/ for the vis HTML and visinfo/ if you choose to save data.

2d_memory_access_pattern

```
2d_memory_access_pattern <arrayname> <columns> <typeofbytes> <save(
    optional)>
```

2d_memory_access_pattern visualizes exactly the complete memory access pattern as described in 4.3.2. Use save argument if you want to save the visualization data and information. The output will be stored in figures/ for the vis HTML and visinfo/ if you choose to save data.

3d_memory_access_pattern

```
2d_memory_access_pattern <arrayname> <columns> <depth> <typeofbytes>
    <save(optional)>
```

3d_memory_access_pattern visualizes exactly the complete memory access pattern as described in 4.3.3. Use save argument if you want to save the visualization data and information. The output will be stored in figures/ for the vis HTML and visinfo/ if you choose to save data.

clear_analysis

```
clear_analysis
```

clears all files, information, figures made of the analysis.

valgrind tools

```
lackey
helgrind
cachegrind
massif
dhat
```

Type any of valgrind tool once you loaded your executable with load_exe and it will run through the specific valgrind plug in tool. The output will be stored in grindanalysis/

5.3 Connecting source file to Gleipnir

The first on connecting your source code file to gleipnir is to include the correct header file.

```
#include "/home/path/to/installed/valgrind/gleipnir/gleipnir.h"
```

Next step is to set pins GL_START; and GL_STOP; into the part of the code you want to analyse. Avoid setting it in a big part for big applications. Try to be specific for what you want and do not consider more information.

For example, assume function1 which does allocation techniques. It can be called on function2, function3 and function4. If you care to analyse the calls through function2, avoid putting pins in the start and the end of function1. Prefer to put pins before and after the call of function1 in function2. Doing like this you will avoid getting unnecessary information from function3 and function4, and of course extreme overhead.

For more information of how to run Gleipnir visit `https://csrl.cse.unt.edu/node/30` (use GL_START; and GL_STOP; instead of GL_START_ISNTR; and GL_STOP_INSTR;)

5.4 Flow

Typically if you need to do different visualizations and analyses you should write scripts to do so. It is a faster and easier way to edit. Prefer it even for one analysis. A typical script looks like this.

```
load_exe /home/to/my/executable
set_src_path /home/to/source/file/dir/
set_mem_alloc H
set_valgrind_path /home/path/to/valgrind/bin/valgrind
run_gleipnir
memory_trace_analysis
filtering_process
complete_memory_access_pattern save
2d_memory_access_pattern arrayname 10 4 save
```

Of course, to run a script after running the tool is to source it.

```
source script
```

You should always set the following first. Order does not matter.

```
load_exe /home/to/my/executable
set_src_path /home/to/source/file/dir/
set_mem_alloc H
set_valgrind_path /home/path/to/valgrind/bin/valgrind
```

Next to analyse you should execute the following commands, with the exact order!

```
run_gleipnir
memory_trace_analysis
filtering_process
```

Then the main part of analysis have been complete. You can try visualize anything any time. You can walk around files and folders to see all the information and data, and save them if you would like to.

Once you type in commands for visualization, your default browser will open to show the visualizations.

5.5 Big application bottlenecks

Gleipnir is a slow tool and files even if optimized are not big. Big applications* may hurt you and your feelings. It may take hours of running. Big files mean and big overhead and much time for tool to parse and analyse. For very big applications you need patience. Also plotly, as a library, has its own limits. It cannot visualize many millions or billion of data.

A good way to overcome all this, is to run gleipnir for a certain amount of time, interrupt it, and then load it as it is. You will get a part of the memory access pattern, but remember it is called pattern because it normally follows a specific pattern.

**big applications refer to millions and millions and millions of accesses*

Chapter 6

EXPERIMENTAL RESULTS

Go back to 4.3.1, 4.3.2 and 4.3.3 for data file explanation.

6.1 Matrix Multiplication

The following examples are not the usual matrix multiplication. They are Columned and Blocked, so do not get too complicated with the results.

Columned

```
#elif MATRIX_COL
      for (int j = 0; j < N; j++) {
       for (int i = 0; i < N; i++) {
          tmp = 0;
          for (int k = 0; k < N ; k++)
              tmp += *(X+i*N+k) * *(Y+k*N+j);
          *(Z+i*N+j) = tmp;
        }
       }
#endif
```

Part of data used for complete memory access pattern

```
8491 0051f13ac  X 16

8492 0051f184c  Y 14

8493 0051f13b0  X 17
```

```
8494 0051f148c  Y 15

8495 0051f13b4  X 16

8496 0051f14cc  Y 15
```

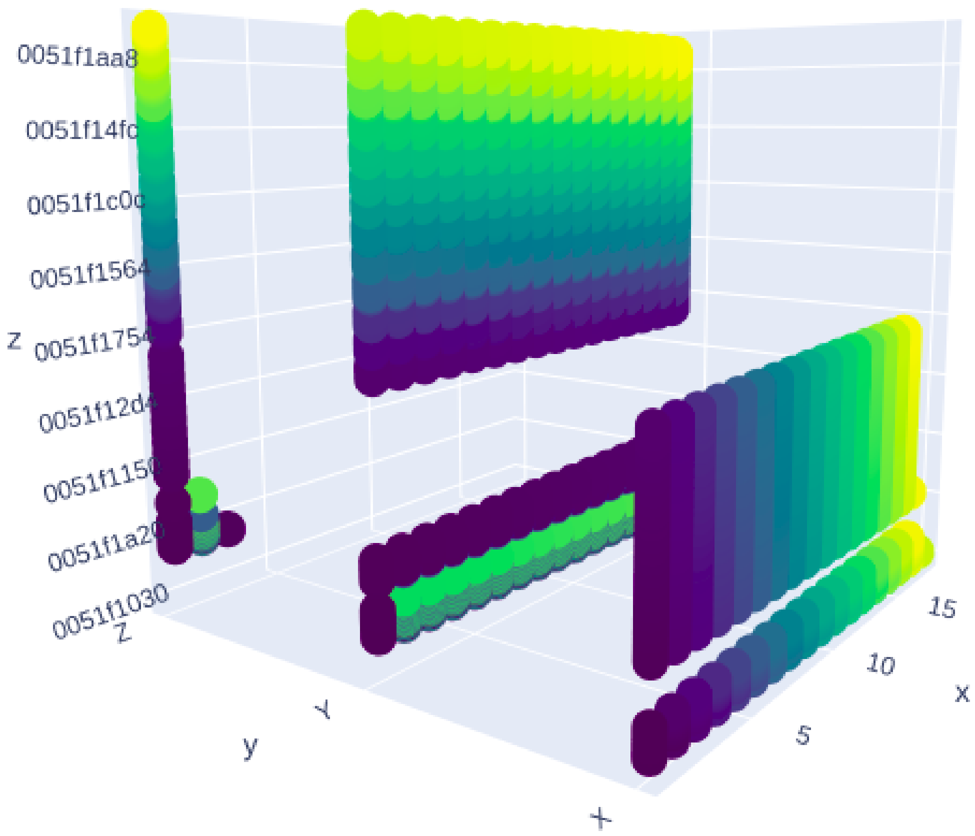


Figure 6.1: Columned Matrix Multiplication complete memory access pattern

Y array visualization part of data

```
4114 0051f178c  Y 15 12.0 15.0

4115 0051f17cc  Y 15 13.0 15.0
```

```
4116 0051f180c  Y 15 14.0 15.0
```

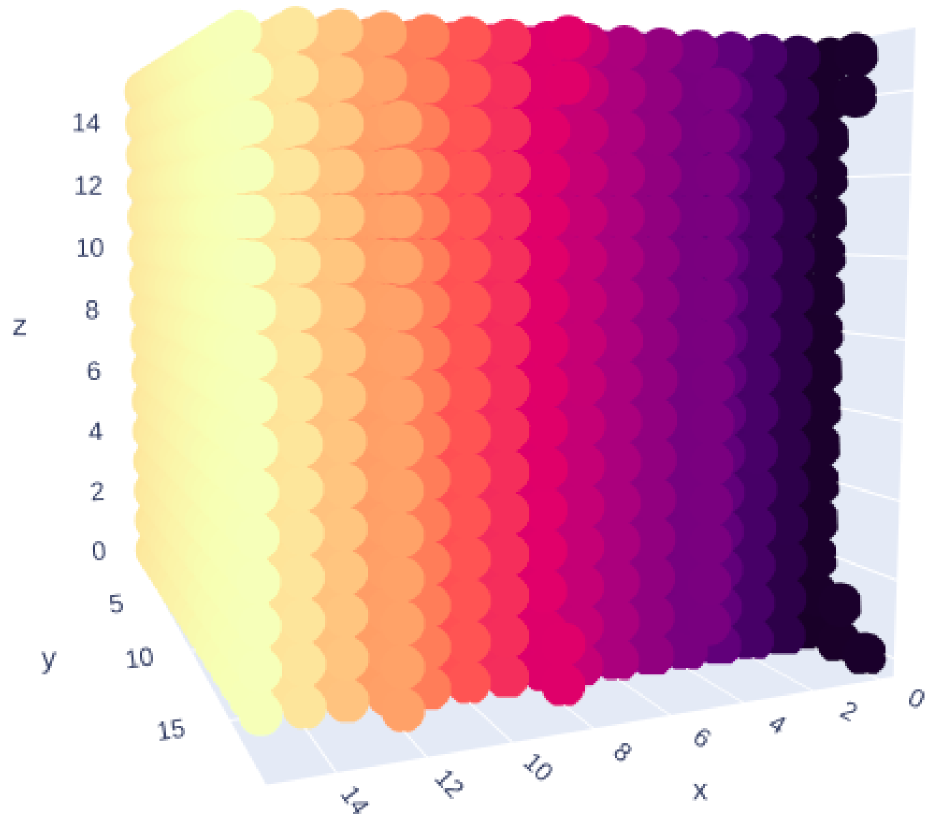


Figure 6.2: Two-dimension Y array memory access pattern of columned matrix multiplication

X array visualization part of data

```
4125 0051f140c  X 16 15.0 7.0

4126 0051f1410  X 17 15.0 8.0

4127 0051f1414  X 16 15.0 9.0
```

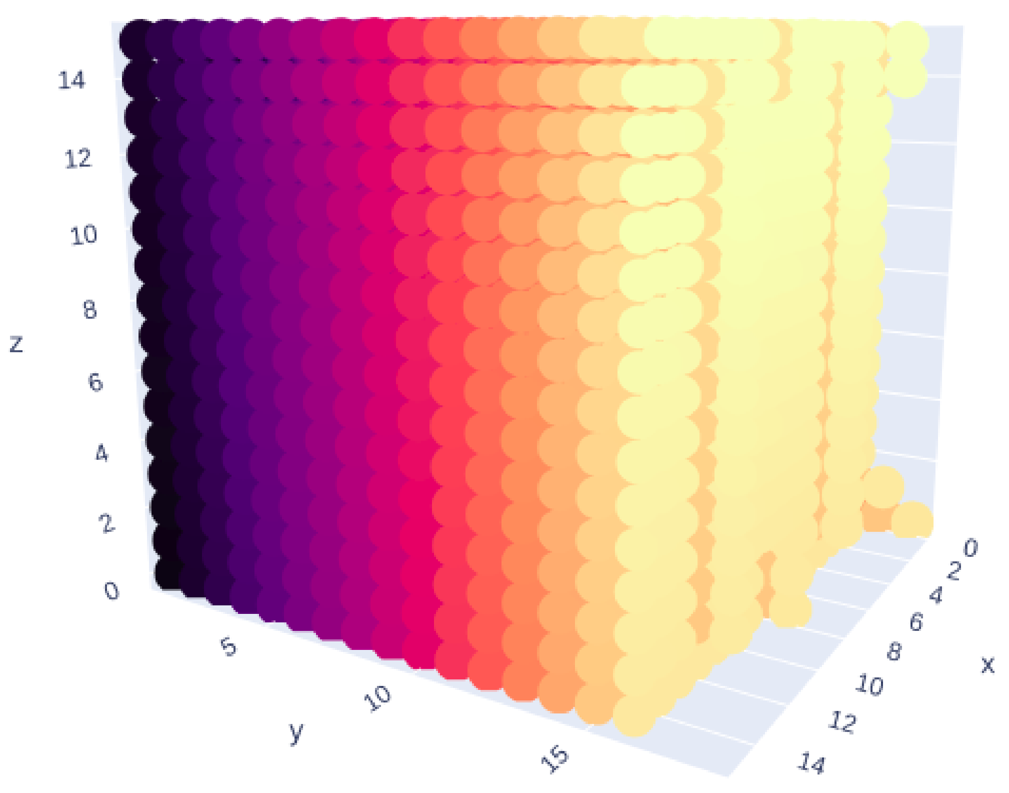


Figure 6.3: Two-dimension X array memory access pattern of columned matrix multiplication

Z array visualization part of data

```
251 0051f19e4  Z 1 5.0 13.0

252 0051f1a24  Z 1 6.0 13.0

253 0051f1a64  Z 1 7.0 13.0
```

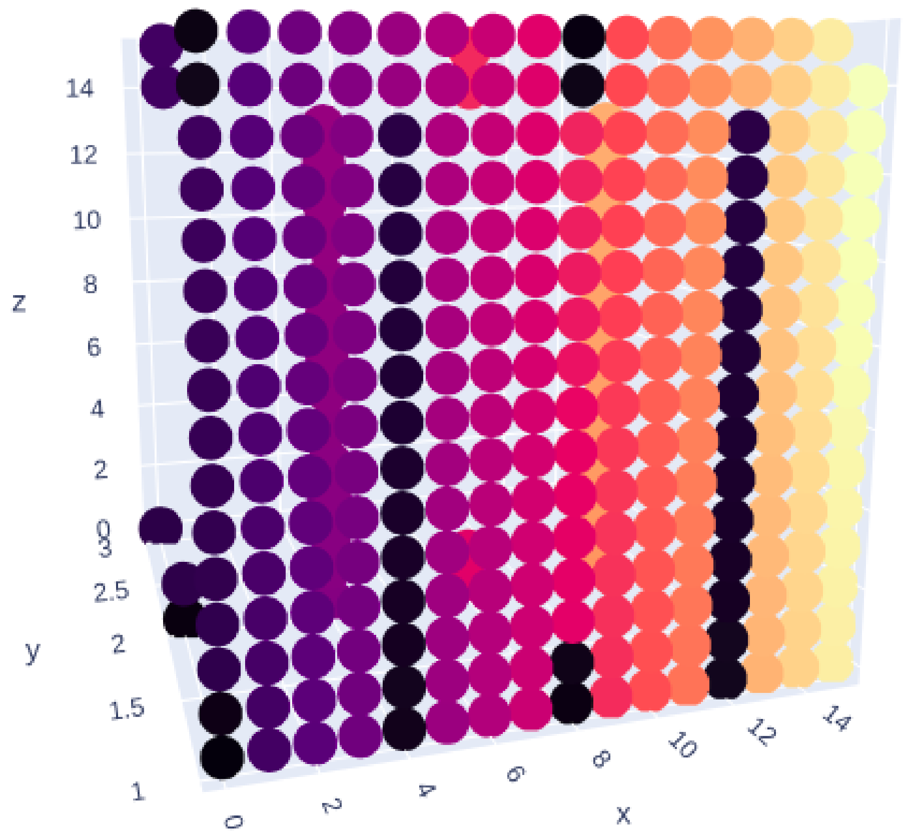


Figure 6.4: Two-dimension Z array memory access pattern of columns matrix multiplication

6.1.1 Blocked

```
#elif BLOCKED
  for (int ii = 0; ii < N; ii+=B)
    for (int jj = 0; jj < N; jj+=B)
      for (int i = ii; i < ii+B; i++) {
       for (int j = jj; j < jj+B; j++) {
          tmp = 0;
          for (int k = 0; k < N ; k++)
              tmp += *(X+i*N+k) * *(Y+k*N+j);
          *(Z+i*N+j) = tmp;
      }
    }
#endif
```

Part of data used for complete memory access pattern

```
13178 0051f142c  X 19

13179 0051f1818  Y 37

13180 0051f13f0  X 21

13181 0051f149c  Y 51

13182 0051f13f4  X 20
```

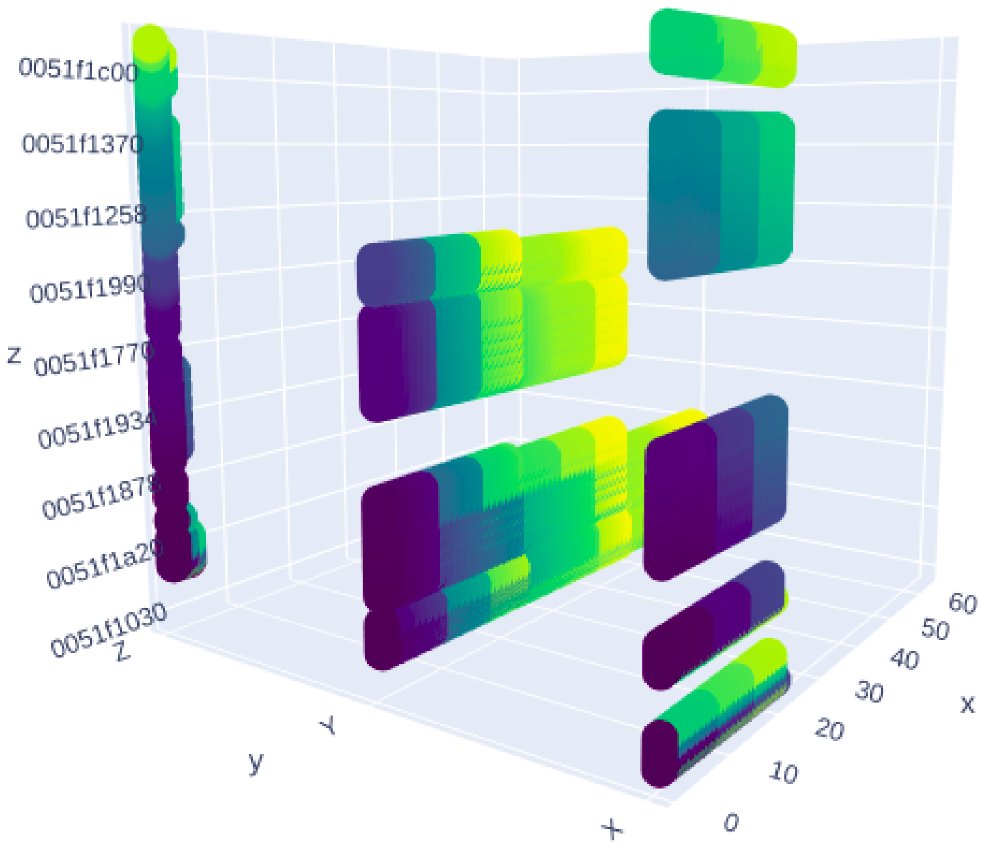


Figure 6.5: Blocked Matrix Multiplication complete memory access pattern

X array visualization part of data

```
5401 0051f13fc  X 21 15.0 3.0

5402 0051f1400  X 21 15.0 4.0

5403 0051f1404  X 21 15.0 5.0
```

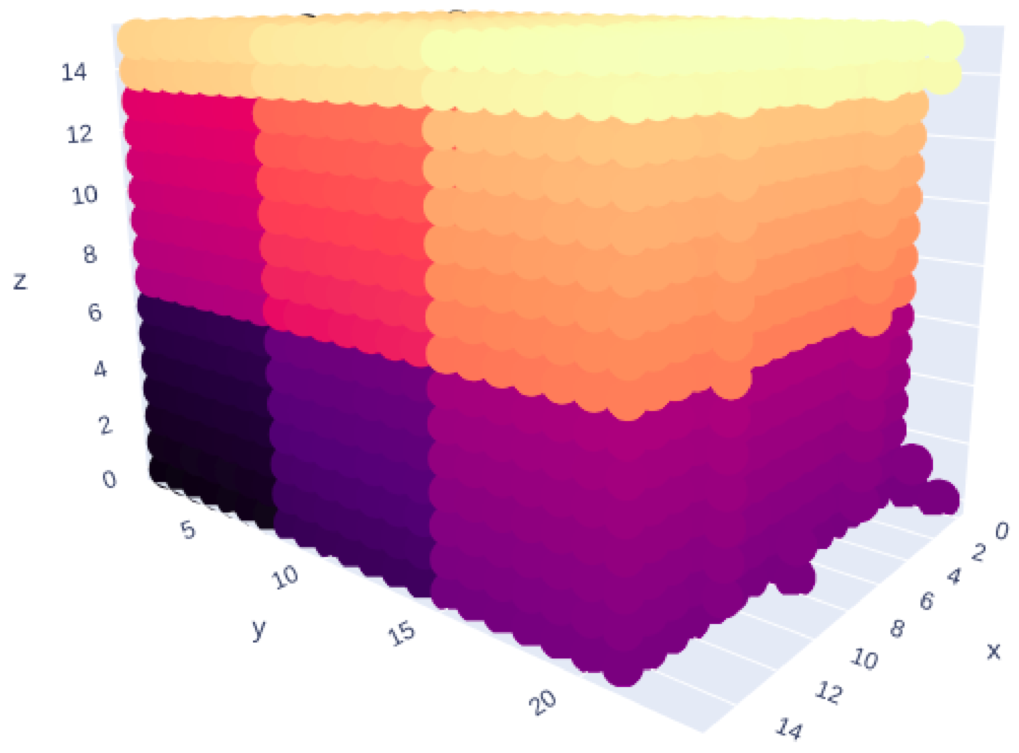


Figure 6.6: Two-dimension X array memory access pattern of blocked matrix multiplication

Y array visualization part of data

```
8492 0051f155c  Y 63 4.0 3.0

8493 0051f1760  Y 43 12.0 4.0

8494 0051f1560  Y 64 4.0 4.0
```

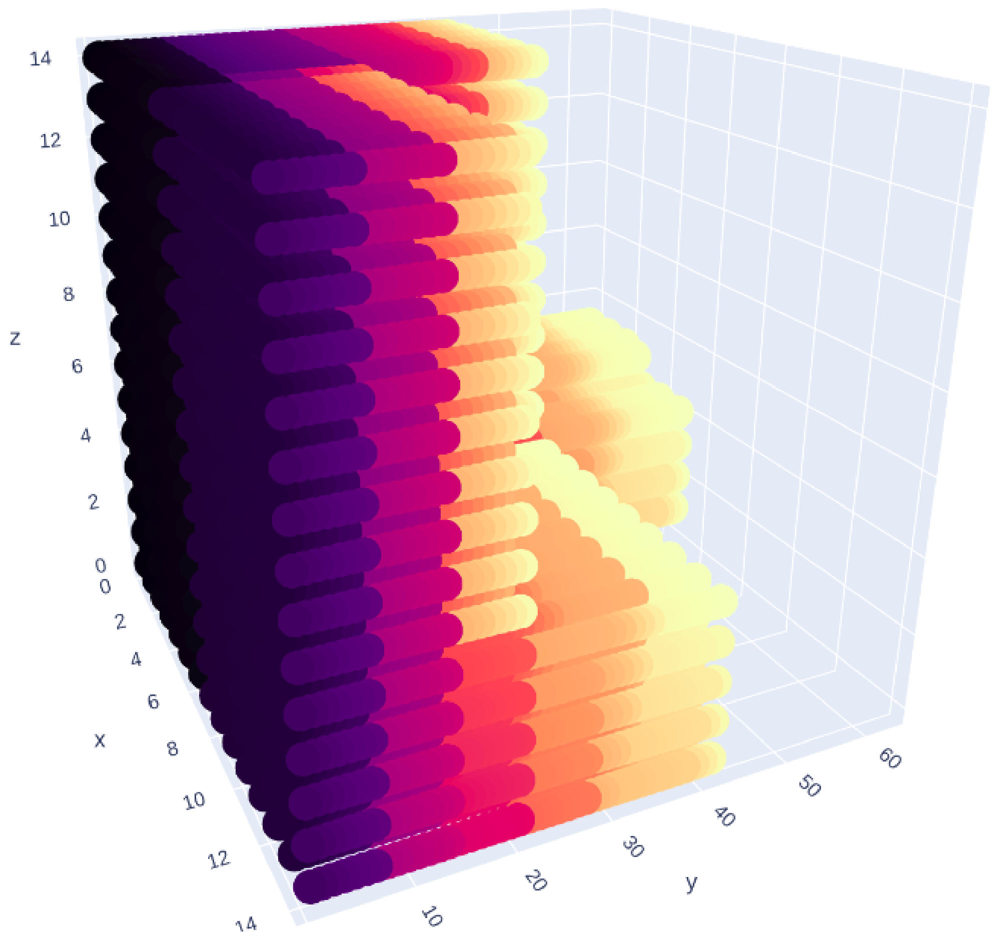


Figure 6.7: Two-dimension Y array memory access pattern of blocked matrix multiplication

Z array visualization part of data

```
362 0051f1c30  Z 3 15.0 0.0

363 0051f1c34  Z 2 15.0 1.0

364 0051f1c38  Z 2 15.0 2.0
```

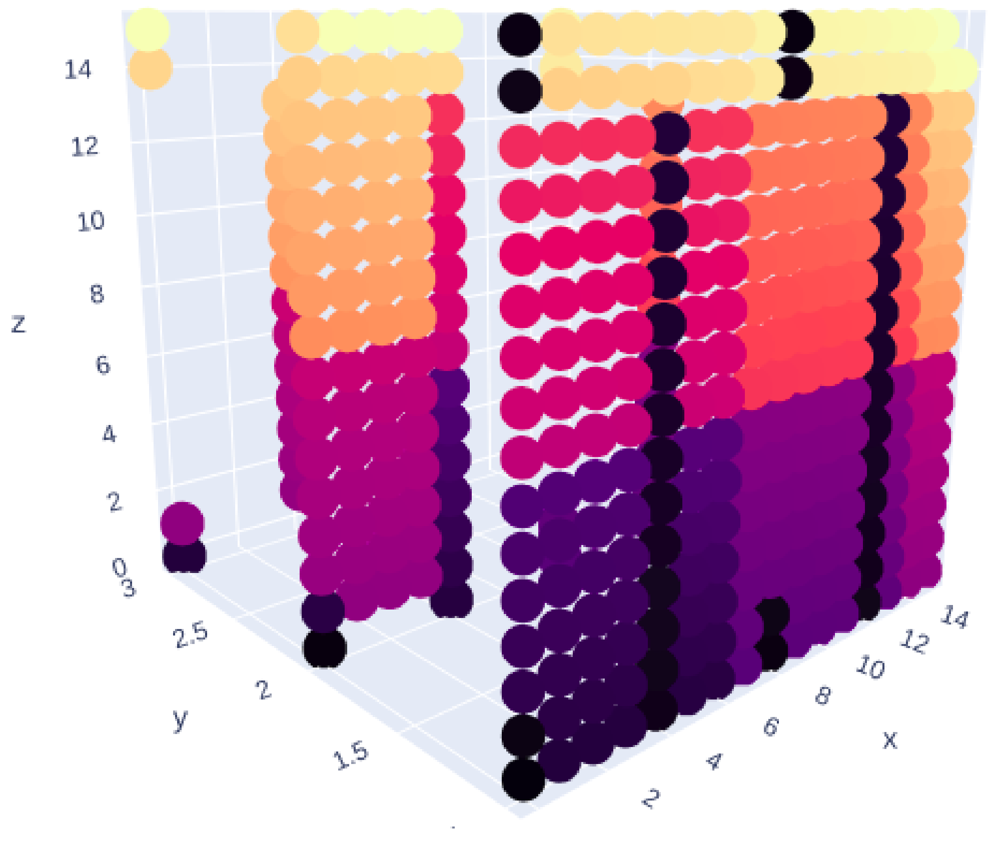


Figure 6.8: Two-dimension Z array memory access pattern of blocked matrix multiplication

6.2 Slambench

SLAMbench [3] is an open-source project for SLAM (simultaneous localisation and mapping). The following information was captured for some frames in function raycast() called in raycastKernel().

Millions of data were produced and plotly could not plot all of them, so a part of it is shown below.

```
if (tnear < tfar) {
        // first walk with largesteps until we found a hit
        float t = tnear;
        float stepsize = largestep;
        float f_t = volume.interp(origin + direction * t);
        float f_tt = 0;
        if (f_t > 0) { // ups, if we were already in it, then don't
            render anything here
            for (; t < tfar; t += stepsize) {
                f_tt = volume.interp(origin + direction * t);
                if (f_tt < 0) // got it, jump out of inner loop
                    break;
                if (f_tt < 0.8f) // coming closer, reduce stepsize
                    stepsize = step;
                f_t = f_tt;
            }
            if (f_tt < 0) { // got it, calculate accurate intersection
                t = t + stepsize * f_tt / (f_t - f_tt);
                return make_float4(origin + direction * t, t);
            }
        }
    }
```

Part of complete memory access pattern data.

```
375073 1ffeffeaa8  nearPlane 26791

375074 1ffeffeaac  farPlane 26791

375075 1ffeffec18  view.data 26792

375076 1ffeffec10  view.data 26792

375077 1ffeffec08  view.data 26792

375078 1ffeffec00  view.data 26792

375079 1ffeffebf8  view.data 26792

375080 1ffeffebf0  view.data 26792

375081 1ffeffebe8  view.data 26792

375082 1ffeffebe0  view.data 26792
```

```
375083 1ffeffebd8  integration.data 26792

375084 1ffeffebd0  integration.dim.y 26792

375085 1ffeffebc8  integration.size.z 26792

375086 1ffeffebc0  integration.size.x 26792
```

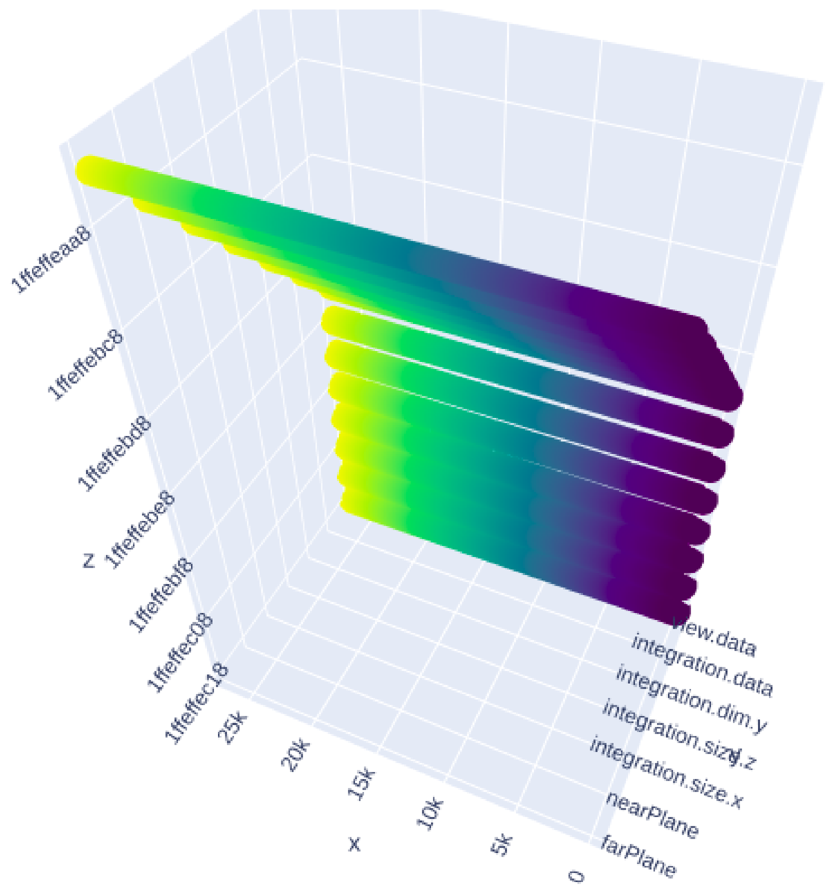


Figure 6.9: SLAMbench [3]

6.3 Mandelbrot

```
for(iY=0;iY<iYmax;iY++)
       {
            Cy=CyMin + iY*PixelHeight;
            if (fabs(Cy)< PixelHeight/2) Cy=0.0; /* Main antenna */
            for(iX=0;iX<iXmax;iX++)
            {
                      Cx=CxMin + iX*PixelWidth;
                      /* initial value of orbit = critical point Z= 0
                          */
                      Zx=0.0;
                      Zy=0.0;
                      Zx2=Zx*Zx;
                      Zy2=Zy*Zy;
                      /* */
                      for (Iteration=0;Iteration<IterationMax && ((
                          Zx2+Zy2)<ER2);Iteration++)
                      {
                          Zy=2*Zx*Zy + Cy;
                          Zx=Zx2-Zy2 +Cx;
                          Zx2=Zx*Zx;
                          Zy2=Zy*Zy;
                      };
                      /* compute pixel color (24 bit = 3 bytes) */
                      if (Iteration==IterationMax)
                      { /* interior of Mandelbrot set = black */
                         color[0]=0;
                         color[1]=0;
                         color[2]=0;
                      }
                   else
                      { /* exterior of Mandelbrot set = white */
                           color[0]=255; /* Red*/
                           color[1]=255; /* Green */
                           color[2]=255;/* Blue */
                      };
                      /*write color to the file*/
                      //fwrite(color,1,3,fp);
               }
       }
```

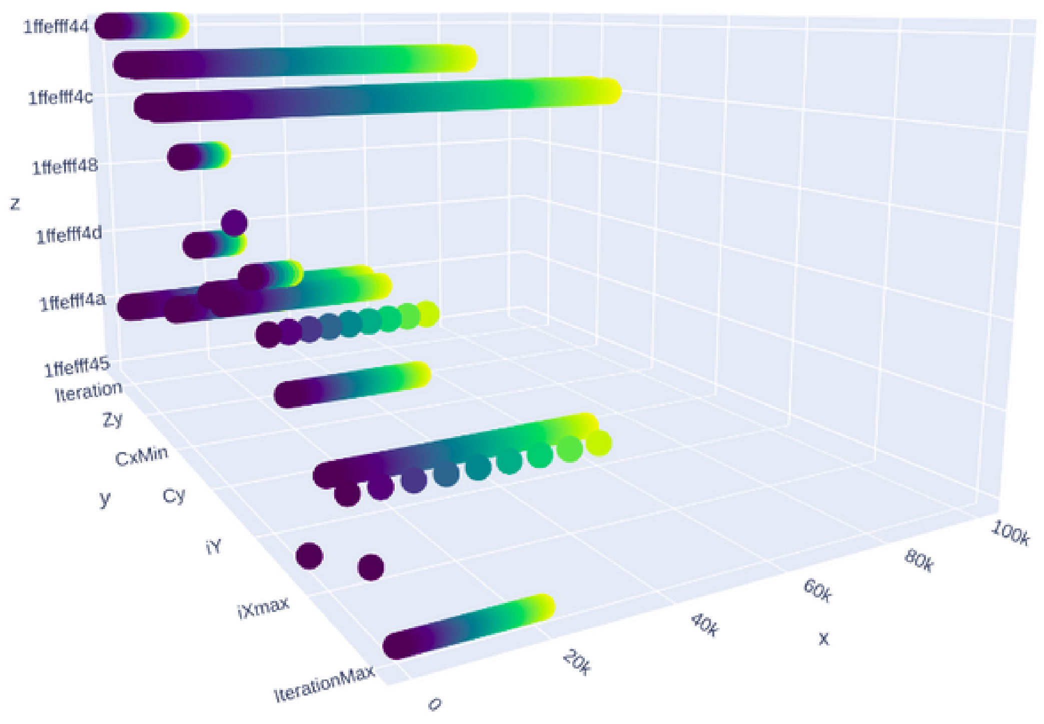


Figure 6.10: Part of Mandelbrot Set complete memory access pattern

6.4 3d array

```
for (i=0; i < ROWS; i++){
    for (j=0; j < COLS; j++){
      for (k=0; k < DEPTH; k++){
```

```
      *(arr+i*(COLS*DEPTH)+j*DEPTH+k) = i;
    }
  }
}
```

Part of data used for complete memory access pattern

```
1714 0051f2af0  arr 1

1715 0051f2af4  arr 1

1716 0051f2af8  arr 1

1717 0051f2afc  arr 1

1718 0051f2b00  arr 1
```

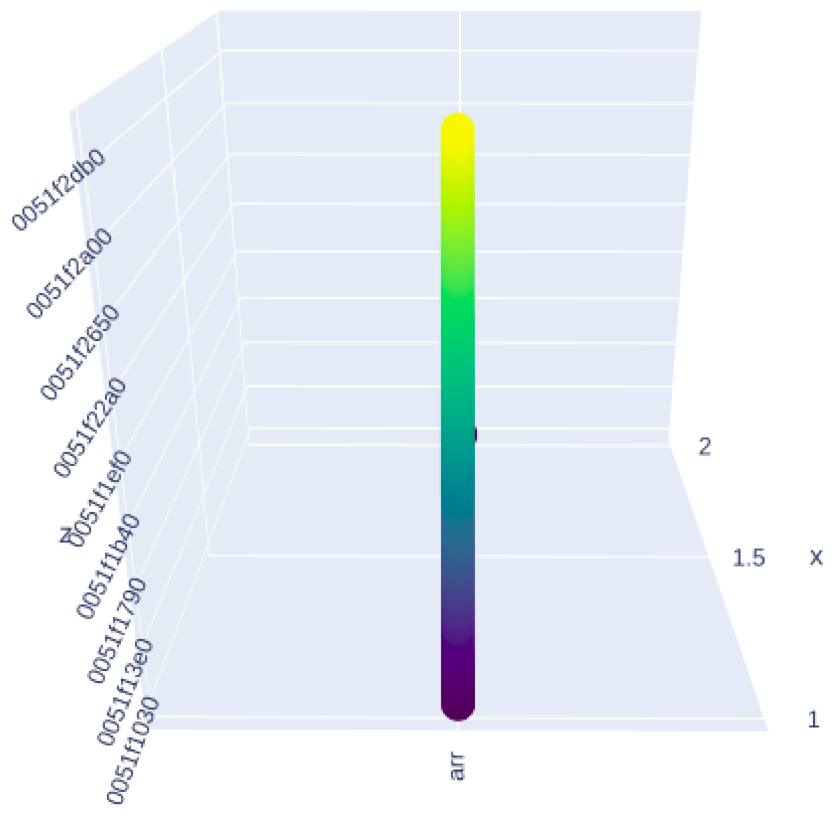


Figure 6.11: Complete memory access pattern of 3d array

Part of data used for three-dimension array reconstruct

```
1993 0051f2f4c  arr 9.0 19.0 1.0

1994 0051f2f50  arr 9.0 19.0 2.0

1995 0051f2f54  arr 9.0 19.0 3.0
```

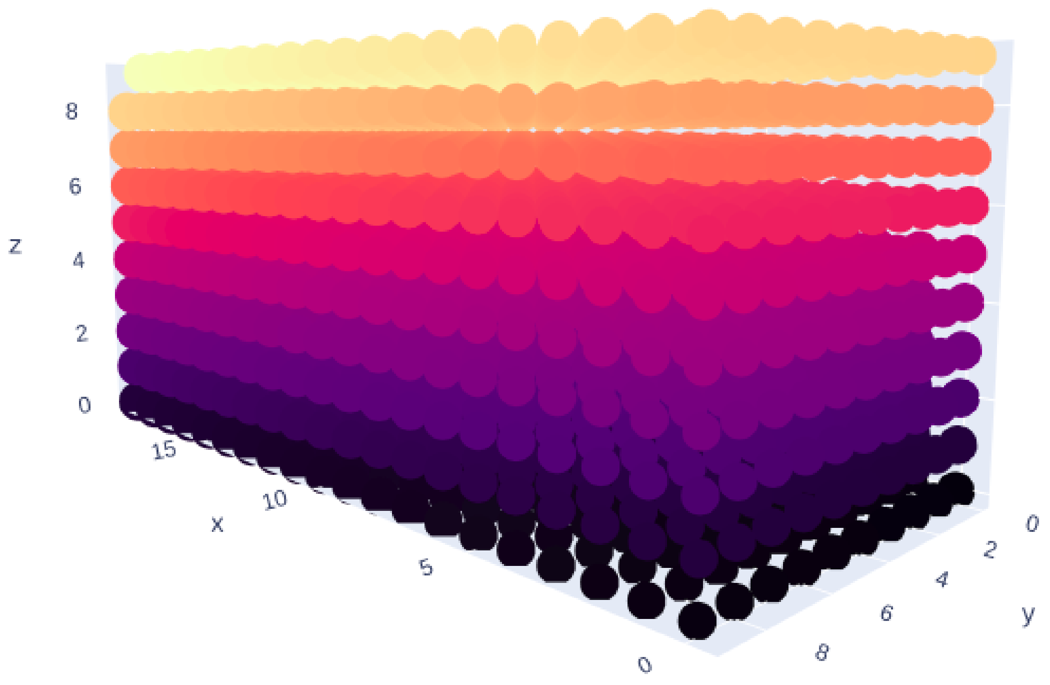


Figure 6.12: Three-dimension array memory access pattern of 3d array

6.5 Expiremental result conclusion

In expiremental results, I tried to show important information that will cover all features. It was obvious because of files, images and code that this section would get big, so if i added more examples and photos, it would probably be more than all the other text.

Chapter 7

CONCLUSION

To sum up, I have first managed to optimize Gleipnir for mapvisual purposes.

Then, a tool has been written in C, to parse and analyse all data from the memory trace. Those data are used to visualize and analyse any application. Visualization happens through python scripts using plotly library.

The tool runs through a TCL shell which makes it easy to understand, has a nice flow and gives you the opportunity to jump some commands if you already done some work previously.

Very big applications create overhead, but with the current TCL shell there is a way to overcome it as said in 5.5.

Chapter 8

REFERENCES

Bibliography

[1] Philip Machanick. Approaches to addressing the memory wall. *School of IT and Electrical Engineering, University of Queensland*, 2002.

[2] Tomislav Janjusic, Krishna Kavi, and Brandon Potter. International conference on computational science, iccs 2011 gleipnir: A memory analysis tool. *Procedia Computer Science*, 4:2058–2067, 2011.

[3] Luigi Nardi, Bruno Bodin, M Zeeshan Zia, John Mawer, Andy Nisbet, Paul HJ Kelly, Andrew J Davison, Mikel Luján, Michael FP O'Boyle, Graham Riley, et al. Introducing slambench, a performance and accuracy benchmarking methodology for slam. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5783–5790. IEEE, 2015.

[4] John K Ousterhout et al. *Tcl: An embeddable command language.* Citeseer, 1989.

[5] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[6] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE micro*, 17(2):34–44, 1997.

[7] Peter Grun, Nikil Dutt, Nikil Dutt, Nikil Dutt, and Alex Nicolau. Apex: access pattern based memory architecture exploration. In *Proceedings of the 14th international symposium on Systems synthesis*, pages 25–32. ACM, 2001.

[8] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[9] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2010.

[10] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[11] Tomislav Janjusic and Krishna Kavi. Gleipnir: A memory profiling and tracing tool. *ACM SIGARCH Computer Architecture News*, 41(4):8–12, 2013.