Blue Waters Petascale Semester Curriculum v1.0

Unit 9: Optimization

Lesson 3: Cache Memory Efficiency

Instructor Guide

Developed by Paul F. Hemler for the Shodor Education Foundation, Inc.

Browse and search the full curriculum at http://shodor.org/petascale/materials/semester-curriculum

We welcome your improvements! You can submit your proposed changes to this material and the rest of the curriculum in our GitHub repository at https://github.com/shodor-education/petascale-semester-curriculum

We want to hear from you! Please let us know your experiences using this material by sending email to petascale@shodor.org

Program Efficiency Enhancement by Effective Caching

Paul F. Hemler, Hampden-Sydney College

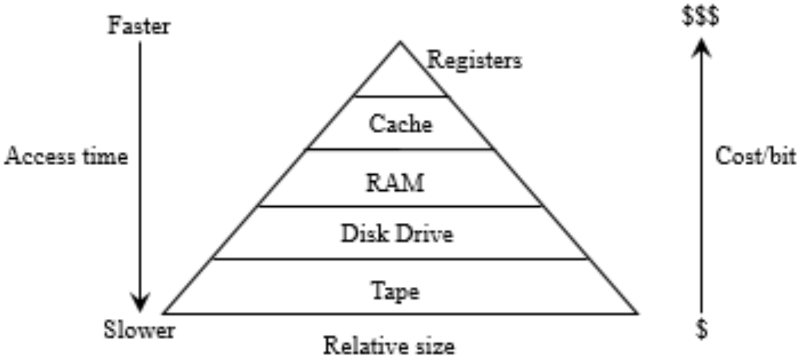
Abstract

In this module the student will experiment with programs and Bash shell scripts to empirically show that program execution time is dependent on accessing the elements in one-dimensional arrays in a cache friendly manner. The module explains what at cache memory is, where it is located in the hardware hierarchy, and the principles that make it feasible. Using this understanding, the module demonstrates how to and how to not utilize the cache efficiently by first allocating a linear array and then timing different access patterns. A Bash shell script is then described to execute a program with different arguments but without user intervention. The results are then plotted to clearly show when the cache is used and when it is not used.

Upon completion of this module the student will understand the technology and basic operation of a cache memory, how to allocate a large linear array and efficiently access all the elements within the array. The student will also learn how to write a shell script to automate the execution of a program with different arguments, time important sections and collect all output.

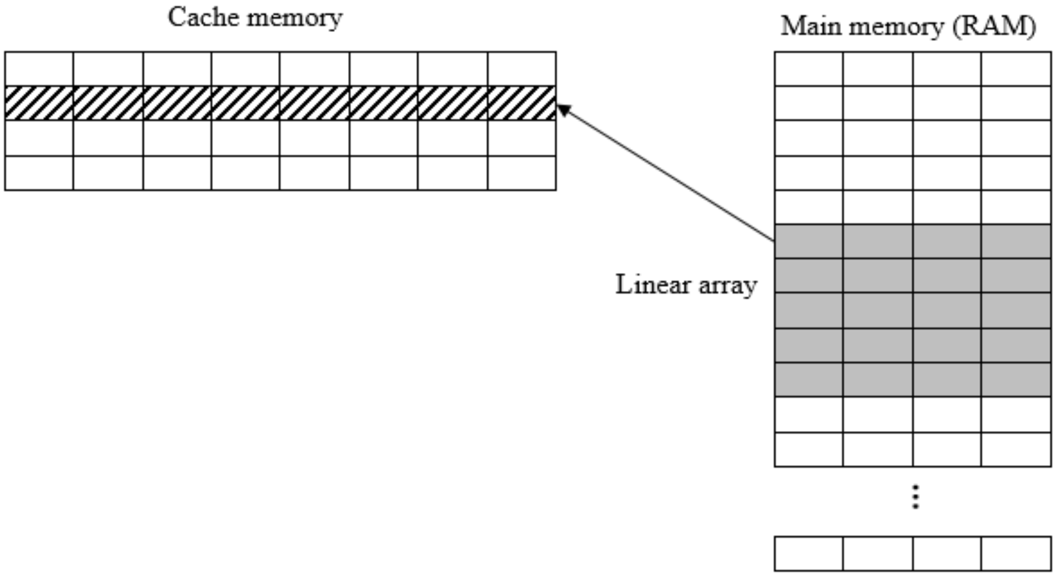
Introduction

A cache memory is an important part of the memory hierarchy in any computer system. It is physically placed between the registers, which is part of the processor and random access memory (RAM). Logically, the size of a cache is larger than the number of registers but significantly smaller than that of RAM. Cache memory is made from static (RAM) SRAM, which is different from the main memory or RAM, which is made from dynamic RAM (DRAM). The purpose of cache memory is to give the illusion of a large (RAM size), fast (cache access speed) memory. In order to maintain this illusion, it is fundamentally important for the program to efficiently use the cache. Otherwise, program performance (the inverse of the execution time) is dictated by the speed of the much slower RAM. A classic diagram of the memory hierarchy is shown in the figure below.



Cache memories operate using the principles of spatial and temporal locality. Spatial locality means that it is very likely additional nearby memory words will be accessed when a memory request is made. To capitalize on this principle several nearby memory words are brought into the cache when a single memory request is made. Temporal locality means if a memory word was just accessed it will likely be accessed again in the very near future. This principle is utilized by keeping memory words in the cache as long as they are recently accessed.

Cache memories are initially empty and are filled as the processor makes memory requests. To exploit the principle of spatial locality, several nearby (typically 64 bytes) bytes are also fetched from main memory and a copy of their content is placed in the cache memory. This way if sequential elements are requested by the processor they are already in cache and are accessible at cache speeds. The figure below depicts a smaller cache with a linear array stored in main memory. When the processor requests the first element in the array several adjacent elements are brought into the cache.



Because cache memories are necessarily small, (small memories are faster memories) it is very possible items in the cache will need to be overwritten with different memory words as a program utilizing a large amount of data executes. When this happens, it is said a cache miss has occurred and accessing the slower RAM is necessary. Minimizing cache misses are fundamental to making programs run efficiently. This becomes very difficult when the amount of data a program processes becomes rather large.

Quick review questions

1. What technology makes accessing the cache faster than accessing the main memory?
2. What are the two principles the cache utilizes?
3. Why is the cache much smaller than main memory?
4. What happens when the cache is full and another memory word must be accessed?

Program description

To experimentally demonstrate the performance of the cache memory, a program called ***memStrideT.c*** was written. It requires the user to supply two integer arguments, which are first tested for inclusion.

```
if (argc != 3)
{
      fprintf (stderr, "USAGE: %s nMB stride\n", argv[0]);
      exit(-1);
}
```

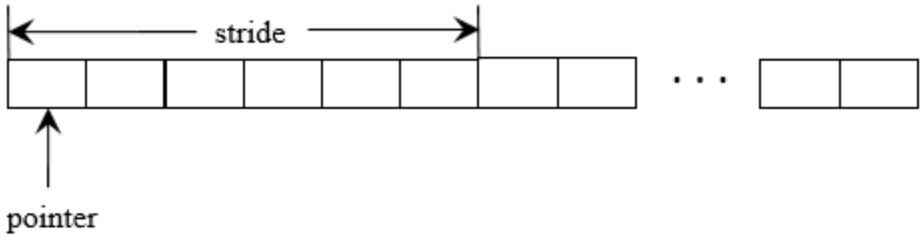
The first parameter is the number of Mebi (abbreviation for 2^{20}) elements that should be allocated. This number represents the size of the problem being solved. Reasonable values might be 50 or 100.

```
int nMB = strtol(argv[1], NULL, 10);
nMB *= MEGA;
```

The predefined constant MEGA has the value of 1024 * 1024 = 2^{10} * 2^{10}. The multiplication converts the value supplied as the first argument into the number of Mebi elements. The second argument specifies the stride, which indicates where the next element in the array of elements is located relative to the current element.

```
int stride = strtol(argv[2], NULL, 10);
```

A value of one means adjacent elements will be accessed. The one means the next element is one element away from the current element. A value of two means to skip one element between accesses, which means the next element to be accessed is two elements away from the current element. There is a similar meaning for other values for the stride argument.



The linear array of data elements is then allocated and checked for a valid allocation. A previous typedef allows the user to change the underlying data type for all the elements in the array.

```
myDataType *buffer = (myDataType*)malloc(nMB * sizeof(myDataType));
 if (buffer == NULL)
 {
        fprintf (stderr, "ERROR: malloc failed\n");
        exit(-1);
 }
```

Nested for loops are used to ensure the elements skipped on previous passes are eventually accessed. To prevent the compiler from optimizing our memory access out of the program a set value will be written into each element. Finally, several iterations of striding through the linear array are averaged to give the final running time of the program to eliminate spurious results.

```
for (int pass = 0; pass < numPasses; ++pass) {
        for (int j = 0; j < stride; ++j) {
                for (int i = j; i < nMB; i += stride)       {
                        buffer[i] = val;
                }
        }
}
```

The time it takes to execute these nested loops is recorded and the reported time is the total time divided by the number of passes. The output is the stride followed by a comma and then the average time for one iteration through the array, providing a comma separated output format.

Quick review questions

5. How would an array of structs be allocated?
6. What would you expect the run time to be if the array was accessed element by element but starting at the end of the array and moving to the beginning?
7. How could you efficiently access random elements within the array?

BASH Shell Script description

The shell is the command interpreter of UNIX/LINUX systems. One of the very powerful software development features of these systems is the ability to control program execution through batch shell scripts. Using a script, an executable program can be executed multiple times passing in different arguments without user interaction.

There are several different “shells” that are common on UNIX/LINUX systems. All of the shells can be programed with a shell script but there are language differences among the shells. One of the more popular shells is known as the Bourne Again Shell, otherwise known as bash. In UNIX/LINUX any file can be made executable by the shell by simply setting a permission, even a text file can be executable. It is convention to give a file extension of **.sh** to a file to identify it to the user as a bash shell script. See the **chmod** command for more details. The first line in a bash script is:

```
#!/bin/bash
```

which is the complete pathname of the program that is to execute the statements contained in the file. When this line is executed a new bash shell is started and each line in the file is executed, just like a normal serial program.

Since a bash script is a program it has many common programing features like variables, conditional statements, loops, and more. The first statement in this bash script is:

```
make memStriderT
```

which ensures the program to be executed within the script is currently compiled. The next statement

```
OUTFILE=memStrideDouble
```

sets the value of a variable named OUTFILE to memStrideDouble. This allows the programmer to simply change the name of the output file once instead of having to search the file and make multiple changes. A conditional expression is then executed to delete the output file if one is in the current directory.

```
if [ -f $OUTFILE.csv ]; then
      rm -f $OUTFILE.csv
fi
```

Following this a variable named COUNTER is assigned the value of one, which is the value to be passed into the program to represent the stride. A stride of one simply means the program should access every element in the linear array. Following this variable initialization the script enters a while loop to execute the program multiple times with different arguments.

```
while [ $COUNTER -le 133 ]; do
      ./memStrideT 100 $COUNTER >> $OUTFILE.csv
      echo $COUNTER           #Progress Visualization
      let COUNTER=COUNTER+1
done
```

This loop will execute as long the variable COUNTER is less than or equivalent to 133. The first statement in the body of the while loop executes the program requesting the array be 100 Mebi elements long with a stride given by the value of COUNTER. The output of the program is appended to the OUTPUT.csv file. The echo statement simply displays the value of COUNTER to the console window so the user can monitor its progress. Finally, the value of COUNT is incremented by one and the loop ends.

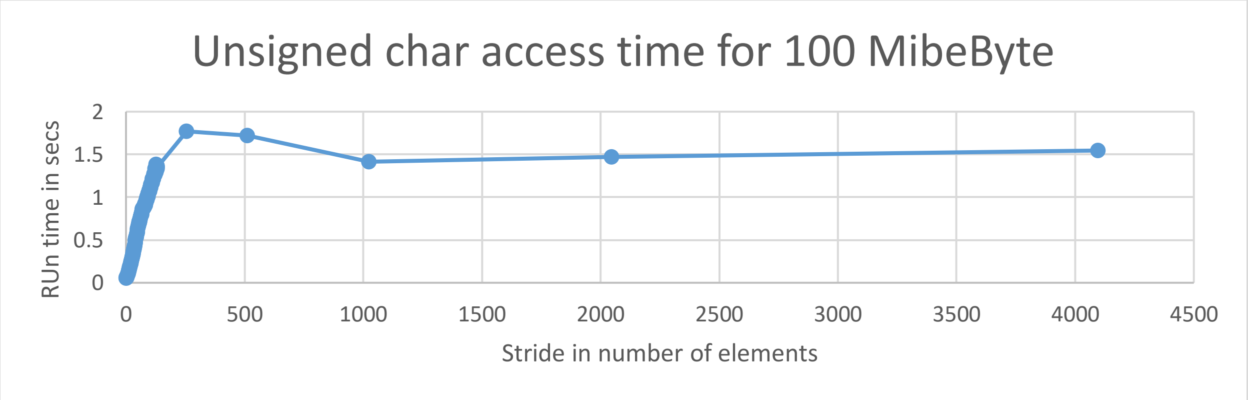
A second, similar loop is used to increment the stride by a power of two from 256 to 4096.

Quick review questions

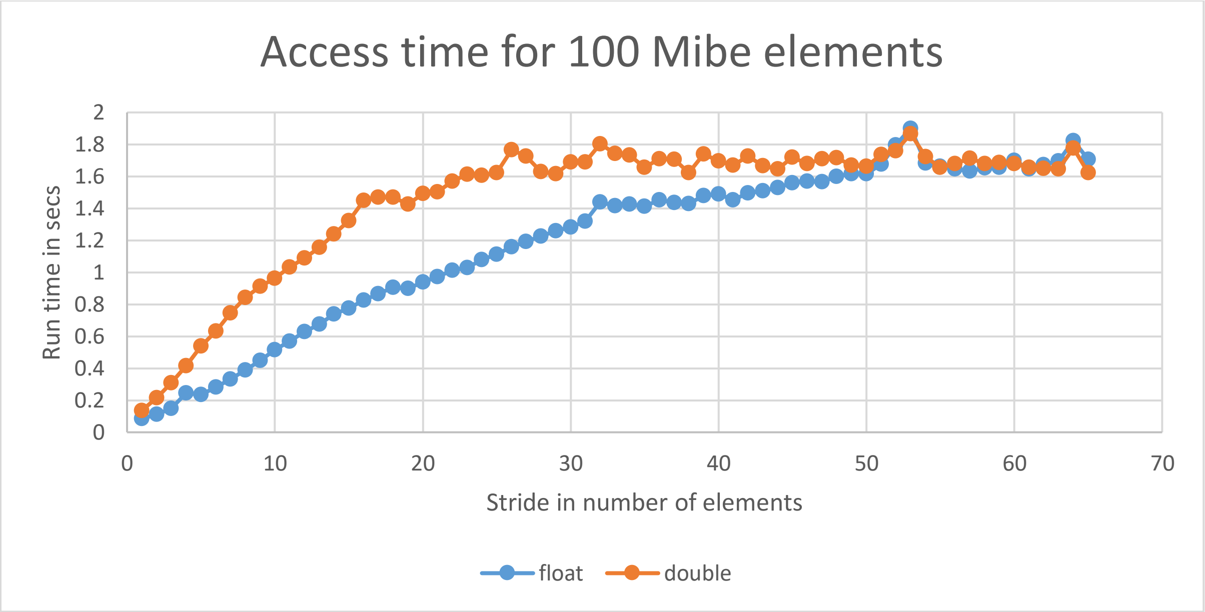
8. Write some statements that could detect if a file was not found in the current directory.
9. Write a while loop that increments a variable by the next power of two during each iteration.

Results

The results of running the program via the shell script for various strides and plotting the results clearly shows the program execution time remains relatively constant when the stride is 128 bytes and larger when traversing an array of unsigned characters as shown in the figure below.



This plot shows that when the stride is 128 bytes or greater the cache is effectively not being used. Many scientific problems require the data to be in a floating point format, so the program and shell script were used to stride through single- and double-precision data as shown in the figure below.



It appears from these plots that 32, single-precision floating point numbers and 16, double-precision numbers or greater the cache is not being used as all memory fetches require a RAM access. Since single-precision floating point numbers are each four byes and double-precision numbers are each eight-bytes the cache becomes ineffective when the stride is greater than 128 bytes.