1 Measuring Execution Time

Print sum (on), hardcoded (off)
–With -O2 enabled it takes 107938 ticks.
–Without -O2 enabled it takes 147217 ticks.
Print sum (on), hardcoded (on)
–With -O2 enabled it takes 78011 ticks.
–Without -O2 enabled it takes 139840 ticks.
Print sum (off), hardcoded (off)
–With -O2 enabled it takes 8021 ticks.
–Without -O2 enabled it takes 69745 ticks.
Print sum (off), hardcoded (on)
–With -O2 enabled it takes 1245 ticks.
–Without -O2 enabled it takes 67005 ticks.

Discrepancy in timing:
O2 is a compiler flag that tells the compiler to optimise your code for performance. Hence, without the -O2 flag the code runs slower and it takes more 'ticks' to execute. Hardcoding, your inputs takes off the overhead of processing your inputs from the command line. e.g we need to make the string "10000" into and integer before we can use it. Hence hardcoding improves performance. Printf, also takes time to execute and therefore the additional ticks when we print the sum.

Best Configuration:
The best config if: print off; hardcoded on; -O2 flag off. With this configuration we exclude the overhead of input/output processing and test the performance of the bare algorithm, since we do not let the compiler do any magic on it.

What is argv:
Argv is an array of character arrays (strings) of length argc where the strings at index 1, ..., n are the inputs in the command line and argv[0] is the name of the executable. argv[1] = "10000" in our case.

What is atoi():
atoi() is a function in the standard library that takes a string as argument and returns the corresponding integer.

2 Measuring Memory Latency

How does the pointer chase benchmark measure memory latency:
By accessing memory addresses in random order within a certain memory bound, you are not exploiting spatial locality. In a way, it is like writing code with no memory optimisation and therefore it gives you a good memory benchmark.

Size in bytes of an array of ints of size n:
Size in bytes is $n \cdot sizeof(int)$. On the hive machines $sizeof(int) = 4$ bytes.

Cycles per step vs the size of the array
The following is a plot showing the relationship between the average number of cycles to execute a step (vertical axis) and the size in bytes of the array (horizontal axis). The size varies from 32KBytes to approximately 8MBytes, with a step that increases by powers of 2. i.e. $X = x|x = 32000 + 2^n$ where n is an integer between 0 and 21 included.

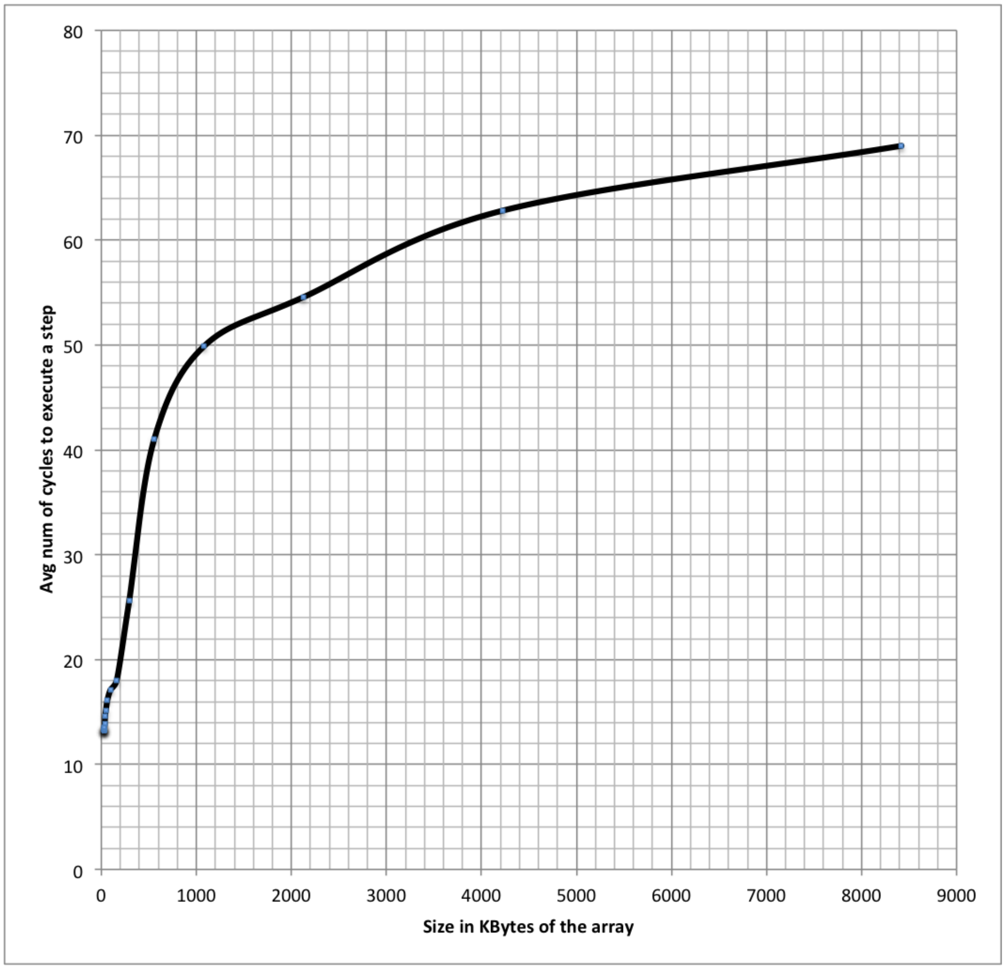

Figure 1: Avg number of cycles to execute a step and the size of the array

As the size of the array increases, so does the average number of cycles to execute a step. This is caused by the larger amount of memory accesses that the program needs to perform with increasing array size. The change in the slope happens when the array is about 1MByte large, because the proportion of L1 cache misses for smaller sizes is larger than the proportion of L2 cache misses for the larger sizes.

How can we make sure that the steps are not being pipelined:
Pipelining would not affect our benchmark since each step is strictly dependent on the previous step. i.e. each step needs to wait for the address that the previous step is fetching to fetch the next address.

BONUS:

Given an initialised int array with n ints, randPerm modifies the given array so that it contains all the ints from 0 to n-1 in random order in a way the pointer chase routine will access every value of the array. This task is achieved by creating a random cycle representation of the permutation of $\{0, ..., n-1\}$ and by converting it into a 2 by n matrix representation, where the first row represents the origin and the second row represents the destination. This matrix representation as a 1-to-1 correspondence with a C-array representation where the first row gets mapped to addresses and the second row gets mapped to the corresponding values of those addresses. This routing ensures that the pointer chase will go through cycles of length n, because of the way a permutation cycle is defined and how we construct the respective array for the pointer chase. With a small modification we could create random permutations of a set n with any cycle recipe.

Listing 1: Random Permutation with 1 cycle

```
/* Sets ARRAY to an array representation of an
   N-Permuation with one cycle.*/
void shuffleArray1c(int array[], int size) {
  // init random cycle permutation  array of size SIZE
  int * cycle_array = (int *)malloc(size * sizeof(int));
  shuffleArray(cycle_array, size);

  //corner case: glue the two ends of the cycle together
  array[cycle_array[size - 1]] = cycle_array[0];

  //go through the cycle permutation array to construct the result array
  for (int i = 0; i < size - 1; i += 1) {
    array[cycle_array[i]] = cycle_array[i + 1];
  }
  free(cycle_array);
}

/* Initializes ARRAY of size SIZE to (0, ..., SIZE-1)
   and shuffles it using Fisher-Yates algorithm. */
void shuffleArray(int array[], int size) {
  //initialize array (0, ..., size - 1)
  for (int i = 0; i < size; i += 1)
    array[i] = i;

  //seed the random number generator with the current time
  srand(time(NULL));

  //shuffle array using Fisher-Yates algorithm
  for (int i = 0; i < size; ++i) {
    int rand_index = rand() % size;
    int t = array[i];
    array[i] = array[rand_index];
    array[rand_index] = t;
  }
}
```

3 Measuring Memory Bandwidth

Average bandwidth vs the size of the array
The followings are plots showing the relationship between the average bandwidth (in Mbps) sustained by the memory system and the total size (in bytes) of the array for each of the three copy routines, our own routine, simd_memcpy and simd_memcpy_cache. The size of the array varies from 32 kilobytes to 8 megabytes.

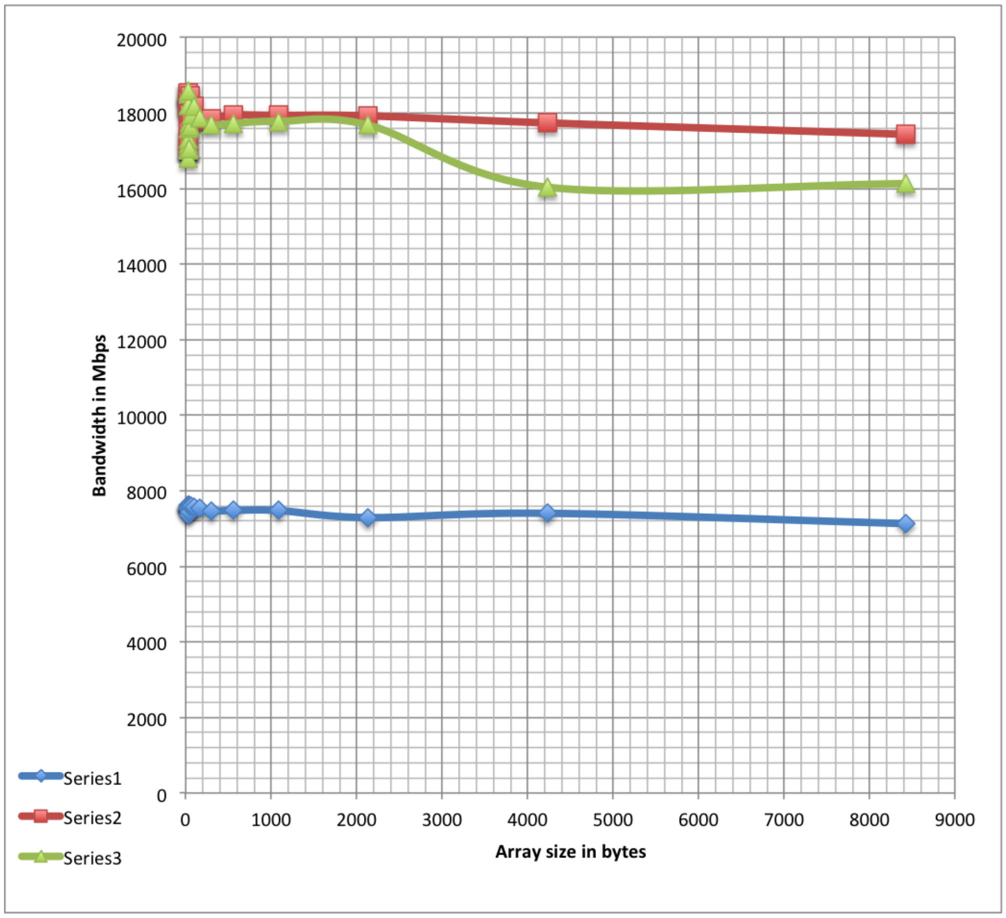

Figure 2: series 1: inefficient routine; series 2: simd_memcpy; series 3: simd_memcpy_cache

Discuss the shape of the curve:
The drop in memory bandwidth around 2MB is due to the fact that by copying larger arrays, we are beginning to utilise L3 cache more. Hence we reduce the average bandwidth by having to fetch data from cache with lower bandwidth more often.

Why is it necessary to warm up the cache:
We are doing a performance test against a system that usually has a high number of hits. Without the warm-up we would get false number because of the initial compulsory misses.

Why an inefficient copying procedure would yield an inaccurate measure of the maximum bandwidth:
Because it will utilise lower level caches or memories more frequently than it could achieve by

optimising the code. Lower level caches and memories have less bandwidth and hence by having to access them more frequently the average bandwidth gets lower than the maxim possible average bandwidth.
Explain the SSE instructions:

- _mm_prefetch (char const* p, int i): Fetch the line of data from memory that contains address p to a location in the cache hierarchy specified by the locality hint i.
- _mm_load_si128 (__m128i const* mem_addr): Load 128-bits of integer data from memory into dst. mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.
- _mm_stream_si128 (__m128i* mem_addr, __m128i a): Store 128-bits of integer data from a into memory using a non-temporal memory hint. mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.
- _mm_store_si128 (__m128i* mem_addr, __m128i a): Store 128-bits of integer data from a into memory. mem_addr must be aligned on a 16-byte boundary or a general-protection exception will be generated.

4 Measuring Flops and IPC

	IPC	GFlops
simd	2.34	9.175
scalar1	2.60	1.104
scalar0	0.73	0.874
naive	0.33	0.195

Floating point operations performed to multiply square matrices of size n:
We require n multiplications and $n-1$ summations to perform the dot product of two vectors of size n. An there are n^2 such dot products we need to perform. Hence, the number of floating point operations (F) to multiply square matrices of size n is:

$$F = (2n-1) \cdot n^2$$

How can you have $IPC > 1$:
Say the we have a simple architecture that has an $IPC = 1$. If we can combine this architecture with another equal architecture in some smart way we could have $IPC = 2$. Processors that implement superscalar architecture could achieve $IPC > 1$.

Why a higher IPC is not always an indicator of an efficient program:
Because we could write a program that could achieve the same task by utilising half the the instructions that is utilising right now. The second IPC would lower than the first one, but the time it takes to execute second program could be worse than the first one.