Monte Carlo, Gaussian Blur, and Radix Sort on CPU

Nidhi Bhatia **Nick Kiesel** **Rajnish Aggarwal**

1 PROBLEM DESCRIPTION

Today, most of the software for data intensive applications, i.e. machine learning, technology behind Ethereum(cryptocurrency) mining etc., are optimized for GPUs than anything else. GPUs have almost 200 times more processors per chip than a CPU. For example, an Intel Xeon Platinum 8180 Processor has 28 Cores [4], while an NVIDIA Tesla K80 has 4,992 CUDA cores [2]. While a CPU core is more powerful than a GPU core, the vast majority of its power goes unused by data intensive softwares. A CPU core is designed to support an extremely broad variety of tasks (e.g., render a webpage, drive enterprise software, etc.) in addition to performing computations, whereas a GPU core is optimized exclusively for data computations. Consequently, these softwares, which perform large numbers of computations on a vast amount of data, can see huge performance improvements when running on a GPU versus a CPU. In fact, many studies claim that GPUs deliver substantial speedups (between 10X and 1000X) over multi-core CPUs on these softwares [5].

We propose to optimize three important throughput computing kernels (generally optimized to run on GPUs) for Intel Xeon Processor E5-2620 v3 (Haswell CPU Platform - ece001.ece.local.cmu.edu). We will consider the Haswell CPU's architectural features for optimization and compare its improved performance with the non-optimized baseline kernel as well as the fine-tuned GPU version(if time permits). In addition, we identify the key architecture features of Haswell CPU that benefit selected throughput computing kernels. In this project we also aim to highlight the importance of platform specific software optimization when benchmarking CPU and GPU.

We selected three kernels, namely, **Monte Carlo, Gaussian Blur and Radix Sort**, that have been identified by previous studies [5] as important components of throughput computing workloads. The code can be found on Github at https://github.com/kieselnb/benchmark-optimizations.

2 MONTE CARLO

Monte Carlo is a computational algorithm that relies on repeatedly random sampling to obtain numerical results. We use an example of Monte Carlo which estimates the value of pi.

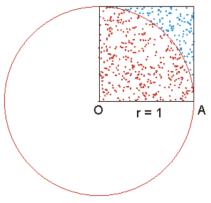

Figure 1: **Points inside quarter circle(red) and bounding quarter square(red and blue)**

It simulates random float (x,y) points in a 2-D plane with domain as a square of side 1 unit. These points can be in any position within the square i.e. between (0,0) and (1,1). Imagine a circle inside the same domain with same radius and inscribed into the square. We keep track of the total number of points, N_{total} and the number of points that fall inside the quarter circle (red, Figure1), N_{circle}. If we divide the number of points within the circle, N_{circle} by the total number of points (red+blue, Figure1), N_{total} we should get a value that is an approximation of the ratio of areas of quarter circle and square,

π/4 as shown below.

$$\frac{\text{area of quarter circle}}{\text{area of square}} = \frac{\pi}{4} = \frac{\text{number of points inside quarter circle (red points Figure1)}}{\text{number of points inside square (red+blue points Figure1)}}$$

Algorithm 1 Estimating the value of Pi

```
1: Pre-generate N_total number of random float (x,y) between (0,0) and (0,1)
2: N_total number of random float x stored in array X
3: N_total number of random float y stored in array Y
4: procedure ESTIMATEPI(X, Y, N_total)
5:     Initialize N_circle ← 0
6:     for i ← 0 to N_total do
7:         d ← X[i]^2 + Y[i]^2
8:         if d ≤ 1 then
9:             incr N_circle
10:        end if
11:    end for
12:    pi = 4 * N_circle / N_total
```

2.1 Optimized Kernel Description

As mentioned above, for estimating the value of Pi we need to check $x^2 + y^2 \leq 1$ for every pre-generated random float point (x,y). If the condition holds true, we increment N_{circle}. The *baseline implementation* of the algorithm is not optimized to use the architectural features i.e. vector execution units of the Haswell CPU. Therefore, it does not give good throughput on Haswell CPU.

In order to optimize the algorithm for better throughput on Haswell CPU, we decided to use vector execution units available on Haswell CPU. Haswell has 256-bit FP FMA and FP FMUL on port 0 and port 1. This makes it possible for FMAs and FMULs to issue on both ports. In theory, Haswell can perform 16 double / 32 single precision FLOPs/cycle [9].

Knowing this, we decided to use capabilities of

- FP FMULs (port 0 and port 1, Latency: 5 cycles, throughput: 2) to calculate x^2
- FP FMAs (port 0 and port 1, Latency: 5 cycles, throughput: 2) to calculate $x^2 + y^2$

To find the number of random float points which follow $x^2 + y^2 \leq 1$, we used vector instructions,

- To Compare - vcmple_oqps (port 1, latency: 3 cycles, throughput: 1)
- Movemask - vmovmskps (port 0, latency: 2 cycles, throughput: 1)
- PopCount - scalar instruction popcnt (port 1, latency: 3 cycles, throughput: 1) [3]

Calculating x^2 is the **only independent operation** in our kernel. FP FMULs which are used for calculating x^2 have the highest latency of 5 cycles along with FP FMA amongst all our operations. In order to hide the latency of FP FMULs, we decided to **pipeline multiple independent instructions**. Since latency of FP FMUL is 5 cycles and throughput is 2, we can pipeline, $5 \times 2 = 10$ independent instructions to hide its latency.

We pre-generate float random (x,y). Vector execution units in Haswell are 256 bit wide, hence by using floats we can run 8 FLOPs/instruction instead of 1 FLOP/instruction in the baseline implementation. This also helps us determine our minimum kernel size, which is equal to

10 independent instructions $\times$ 8 FLOPs/instruction = 80 independent FLOPs

Hence, our kernel simulates 80 random float (x,y) points each time its called.

2.2 Theoretical Peak Calculation

Of all our kernel operations, FP FMULs and FP FMAs have the highest latency of 5 cycles, which means they form the performance bottleneck for our implementation. There are 2 functional units

each for FP FMUL and FP FMA. As mentioned before FP FMUL and FP FMA are present on both port 0 and port 1. This implies that in 1 cycle we can run

$$8 \times (2 \text{ FMA ops} + 1 \text{ FMUL op}) = 8 \times 3 \text{ FLOPs} = 24 \text{ FLOPs}$$

Hence, the **Ideal theoretical peak** for single core implementation becomes

$$\text{Base Frequency (GHz)} \times \text{ CPU FLOPs per cycle} = 2.4 \text{ GHz} \times 24 \text{ FLOPs/cycle} = \textbf{57.6 GFLOPs/s}$$

But the other dependent instructions, vcmple_oqps, vmovmskps and popcnt also lie on port 1, port 0 and port 1 respectively. Since, FP FMULs and FP FMAs are also on port 0 and port 1, this creates congestion on port 0 and port 1 (shown in Figure 2).

Due to congestion on port 0 and port 1, our kernel may never be able to hit the ideal theoretical peak mentioned above. In order to reduce congestion on port 0 and port 1, we tried finding other alternate instructions to replace these dependent instructions but we were unsuccessful. All these instructions are important for the correct implementation of our kernel. Therefore, we scaled our ideal theoretical peak to a more realistic and achievable theoretical peak.

Number of Cycles	Port 0	Port 1	Ideal	Realistic
Cycle 3		PopCount		24 FLOPs/(2-3) cycles
Cycle 2	Movemask	Compare		24 FLOPs/approx. 2.5 cycles
Cycle 1	FP FMA	FP FMUL	24 FLOPs/cycle	9.6 FLOPs/cycle

Figure 2: **Haswell Pipeline: Port 0 and Port 1 congestion**

In the ideal scenario (as mentioned above), we can run 24 FLOPs/cycles. As shown in Figure 2, due to congestion at port 0 and port 1, 24 FP FMUL and FP FMA operations will now run in 2-3 cycles(approx. 2.5 cycles). Hence, we can run approx. 9.6 FLOPs/cycle. Using this we calculate our **realistic theoretical peak**.

$$\text{Base Frequency (GHz)} \times \text{ CPU FLOPs per cycle} = 2.4 \text{ GHz} \times 9.6 \text{ FLOPs/cycle} = \textbf{23.04 GFLOPs/s}$$

We will use **realistic theoretical peak** for our forthcoming single core comparison plots and will refer to it as **theoretical peak** from now on.

2.3 Single Core Performance

The optimized implementation of monte carlo algorithm for single core Haswell CPU clocks at $\sim$**90%** of the theoretical peak, also shown the Figure 3.

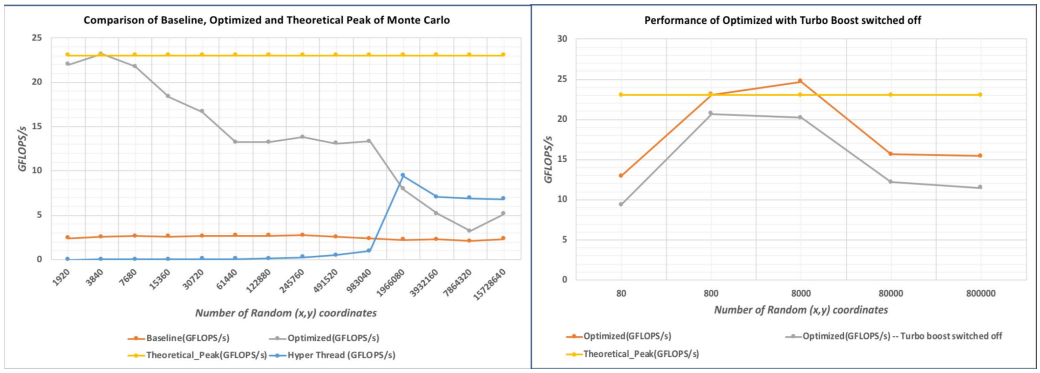

Figure 3: **(Left) : Single Core Performance Plot: Shows Baseline implementation performance, Theoretical Peak Performance, Optimized Kernel Performance, Hyperthread Expts Performance. (Right) : Plot showing performance of optimized kernel on ece cluster Haswell CPU w.r.t Haswell CPU whose Turbo Boost Mode has been switched off.**

KEY OBSERVATIONS

- As you may also notice on the left plot in Figure 3, optimized kernel performance curve (in grey) almost crosses the theoretical peak when number of random (x,y) simulated (on x-axis) is 3840.

  Ideally, the achieved performance of optimized kernel should never cross theoretical peak. *But this might happen because of anomalies in recording elapsed cycles due to turbo boost contributing to the faster-than-light measurements.*

- Since for many random (x,y) data set sizes on x-axis in Figure 3, we are achieving performance for optimized kernel really close to the theoretical peak, we performed two important experiments to confirm the authenticity of our results.
  - **Hyperthreading** : We re-simulated our optimized kernel on hyperthreads of the single core. Performance plot of hyperthreading experiment (shown in Figure 3) shows almost no performance gain. This implies that our single core optimized implementation is already saturating the available functional units. Thus, we are running really close to the theoretical peak. **Please Note:** These measurements are subject to vary based on the loading on the machine you are working on.
  - **Switch off Turbo Boost Mode** : We also re-simulated our optimized kernel on Haswell machine where turbo boost mode has been switched off (shown in the Figure 3 on the right). We observed that for all the points which crossed theoretical peak or went really close to it, was only because turbo boost was contributing to their faster-than-light performance. This confirms that our optimized kernel runs at about $\sim$**90%** of the theoretical peak but never crosses it.
- **Why only 90% of theoretical peak** : This question may come to your mind, why are we achieving only 90% and are happy about it? Why did we not reach 100%? We stopped at 90% and became all happy is because its really hard to achieve further performance improvement once you are so close to the theoretical peak. The reasons for this could be memory boundedness or overheads like loop branch, pointer arithmetic etc. It is really hard to avoid these beyond a point.

2.4 Parallelism

We enabled our kernel to run on different cores available on the Haswell CPU. We used OpenMP to parallelize our algorithm. We tried experimenting with parallel for, parallel region, tasks etc. but *parallel for* seems to work the best for us. Hence, we used OpenMP parallel for to parallelize our algorithm. Figure 4 below shows the performance plot for our parallel implementation for 6, 12 and 24 threads.

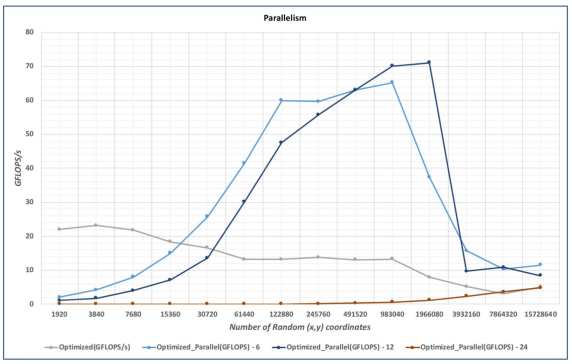

Figure 4: **Parallelism Performance Plot: Shows Single Core Performance, Performance for 6, 12 and 24 threads**

KEY OBSERVATIONS

- When we introduce parallelism, for smaller data sets of random (x,y) points, the overhead of parallelism is high. Due to this we observe, that performance provided by 6, 12 and 24 threads is lower than single core performance (shown in Figure 4).

- For any given number of threads only if size of input data set crosses a certain threshold size, is when it starts to show performance higher than the corresponding single core implementation. (shown in Figure 4)

- After input data set size of 15360K (point 983040 on x-axis, Figure 4) performance starts to degrade because beyond this L3 is full and data access happens from DRAM-based main memory.

- For 24 threads, performance gain is negligible. This is due to overhead of parallelism introduced because of 24 threads and impact of non-uniform memory access (NUMA). The Haswell machine we worked on has two NUMA nodes, effect of lauching 24 threads may have spawned few threads on the different NUMA nodes. Due to which we see a severe performance drop.

KEY LEARNING PARALLELISM

While optimizing parallel implementation for performance, we captured a key phenomenon called *false sharing*. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which hurts performance.

We observed this while optimizing our parallel implementation for performance. In our parallel implementation, in order to assign each thread a local copy where it can increment the number of points inside circle, N_{circle} independent of other threads, we allocated an array whose number of elements (unsigned long long) were equal to number of threads. Since, all these elements will be continuous in the memory and will belong to the same cacheline, false sharing will happen.

In order to avoid false sharing, we allocated array of size of cacheline times number of threads. We indexed inside array in such a way that each thread now has its own exclusive cacheline to work on. There is no performance degradation due to cache coherence in our latest parallel implementation. Figure 5 below captures the impact of false sharing seen for parallel implementation using 12 threads.

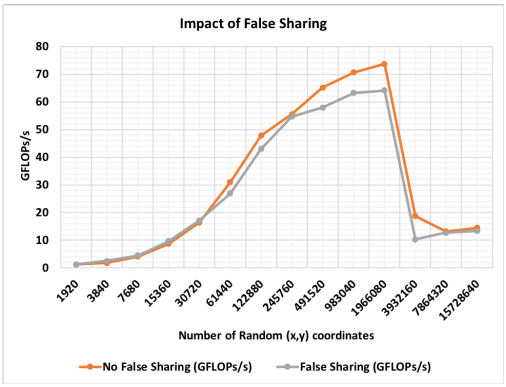

Figure 5: **Impact of False Sharing for parallel implementation using 12 threads**

2.5 FUTURE WORK ON MONTE CARLO

- We plan to work on determining the accuracy of a novel idea to estimate the value of pi. In this we consider points which lie exclusively inside circle and not on its boundary. This helps us remove compare operation from our kernel which will help us further increase the throughput of our algorithm.

- We did few experiments to reduce the impact of NUMA on parallel implementation with 24 threads but it did not give us much benefit. We plan on trying double buffering and check if that helps us to improve the parallel performance with 24 threads.

3 GAUSSIAN BLUR

3.1 Algorithm Description

The Gaussian blur is a way of smoothing edges in an image. This is done by performing a convolution on the image with a very particular mask. This mask is generated using a Gaussian distribution. This is scalable to n-dimensions - we will be focusing on the two-dimensional case, in which a symmetric matrix is produced whose numbers follow a Gaussian surface. The mask is generated with the equation $G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$. For each output pixel, the mask centered on the corresponding input pixel - the pixels overlayed by the mask are scaled by the value of the mask at its location and added into the output pixel. Thus, the output pixel is produced by combining the input pixel with exponentially decreasing amounts of the surrounding pixels.

3.2 Peak Performance

- **Core Kernel** The generation of the filter is a one-time process, so it will not be considered for determining the bottleneck. The core kernel is that of multiplying a filter of values with a same-sized section of the input image, then summing all of the products into a single output pixel. However, the algorithm can be modified so that one mask value is multiplied to a vector of input pixels, which are then summed into a corresponding vector of output pixels. This removes the sequential addition from the kernel, allowing everything to be done in parallel.
- **Bottleneck** Since everything is now done in parallel, the bottleneck is the most-used instruction that has the highest latency and lowest throughput. The only instruction used in the computation part of the kernel is a fused multiply-add, so that will be the bottleneck of this algorithm.
- **Theoretical Peak** The target CPU architecture - Intel Haswell - has two (2) functional units that support the FMA vector instruction, and has a latency of five (5) cycles [1]. We will be targeting single-precision floating point values with this kernel, and the FMA instruction can operate on eight (8) single precision floats at once. This puts our peak throughput at 8 x 2 = 16 FMA instructions per cycle, which corresponds to 32 floating point operations. Taking into account the frequency of the target machine, the peak throughput is 2.4 x 32 = 76.8 GFLOP/s.

3.3 Optimization Journey

3.3.1 Baseline Implementation

The base implementation of the Gaussian blur comes from Github[1]. It loops over each input image pixel and performs the convolution sequentially. The base algorithm is described in Algorithm 2.

3.3.2 Phase 2 Improvements

As mentioned earlier, all output pixel calculations are completely independent, allowing SIMD vectors to be used. Thus, the core kernel was expanded to have a width of 8 to match the SIMD floating point vector width and a height of 10 to hide the latency of the SIMD fused multiply-add (FMA). This resulted in a small performance gain since many operations were doing simultaneously - however, double-checking the generated binary revealed that 10 FMA instructions were too many for the architecture to handle. There are only 16 architected vector registers, and this version of the algorithm needs a register for both the input and output vectors, plus the broadcasted filter vector. Thus, for a kernel of 8x10, 21 vector registers are needed, which resulted in the program pushing vectors to the stack in order to complete the computation. Reducing the kernel to 7x8 brought the needed register count to 15, which removed the need to push values off to the stack.

In addition, loop interchange was tried in order to reuse values already in the vector registers. Progressing down through the filter allowed 6 of the 7 vector registers to be reused as is, so only one

[1] https://gist.github.com/OmarAflak/aca9d0dc8d583ff5a5dc16ca5cdda86a

Algorithm 2 Baseline implementation of Gaussian blur

```
1: InputImage ← image loaded from disk
2: OutputImage ← allocated the same size as InputImage
3: Filter ← generated Gaussian filter
4: procedure ApplyFilter(InputImage, Filter)
5:     for c ← 0 to ImageChannels do
6:         for y ← 0 to ImageHeight do
7:             for x ← 0 to ImageWidth do
8:                 for i ← y to y + FilterHeight do
9:                     for j ← x to x + FilterWidth do
10:                        OutputImage[c][y][x]+ = InputImage[c][i][j] ∗ Filter[i − y][j − x]
11:                    end for
12:                end for
13:            end for
14:        end for
15:    end for
```

register needed to access memory. This is in contrast to horizontal scanning that is normally done in convolution kernels - and is done in both the baseline and optimized versions described here - in which every vector needs touch memory in order to load the next element. The vertical scanning, however, saw a large drop in performance. This was due to the fact the kernel was now accessing a new row of the input image, leading to a very large jump in memory access, which guaranteed a long wait since the data was not very high in the memory hierarchy. The horizontal scanning method was accessing just-used cachelines on subsequent iterations of the filter, which resulted in faster load times despite having multiple vectors needing new data.

3.3.3 Helping Out the Compiler

After staring at the object dump of the binary with the mindset of interleaving independent instructions, I wondered if interleaving the loads with the FMAs would give any performance improvement. Interleaving the loads and FMAs gave quite a large improvement for a variety of reasons. First of all, it allowed the compiler to reuse architected registers. This allowed the kernel to return to its former size of 10x8 in order to hide the latency of the FMA. *However*, this resulted in the compiler using a version of the FMA instruction that goes directly to memory for one if its arguments - the input image, in this case. This is fine because after the first load, all next loads are sequential accesses in memory, which means the data is already in L1 cache.

3.3.4 Taking Advantage of Associative Cache

The next task was to be nicer to the cache when accessing data. The platform we targeted has 8 ways in its L1 cache, so the current version of the algorithm caused a lot of cache thrashing. This is because 8 rows of the input and 8 rows of the output are accessed every single time the core kernel executes. To solve this, the kernel was flattened to 5x16 instead of 10x8. Having the kernel be 16 elements wide preserves the alignment with the width of the SIMD vector registers, and having a height of 5 keeps the 10 FMA instructions needed to hide the latency. However, this still requires 5 ways of cache for each image. A final modification to make the kernel 4x16 was made, which saw a small but noticeable improvement in performance. Even though we are again below the number of FMA instructions required to hide its latency, the gain from not having to go further than L1 cache greatly outweighs that loss.

3.3.5 The Achilles Heel

The last thing we have been ignoring in this algorithm is the size of the filter. The current implementation allows for a variable size filter, which is the root cause for most of the cache thrashing. This is because the number of rows of the input image that we access does not match the number rows of the kernel - it is actually the number of rows in the kernel *plus* the width of the filter. Thus, for each iteration of the kernel, we will only access four rows of the output image, but more than four rows of the input image, causing more cache thrashing. In testing, giving a large filter width (> 8)

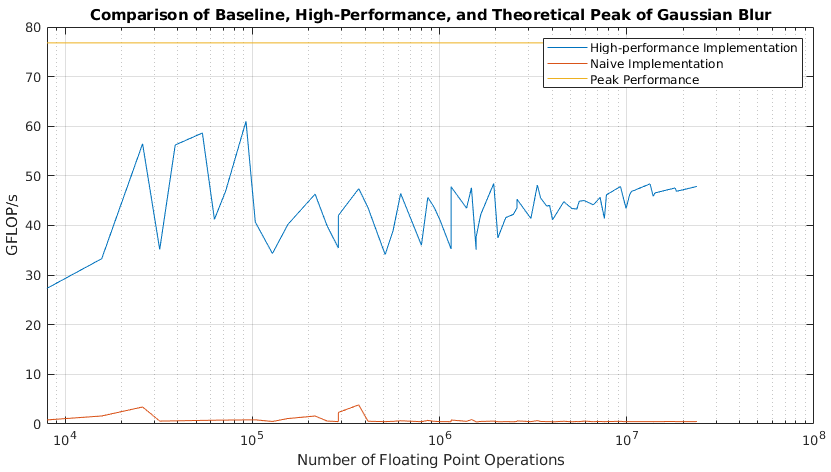

Figure 6: Performance of the baseline and optimized Gaussian blur kernels as compared to the peak compute performance. To vary the dataset, the test image was converted to multiple different sizes. For each image size, the filter radius was varied from 2 to 8.

caused performance to drop drastically. This makes sense because we are forcing each iteration of the core kernel to have to access at least L2 cache.

3.4 Performance

The performance of the last implementation of the Gaussian blur benchmark is shown in Figure 6. As can be seen in the figure, the current implementation does much better than the baseline implementation. At the maximum, this implementation is roughly 73% of the theoretical peak, and has an average of 57% of the theoretical peak.

3.5 Future of Gaussian Blur

3.5.1 Partial Updates

As mentioned in section 3.3.5, the main drawback of the current implementation of this algorithm is access pattern of the input image. Moving forward, a method of partial updates of the output that restricts the input image access to match the height of the kernel would greatly improve performance of the algorithm by allowing each input and output to have their own ways of cache.

3.5.2 Permute-Blend

After the move from a 10x8 kernel to a 4x16 kernel, we are now able to better reuse data between registers. This is because the two columns of vector registers sit side-by-side and we progress horizontally through the filter, so the leftmost element of the right hand vectors jumps over to the rightmost element of the left hand vectors. This was not attempted thoroughly because the compiler was using many instructions that have very long latencies, which ultimately took longer than the access to L1 cache. A more thorough approach by using hard-coded assembly may be possible with the permute and blend instructions to shift the elements over by one. However, the right hand vectors will still need an access to memory to retrieve the next element of the horizontal pass.

4 RADIX SORT

4.1 Algorithm Description

Radix sort is a not so popular sorting algorithm because although beign a stable sort, it requires additional memory to keep the sorted array. A typical radix sort uses individual digits in a number to sort over the given input. These numbers can start from the least significant or the most significant

digit. If the maximum number of the array has K digits, then we need K passes to sort the entire number. For every pass we:

- Put the digit in consideration into its radix, do this for all the numbers
- Construct a histogram with the total count of numbers up to this radix
- Re-compute the radix for every number and put it in its correct position

4.2 Peak Performance

The current implementation of radix sort uses the SIMD vector extract instruction within the kernel. This instruction is the bottleneck of the overall implementation. With a latency of 1 [3] and a CPU operating at 2.4GHz, the peak theoretical performance for this radix sort implementation is 2.4 GFLOPS/s. Considering the fact that there is a 87% chance that keys collide, the approximate number of cycles per iteration is 19. Hence if we have n numbers to be sorted, expected number of cycles are n * 19.

An important thing, that will be explained in a later section is that there can be faster instructions which we can use in radix sort, but there is a trade off between memory and speed that we need to make here. Using faster instructions in the compute stage, implies many memory load/store operations, which become the new bottleneck when the algorithm is implemented in a different way.

4.3 Optimization Journey

4.3.1 Baseline Algorithm

The basic implementation of Radix sort from GeeksForGeeks, employs trivial methods for sorting an integer array. Following are the steps they take using a base-10 radix:

Algorithm 3 Baseline algorithm for radix sort

1: Find the maximum number in the array and calculate the count of digits in it
2: For each of the counts, do
3: **procedure** BUCKET SORT(user input array)
4: Find the radix of the numbers 1 -> n using division operation
5: Maintain a count of how many times each number occurs
6: Construct a prefix sum/histogram to figure out where each number must be placed in a partially sorted array
7: In the last step, compute the radix again and based on the position calculated in previous step, put this number in it's correct position in a new temporary array
8: Copy all elements of the temporary array to the actual array

4.3.2 Phase 2 improvements

- The first thing that we noticed was that the division operation was a very costly one and there was no need to calculate the maximum array element. Given that we can work with binary, a shift operation would be best suited. With a shift width of 4, the total number of passes for each number is an exact number 8
- The second observation was that there was no need of copying data from one array to another. A simple pointer arithmetic could be used to reassign values
- Step 1 of radix sort does the shift and AND on the same element 8 times, do it in 1 go and store all the values
- Step 1 can also use SIMD for loading in data, but then is bound by the fact that we need to do a sequential read of all radix after shift and AND in SIMD. As expected, this did not give much performance benefit
- **We thought that the shift operation is really fast, however, shift with a constant value has a throughput of 2 whereas shift with a value loaded in register is 4 times slower [3]**

- All this time, we had a notion that the shift operations were preventing radix sort from reaching the peak (NAIVE!)

4.3.3 Wait! What is my bottleneck?

Spending all this time on optimization, it was still unclear what was preventing the algorithm from achieving theoretical peak. Was this implementation memory bound, or compute bound ? Literature had a lot to say, but no specifics for implementation and the why was broadly unclear [7][1][5][8]. Let's look a little closer at the implementation in phase 2.

Crucial Observations
Radix sort has 3 steps, step 1 calculates the radix using a shift and an AND operation. Step 2 does the prefix sum to find the correct position of the number in the final array. Step 3 recomputes the radix and copies the element in a temporary array in the correct position.

Wait! I already knew this! Well, let's observe the latent states of radix sort, ones we ignored

- Step 1 loads each array element once into a register (SLOW!) does the shift and AND. It is then stored in an array using a LOAD, STORE (SLOW!)
- Step 2 loads (SLOW!) each element, increments it's value and stores (SLOW!) it back
- Step 3 loads each array element again, computes the radix and stores in another array using LOAD, STORES (SLOW!)
- A major lesson here was that once we did away with the division operation, we were no longer bound by compute operations. It was memory that was bounding us

4.3.4 Introducing Cache Blocking and an opportunity to exploit SIMD

Having figured out the above reasoning, we wanted to now solve the problem where we could reduce the number of load stores. The main idea here was to eliminate creation of the histogram and allocation of a temporary array. This is where cache blocking came to our rescue:

- Let's say we consider 256 elements at a time in our kernel. Load 8 of these elements at a time using SIMD integer loads. Perform the SHIFT and AND operation to calculate the radix for these 8 elements.
- Now, image that our L1 cache is a 2D map occupying 4 ways and it organized as a 4 x 4 block. This implies each of the 16 blocks can store 1K Byte data or 256 32 bit elements. In the worst case, where all the elements map to the same radix, we have enough space in the cache to store each one. In our kernel, we have pointers to these 16 blocks of memory.
- At this stage, we have 8 input numbers and their corresponding radix in SIMD registers. Reading these values out of registers is independent of each other and we do this using the **SIMD extract instruction** which is the current bottleneck. Now, extract the radix and using the pointers place it in the correct position in the cache. Doing this, not only did we arrange the elements in a sorted order but also computed the number of elements belonging to each radix
- In the last step, read elements from the cache using SIMD loads and store them in the input array using SIMD stores, each of which are really fast instructions
- Also note how we eliminated the need to do a memory allocation on each iteration for the temporary array to just one, which can be done globally in the main function and is no more a part of the kernel

4.4 Performance

As can be seen from the figure, the Orange line represents the current best performance and the Red line shows results from experiments in Phase 2. At small input values, set up times are also significant and hence achieving theoretical peak is hard. We hit 80% of theoretical peak with the current radix sort implementation. The remaining 20% account for heuristics of whether keys will collide or not. Also, we have not accounted for the time taken to load store data from cache to input array as it is approximately insignificant compared to the main loop in the kernel.

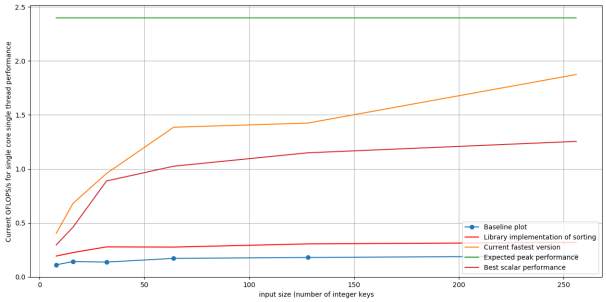

Figure 7: Radix sort performance for different optimization applied

4.5 Experiments that did not work out

- One of the major experiment that did not work out was use of SIMD with the scalar implementation. The primary reason for this was that SIMD could be used to only load elements and then after the SHIFT and AND, we had to do a sequential read of the values. We also update the counters for each radix after this, which involves another load store operation.
- Another idea we explored was using the American Flag Sort [6] algorithm in our kernel rather than the usual Bucket Sort. This eliminates the need for keeping a temporary array by doing an in-place swapping in the input array. This idea did not work because we added huge computational overhead while trying to do away with load/store operations. (THERE IS NO FREE LUNCH!)

4.6 Future of Radix Sort

Some of the things we would like to try out in future with radix sort include:

- Try reduce the kernel size to 8, instead of 256. In that way, we can use only our registers to sort all the elements
- Implement a fast merging algorithm to achieve good performance when sorting a big input array with sorted 256 element subsections

5 FUTURE WORK

The scope of the project called for benchmarking optimized CPU algorithms with GPU's. Now that we are ready with our CPU benchmarks, future work will involve their benchmarking. Last, the thought of having fast kernels that carry a general template for CPU algorithms is inspiring and we would like to move in that direction.

6 ACKNOWLEDGEMENTS

We would like to thank Professor Low, Mark Blanco, and Elliott Binder for their time and guidance throughout the semester.

References

[1] Arne Andersson and Stefan Nilsson. Implementing radixsort. *Journal of Experimental Algorithmics (JEA)*, 3:7, 1998.

[2] CPU-WORLD. *Nvidia Tesla K80*, 2014 (accessed December, 2018). `http://www.cpu-world.com/news_2014/2014111701_NVidia_Unveils_Tesla_K80_Dual-GPU_Accelerator.html`.

[3] Agner Fog. *Haswell Instruction Tables*, 2014 (accessed December, 2018). https://www.agner.org/optimize/instruction_tables.pdf.

[4] Intel. *Intel Xeon Platinum 8180*, 2014 (accessed December, 2018). https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38-5M-Cache-2-50-GHz-.

[5] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[6] Peter M McIlroy, Keith Bostic, and M Douglas McIlroy. Engineering radix sort. *Computing systems*, 6(1):5–27, 1993.

[7] Stefan Nilsson. *Radix sorting & searching*. Department of Computer Science, Lund University, 1996.

[8] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 351–362. ACM, 2010.

[9] Wikichip. *Haswell Micro Arch*, 2014 (accessed December, 2018). https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client).