PA 1 Solution Set

April 18, 2012

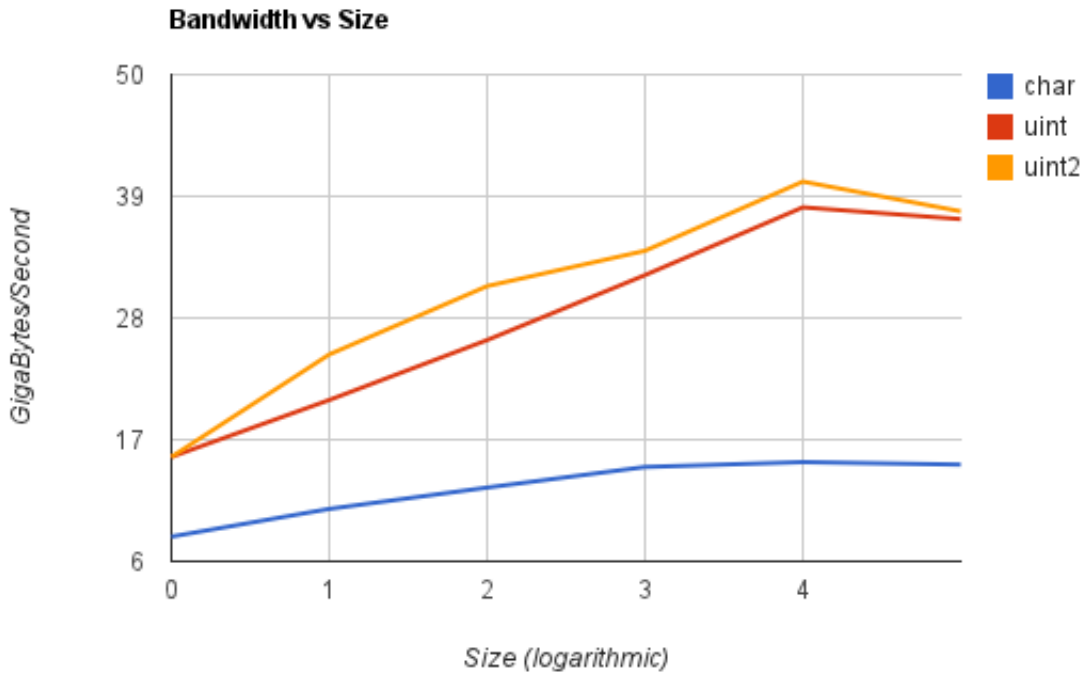
Correctness

See CUDA solutions for a correct implementation

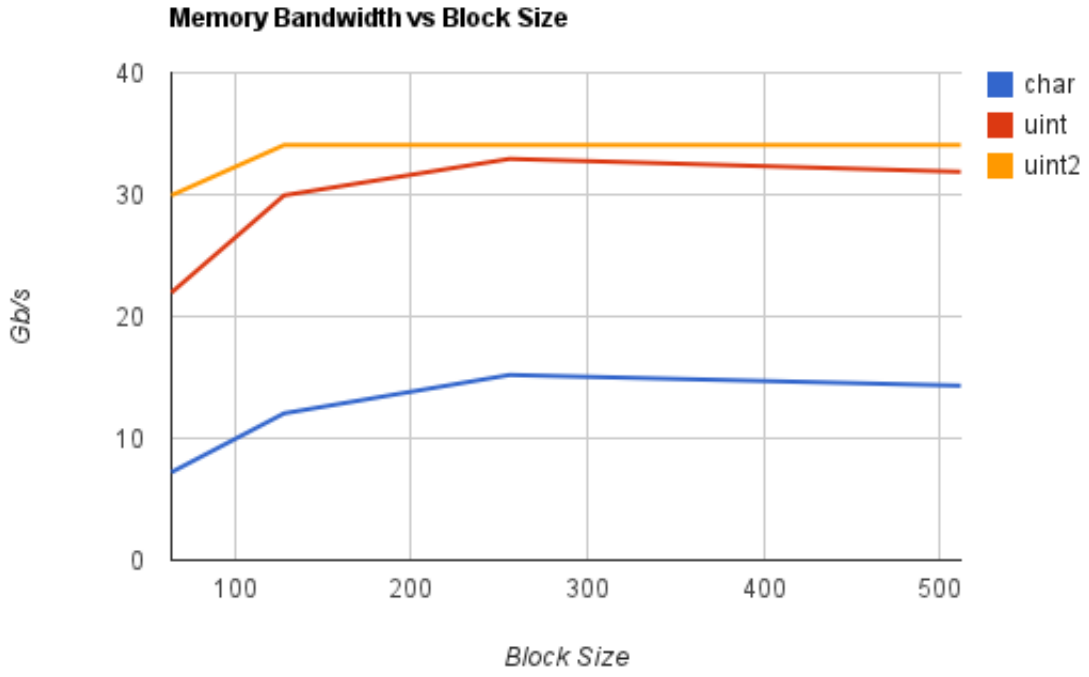
Bandwidth

Caesar

When we move from character to int, we are using an entire 32 bit word instead of 8 of the 32 bits in the word. We reduce the number of threads by a factor of 4, and are accessing four times less memory locations. This results in a speedup between 2x–3x. The reason we do not see 4x speedup is because we lose some caching effects between char and uint. When a memory request is made, 128 bytes are returned. A warp of 32 threads that each request 1 (consecutive) byte makes $\frac{32}{128} = 1/4$ request, but a warp of 32 threads that each request 4 consecutive bytes makes $\frac{32\,4}{128} = 1$ request. The $1/4$ request means that we waste almost $3/4$ of the bandwidth. Moving to uint2 does not have the same affect, as we are already using all of the bits in a word with int. We can see that without bit-packing the implementation plateaus as we immediately hit its maximum bandwidth, but the integer versions scales its bandwidth with the size of the input. Note that this graph is using bandwidth as defined as "amount of input and output data we reading or writing from/to memory per second." We can see that uint and uint2 are better than char, but worse than their theoretical maximum of 4x and 8x respectively.



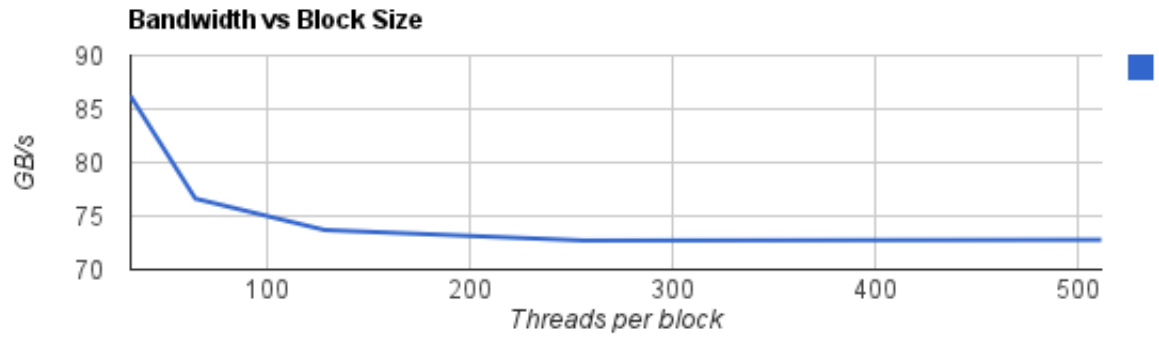
As a demonstration, for a fixed data input size (2^3 `moby_dict`), we compare the bandwidth against the block size. We can see that it does not play a major role, but there is a small improvement in having a larger block size. This is most likely the result of doing more of the computation with less parallel overhead. When the block size is 64, we can only have 8 blocks (the maximum) in each multi-processor (SM) so that the total number of threads is 512. Instead when we reach 256 threads per block, we are limited by the total number of threads per SM, 1536. In that case we only have 6 thread blocks for a total of exactly 1536 threads.

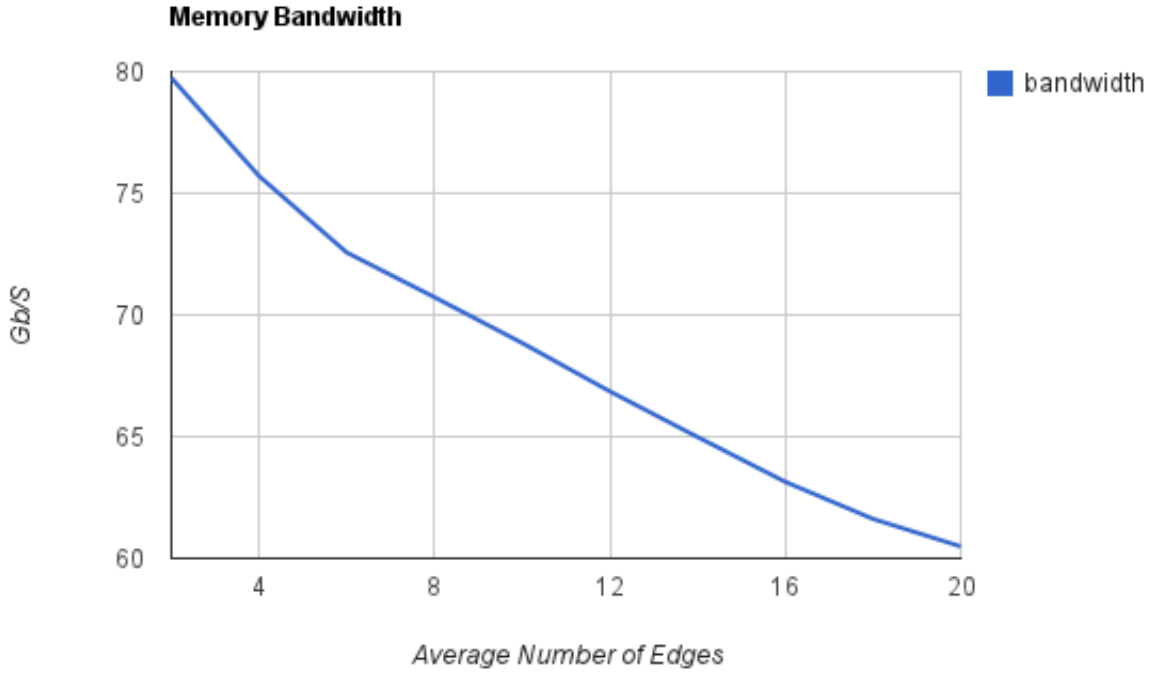


The Bandwidth *to* the device is greater than the copy *from* the device. To averaged about 50 GB/s, From averaged 35 Gb/s.

PageRank

Changing the block size should have no major effect as long as it is an integer multiple of a warp size. The change in performance might be due to the excessive number of memory requests when the number of threads becomes large. Since the memory access is random each memory request by a thread requires loading 128 bytes from memory. The large number of threads can therefore lead to a large number of requests. Random memory access is detrimental to the performance of the hardware and the processor seems to have difficulty handling a large volume of requests at random locations.





We see greater bandwidth when there are less average edges. When we increase the average number of edges, there is greater variance in the number of accesses per thread. This results in more divergence in the threads. As we will discuss in class, a warp is controlled by a single context, and divergence causes unnecessary work to be computed. In the pagerank example, checking the statement of the for loop is a *conditional branch*. Our graph creation scheme does not create an even partition of the edges to the nodes, instead it assigns them in increasing quantities. For example, if the average number of edges is 2.5:

node	edges
0	1
1	2
2	3
3	4
4	1
5	2
6	3
7	4

Because there are 32 threads in one warp, we will have multiple threads under the same control who have a different number of iterations to complete when accessing the edges of a node. This divergence causes us to under-utilize our cores. When the average number of edges is 2, 32 threads have at least one edge, 24 at least two, 16 three and 8 four. Then, because there is only one context of control, threads doing useful work will iterate as: 32, 24, 16, 8, for each warp we create. The wasted time is then $(32-24)+(32-16)+(32-8)=48$. If the number of average edges was instead 8, then similarly we would iterate as 32, 28, 24, 20, 16, 14, 12, 8, 4. Here the wasted work is 144. This leads to a reduced *bandwidth* as we are idling more of our cores.