Boston University College of Engineering

A CUDA Based Bitcoin Miner

EC527 Final Project

Gerardo Ravago

EC527 - High Performance Computing

Prof. Herbordt

May 8, 2015

Abstract

Bitcoin mining is a computationally intensive application that currently consumes 4,270 exaflops of the world's computing power. The goal of this project was to understand and optimize the hashcash proof of work algorithm used in mining to secure the Bitcoin network. Despite being computationally difficult, mining computations can be made efficient by exploiting the massive parallelism available on GPUs. Using Nvidia's CUDA platform a speedup of over 100 times was achieved over a naïve serial CPU implementation. In understanding the characteristics of the proof of work algorithm a parallel implementation was devised to effectively exploit the hardware on a GPU to achieve such dramatic results over the CPU.

Problem Statement

Bitcoin is a currency whose value and scarcity is guaranteed using digital cryptography. Through a distributed network of peers, the ownership of Bitcoin on the network is tracked in a decentralized manner which is truly revolutionary in the modern financial system of central banks. In order to make Bitcoin a secure and stable network three components must exist: transactions, the block chain, and proof of work mining. The first are digitally signed transactions using ECDSA public/private key pairs. Bitcoin are assigned to specific public keys and value is transferred by signing a transaction with the private key. Every 10 minutes a block containing all the transactions on the network is appended to the block chain which is stored on each Bitcoin node. With the entire block chain, one has a record of each and every transaction that has ever occurred on the network which can be used to verify ownership of Bitcoin through a recursive algorithm. Finally, mining is how blocks are added to the block chain in a manner that ensures the security of the network. It consists of a proof of work algorithm that is difficult

to solve but trivial to verify and strict rules enforced by cryptography that make it economically unfeasible to try and fool the network.

Mining is at the core of the Bitcoin's security which gives miners the most power on Bitcoin's decentralized network. In exchange for providing security for the network, miners may collect fees on transactions and a block reward for each block solved. Needless to say, it is essential that the rules of mining incentivize good behavior amongst miners. In order for a miner to add a block to the block chain, they must find a cryptographic nonce such that when added to the block header its SHA-256 double hash has a value less than the current difficulty. The SHA-256 double hash has the property that a small change in the input results in a large change in the output and that given an output it's difficult to find a corresponding input. Each block's block header then encodes all the transactions and the hash of the previous block which means that the miner cannot change any of these parameters without having to search for a new nonce. Brute force of the 2^32 nonce values is the only means of solving the block so it's in the miner's best interest to solve a valid block than to try and solve a fake one before the network's 4,270 zettaflops.

The Serial Algorithm

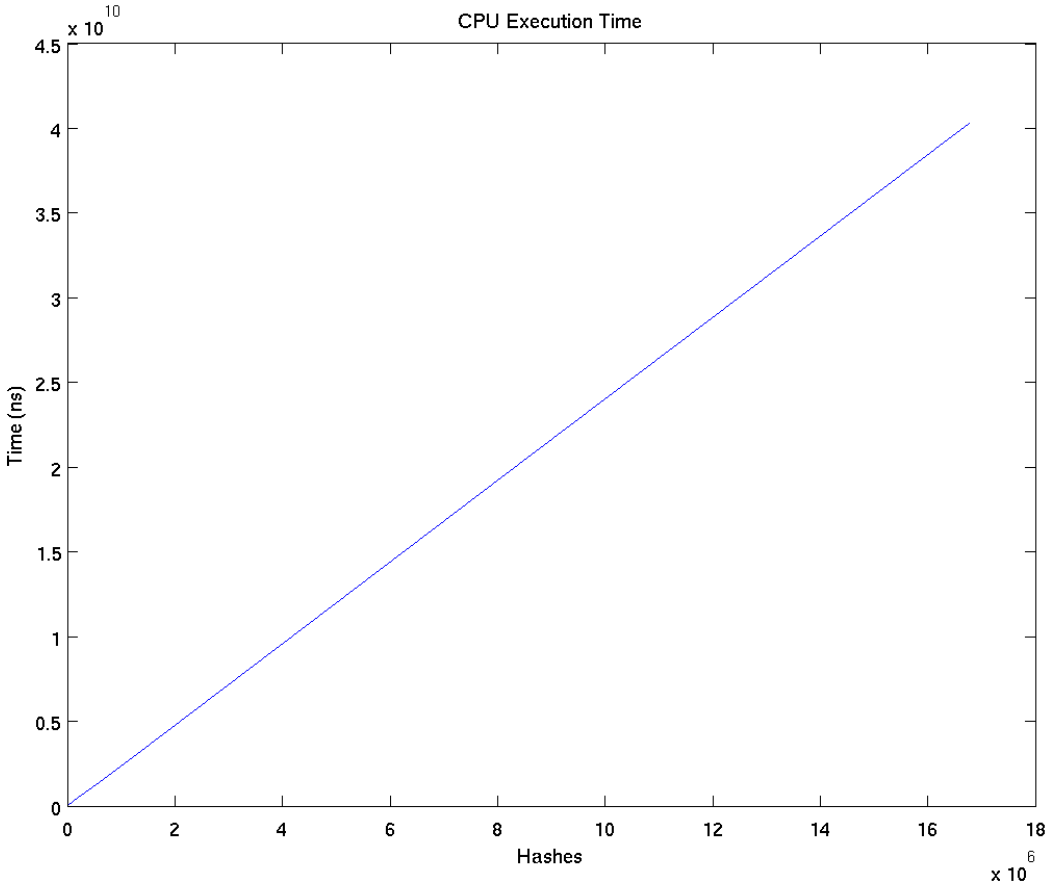
Bitcoin mining follows the simple pattern on the right of fetching a block to mine then within a tight loop the miner must compute the SHA-256 double hash for each possible nonce value and compare the results to the current difficulty level. If the block has not been solved by the network and a solution was not found, the block's timestamp and

```
while true
	if not block.isValid()
		block = getNewBlock()
	else
		block.updateTimestamp()
	for nonce = 0 to 2^32
		block.nonce = nonce
		hash = sha256(block)
		hash = sha256(nonce)
		if(hash < difficulty)
			submitBlock(block)
			block.setValid(false)
			break
	end
end
```

merkle root can be updated to mine additional nonce values. The compute time of the serial algorithm is dominated by the two SHA-256 computations that must occur for each nonce value.

A serial pseudo code version of the SHA-256 algorithm is available from the original NSA specification which can readily be transformed into working C code. The calculation starts by taking the input message and padding it to a multiple of 512-bits. Since the input message in this case is the block header, it has a fixed length of 80-bytes differing only in the 4-byte nonce value which is replaced on each iteration. For each block, SHA-256 expands the 512-bit block into a 64 word message schedule. The SHA-256 compression function is then applied to each word in the message schedule which compresses the message in a nonlinear fashion using integer arithmetic consisting of XORs, ANDs, ORs, NOTs, RORs, SHRs, and ADDs. The intermediate output hash of the compression function is then used as an initialization vector for the next block. The output of the final compression function is the SHA-256 hash.

The important characteristic to note is that this problem is highly parallelizable in that the SHA-256d computation that must occur for each nonce value is completely independent of each other. It's also very much computationally bound with the problem fitting comfortably in the L1 cache because of the small fixed message lengths. All the math is simple integer arithmetic in a tight loop. These characteristics make the GPU an ideal candidate for mining. A naïve implementation on an Intel Core i5 Haswell processor achieved a hash rate of 418.35KH/s on a Haswell i5 processor. The results of the experiment are shown on the next page and highlight how the problem scales linearly. No further optimization was performed because any comparison to a naïve GPU version would be unfair from the start.



Parallel Modifications

For this project, the target hardware was the Nvidia Quadro NVS 295 found on the lab machines which supports the CUDA platform. Adapting the serial algorithm to a GPU based parallel implementation is fairly straightforward. The main idea is to perform the serial computation of the SHA-256d algorithm on many cores on many different nonce values at the same time. Start by obtaining a block header to mine on the host and transferring this to the device's global memory. After that, instantiate 2^32 threads mapped linearly onto blocks. Each of these threads are responsible for computing the SHA-256d for a nonce corresponding to its thread ID. It then checks its hash against the difficulty, if it solved the block signal this to the host and transfer over just the corresponding nonce value. As far as data transfer between host

and device is concerned, only 80 bytes need to be transferred to the device and 8 bytes back to the host which is a negligible amount of time spent on memory overhead while the benefits of massively parallel computation are so great.

Optimizations and Experiments

In order to simplify comparison of each successive optimization, the hash rate metric is used heavily which is a measure of the number of SHA-256d computations that a particular algorithm can perform. The measurements below compute the execution of a kernel executing 65,535 blocks of varying thread dimensions. However, the experiment above shows that this problem scales linearly so as long as the hash rate is used the comparison is fair.

Gpuminer1 – First working implementation

The first implementation focused on porting the SHA256d computation to CUDA and verifying every step of the computation to have a working platform to start with. It did some preprocessing of the block header on the host side to precompute the first compression function of the first SHA-256 hash and padded the block header which is an operation that's uniform across all threads. On the device side, it took advantage of the knowledge that the input to the second hash computation was the output of the first which reduced the operations dedicated to transitioning between the two. This implementation was able to accomplish a hash rate of 26.28Mh/s which is already 62x speedup over the CPU implementation and strongly suggests that the GPU implementation is definitely a step in the right direction.

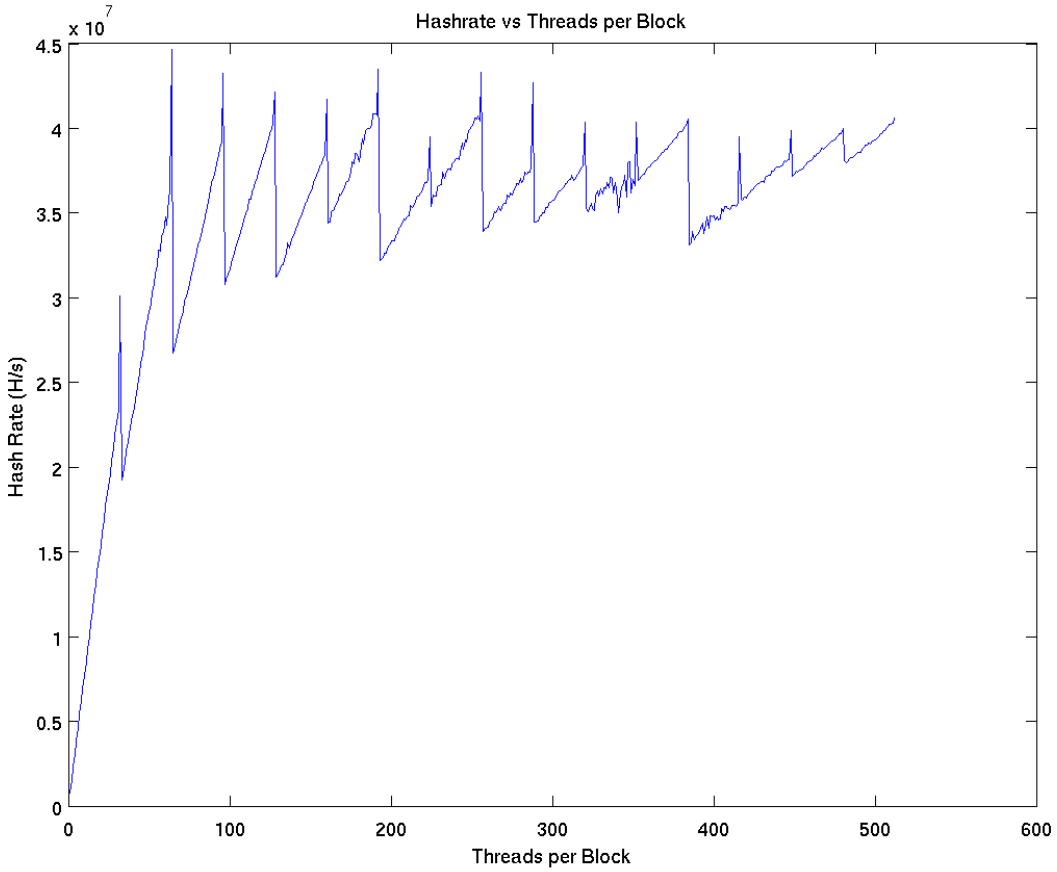
Gpuminer2 – Use Constant Memory for SHA-256 Constants

SHA-256 makes use of a 64 word constant array k that is derived from the fractional portions of the first 64 primes. A word from k is referenced with stride-1 in each round of the SHA-256 compression function which is computed twice for each thread. Previously, these

constants were stored in local memory as an array unique to each thread which lead to many unnecessarily long memory accessed. The switch to constant memory to store k resulted in the single largest optimization for the GPU algorithm shooting the hash rate up to 40.55 Mh/s, a 1.55x improvement. After this optimization the gains have only been incremental.

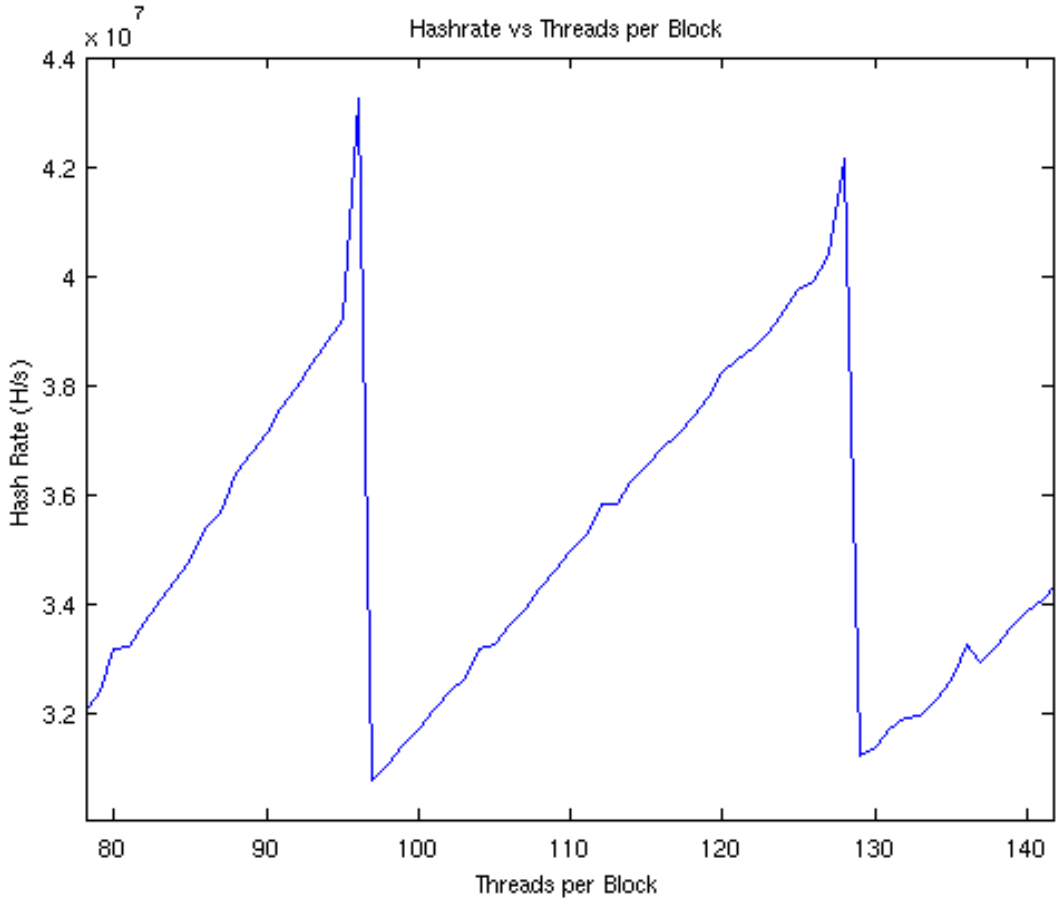
Gpuminer3 – Optimized Threads per Block

Up to this point, a block dimension of 512 threads was chosen to arbitrarily to maximize the threads per block. In hindsight, it wasn't too bad of a choice to begin with. However, due to the liberties taken by the CUDA platform in mapping blocks to SMs and threads to warps this parameter needed further experimentation. Since the GPU could very quickly churn through the 2^16 blocks of a single grid row, it made sense to check the performance of each possible block



size for threads 1-512. Below is the plot for this experiment and illustrates an interesting periodic behavior relationship between performance and block size

A closer look at this periodicity reveals the trend where performance peaks whenever the number of threads is a multiple of 32 and troughs immediately after that. This is because this matches exactly with the number of threads per WARP and utilization improves linearly up to that point and drops off sharply. The sharp peak between block size 31 and 32 is likely due to increased efficiency of memory accesses by complete WARPs. Looking at just the peaks, performance is maximized at 44.87Mh/s when the number of threads is 64 with two full WARPs per block. It also likely results in a favorable mapping of blocks to SMs.



Gpuminer4 – Code Motion

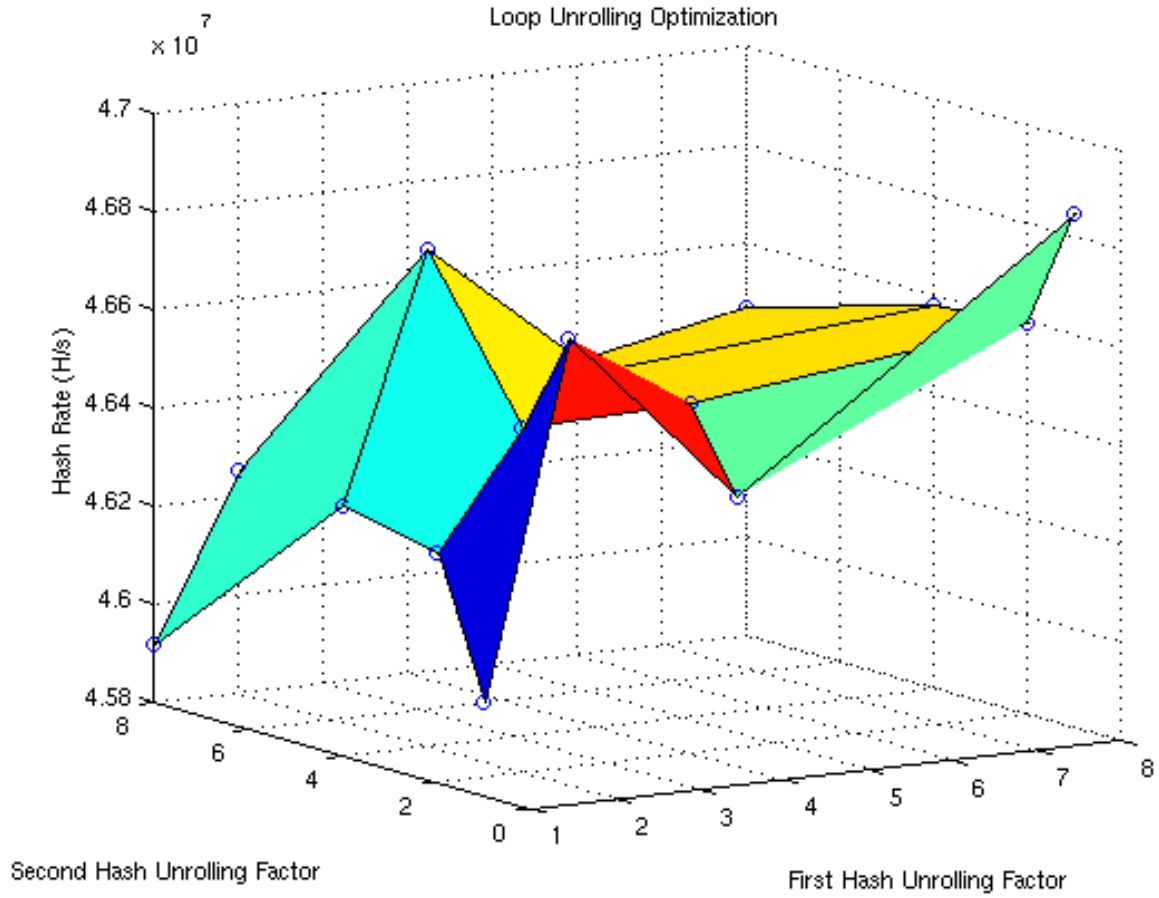
In gpuminer1 a major optimization was made that wasn't made for the serial implementation. This was to move a redundant computation out of kernel and let the device start calculating from the second compression round. This could go further by rearranging the word data on the host side to be little endian so a pointer on the device side could dereference it natively into the proper endianness. In doing so, the redundant operations from converting endianness pushed the hash rate up to 46.04Mh/s.

Gpuminer5 – Loop Unrolling

Within each thread there are two types of loops that occur once for each of the two SHA-256 computations. The first is the message schedule loop which expands the message block into an array W of 64 words. It does so by combining previous words in the following fashion.

$$W_j \leftarrow \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16}$$

From experimentation, unrolling this loop did not result in improved performance likely due to the irregular memory access patterns and dependencies arising from the W_{j-2} term. The second loop is the SHA-256 compression function which did result in a performance boost due to the many arithmetic operations and data paths that expose opportunities ILP from WARP scheduling. An interesting observation though was that the optimal performance boost depended on both the unrolling factor and the specific compression function that was unrolled. Trying all permutations of these dimensions resulted in the 3D plot below. The cliff on the left illustrates the necessity of unrolling the first loop by at least two and that the second loop is not nearly as important. The best performance was consistently found when the second loop is unrolled by a factor of 8 and the second loop is not. Without further visibility into the device's internals, this behavior is difficult to explain other than it resulted in boosting the hash rate to 46.92Mh/s.



Gpuminer6 – Coalesced Memory Access and Shared Memory

Unlike the previous optimizations this turned out to result in worse performance. Looking at the structure that needs to be transferred to the device only data, initial hash values, and the difficulty are actually important. This data all adds up to exactly 128 bytes and there are 64 threads to each block. Since this was all data to be shared by all threads, it made sense to copy it into shared memory. With 64 threads and 128 bytes of aligned data each of the 4 half warps could copy all the data into shared memory with 2 complete bus transfers from global memory each. The result was a drop in the hash rate down to 46.64Mh/s. Even with such a favorable memory access pattern the overhead of copying the data to shared memory and synchronizing

the threads was not worth it since the values being copied had a useful life of about one or two accesses and was not a particularly large amount of memory to begin with.

Gpuminer7 – Avoiding Local Memory

The one major optimization that would have provided a major improvement in performance was to avoid using local memory for the message schedule. The reason for this was more the result of a flaw in the language's semantics than a technical limitation imposed by hardware. There's no easy way to declare an array of data private to the thread that isn't stored in local memory. The alternative would be to either create 64 registers which don't have the semantic convenience of arrays, with the unrolled loops and nature of unrolled loops this would make the code unreasonably difficult to work with. Allocating a large chunk of shared memory and allocating pieces to individual threads did not work either. The result was a very slowly executing kernel call that was likely due to the poor mapping of blocks to SMs.

Results

Program	Description	Hash Rate H/s	Improvement
cpuminer	Serial Algorithm on CPU Haswell Core i5 4200U	418,350.00	N/A
phoenix	OpenCL based miner that runs on NVS 295	1,700,000.00	4.06
ufasoft	Best CPU miner running on Core i5 2500k	20,600,000.00	49.24
gpuminer1	**First GPU implementation, minimal optimization**	**26,279,590.21**	**62.82**
gpuminer2	**Constant memory for array k**	**40,554,607.06**	**96.94**
gpuminer3	**Optimal threads per block**	**44,873,970.71**	**107.26**
gpuminer4	**Code motion**	**46,042,351.40**	**110.06**
gpuminer6	**Coalesced memory acess and shared memory**	**46,635,748.39**	**111.48**
gpuminer5	**Loop unrolling**	**46,923,079.87**	**112.16**
rpcminer-cuda	Best Nvidia miner running on a GTX570 (60x more SPs)	165,000,000.00	394.41
HashCoins Zeus	The fastest ASIC based Bitcoin miner	4,500,000,000,000.00	10,756,543.56

The table above compares the results of various optimizations made during this project to the naïve serial implementation as well as to hash rates reported by the Bitcoin community for bench marking purposes. Switching to the GPU resulted in at least a doubling in performance when compared to the best CPU implementation meaning the extra hardware was put to good

use. Further, this CUDA based miner beat the only user reported hash rate for the NVS 295 by a wide margin but it is not a fair comparison because the latter was running an unoptimized OpenCL implementation on Nvidia hardware. It's also worth mentioning that like the CPU miners before them, GPU miners have long gone out of style when compared to the fastest ASIC mining servers on the market. With their high acquisition costs and electricity usage, GPU mining is unprofitable outside of situations where the user does not have to pay for either such as the case of using university lab machines. Running this project's best gpuminer implementation on all 25 PHO 207 lab machines will return about $0.08 per month if ran 24/7.

Discussion

For this project, the memory bandwidth from host to device was incredibly negligible as was evidenced by the reduction in performance by optimizing for memory. The project was computationally bound on the GPU with the goal of maximizing utilization on the Quadro NVS 295's 8 CUDA cores. To do this, the best approaches were the ones that increased utilization of the hardware and allowed more instructions to be executed in parallel on the GPU. This meant making sure there was always a WARP ready to be scheduled and that WARP was always full of threads. It also meant moving as much redundant calculation off the GPU as possible so it could focus on the independent portion of the calculation. Despite being compute bound, a fundamental understanding of memory on the GPU was key to getting acceptable performance by exploiting the constant memory. Further, rearranging the bytes to fit the native endianness saved a surprisingly large amount of time. Overall, this implementation is close to the limit that the lab hardware can provide and thanks to the CUDA architecture it could very well scale invisibly onto better and more capable hardware to further improve the hash rate.

References

Bitcoin Developer Reference. (n.d.). Retrieved May 7, 2015, from https://bitcoin.org/en/developer-reference

Dashjr, L. (2012, February 28). Getblocktemplate - Pooled Mining. Retrieved May 7, 2015, from https://en.bitcoin.it/wiki/BIP_0023

Guilford, J., Yap, K., & Gopal, V. (2012, May 1). Fast SHA-256 Implementations on Intel Architecture Processors. Retrieved May 7, 2015.

Cpuminer. (n.d.). Retrieved May 7, 2015, from https://github.com/pooler/cpuminer

Crypto-algorithms. (n.d.). Retrieved May 7, 2015, from https://github.com/B-Con/crypto-Algorithms

Libblkmaker. (n.d.). Retrieved May 7, 2015, from https://github.com/bitcoin/libblkmaker

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.*Consulted, 1*(2012), 28.

PUB, F. (2012). Secure Hash Standard (SHS).