Optimizations to NFS LA

Patrick Stach

NFS Linear Algebra

- Solve for a vector x such that:
  - x != 0 and B*x = 0
- One of, if not the worst, scaling steps of NFS
  - RSA-640 (193 digits) to RSA-200
    - 1.5 months to 3 months increase
  - SNFS-1024
    - 66m x 66m, 9.5b non-zeros, 59 days on 110 Prescotts

Covered Topics

- Comparing Block Wiedemann and Block Lanczos
- Optimizing expensive operations in these algorithms on x86-64 and nVidia GPUs
- Optimizing distributed computation when dealing with large matrices
- One note: All the original code is in assembler, which translates very poorly to slides, the C code present is not tested thoroughly

Wiedemann vs Lanczos

Block Wiedemann

- Krylov sequence generation
  - **(N / m) + (N / n) + 1** matrix vector products
- Berlekamp-Massey
  - **(N / m) + (N / n) + 1** matrix polynomial multiplications
- Polynomial Evaluation
  - **(N / m) + (N / n) + 1** matrix vector products
  - **(N / m) + (N / n) + 1** n x n times n x N
  - **(N / m) + (N / n) + 1** vector additions
  - **2** vector “maskings” (masking on and off of bits)

Block Lanczos

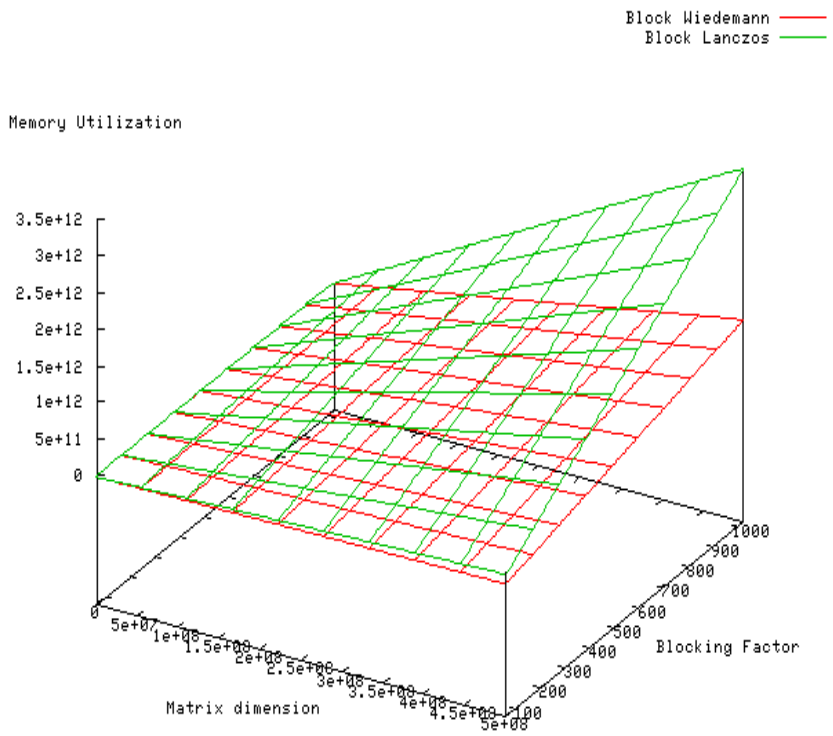
- Runtime not formally proven (thus all numbers are not worst case as in last slide)
  - Approximation of **N/(n – 0.73)**
- **N/(n – 0.73)** matrix vector products
- **N/(n – 0.73)** transpose matrix vector products
- **2N/(n – 0.73) + 4** inner products
- **4N/(n – 0.73)** vector and n x n matrix products

Cost of Matrix Vector Ops

- BW only requires matrix vector products
- BL requires both matrix vector and trans(matrix) vector products
  - One operation produces random writes unless matrix is stored in both row & column major formats
  - Random writes increase exponentially as **N** increases, however graph partitioning algorithms can be used in the filtering stage to partially reduce the overhead of constant cache invalidation
    - Msieve 1.37 rsa110 (298k x 298k) vs rsa120 (578k x 578k)
      - 4.59x increase in abs(time(matvec) - time(trans(mat)vec))
    - Could be calculated for a given polynomial, LP bound, sieving range, matrix size, and partitioning algorithm

Cost of Memories

- BW requires matrix plus two n x N bit vectors
- BL requires matrix plus four to six n x N vectors
- 300m x 300m matrix with 512 bit blocking - just vectors
  - BW = 35.76GB
  - BL = 107.29GB



Basic Operations

Matrix Organization

- Stored row major
- Sorted by row weight
- Dense rows
  - 4 bytes per 32 rows times number of columns
  - Rows are considered dense if sum of 32 consecutive rows > number of columns
- Sparse rows
  - Stored in column offset format

Naive Dense Operation

- Accounts for 10% to 15% of computation time of a matrix vector product
- **N(32 + n)** uncached linear bits to be read per 32 dense rows multiplied

```
void mul_dense_32xN_Nx32(uint32 *b, uint32 *A, uint32 *x, uint32 N) {
        uint32 i, j, rA, rx[2];
        rx[0] = 0;
        for(i = 0; i < N; i++) {
                rx[1] = *(x++);
                rA = *(A++);
                for(j = 0; j < 32; j++) {
                        rb[j] ^= rx[rA & 1];
                        rA >>= 1;
                }
        }
}
```

Naive Sparse Operation

- Accounts for 85% to 90% of computation time of a matrix vector product
- **offset_size * row_weight** uncached linear bits to be read
- **row_weight * n** random reads of **n** bits each

```
void mul_sparse_1xN_Nx32(uint32 *b, uint32 *A, uint32 *x, uint32 rlen) {
        uint32 i, rb;
        for(rb = 0, i = 0; i < rlen; i++)
                rb ^= x[*(A++)];
        *b = rb;
}
```

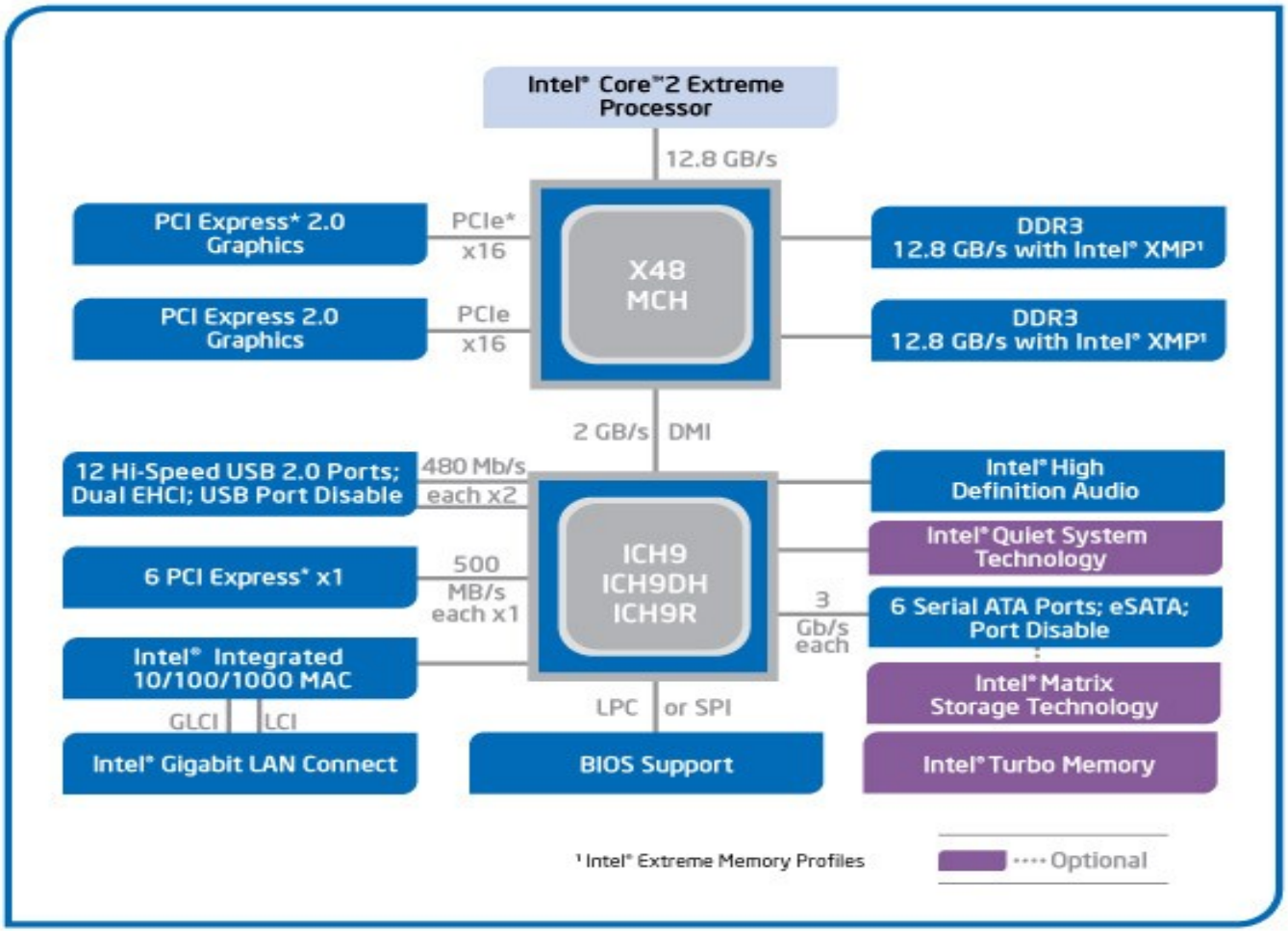
Other Operations

- Inner product
  - **2Nn** bits of linear read
- n x n times n x N
  - **Nn** bits of linear read
  - **Nn** bits of linear write
- n x N times n x N
  - **Nn** bits of linear read
  - **Nn** bits of linear read/write
- n x N masking
  - **Nn** bits of linear read/write

- Performance increases and methodologies demonstrated with dense operations seems to hold true for these “other operations”

Modern x86 Hardware

System Diagram



Intel® X48 Express Chipset Block Diagram

From MCH to CPU
Intel vs AMD

- Intel interfaces its memory controller via the FrontSide Bus (FSB), 128 bits data 1.66ghz+, 1/10th FSB speed address bus
- Supports DDR3

- AMD interfaces its memory controller via the Hypertransport Bus, 16 bits data, 3.2ghz+, dual data rate
- Supports DDR2

Multi-Core Caching

- Set CPU affinity to avoid OS instigated invalidates
- Cache coherency
  - Cache line corresponding to an address can exist in only one cache – Avoid this as it causes stalls and wastes cache
  - L1 is split, L2 is shared to two cores on Intel and split on AMD
- Avoiding thrashing the same cache set with respect to the associativity model of the cache
- Stride between threads must be large enough to avoid line ownership conflicts, but close enough to provide read and write coalescing due to shared per physical package memory bus

Dense Op. Changes

- Prefetch data to be used in (current step + offset) step from matrix and X vector set
- Interleave each core by L1 cache line size pieces of work
  - With 32 bit dense entries and 64 byte L1 line size, each thread starts at an offset of 16 entries from each other
- Group dense rows by 128, 64, or 32 entries as possible to reduce number of loads and swizzle operations on SSE registers
- Padding with all zero entries on dense row data and X may be necessary due to other changes to avoid segmentation faults or incorrect data

Opt. Dense Operation

```
#define BLOCKSIZE (L1_DCACHE_LINESIZE / sizeof(uint32))
void mul_dense_32xN_Nx32(uint32 *b, uint32 *A, uint32 *x,
        uint32 N, uint32 throff, uint32 nthr)
{
        uint32 i, j, k, rA, rx[2], rb[32];
        rx[0] = 0;
        for(i = 0; i < 32; i++) rb[i] = 0;
        i = (throff * BLOCKSIZE); A += (throff * BLOCKSIZE);
        for(; i < N; i += (nthr * BLOCKSIZE)) {
                for(j = 0; j < BLOCKSIZE; j++) {
                        prefetcht0(A + (nthr * BLOCKSIZE * 16));
                        prefetcht0(x + (nthr * BLOCKSIZE * 16));
                        rA = *(A++);
                        rx[1] = *(x++);
                        for(k = 0; k < 32; k++, rA >>= 1)
                                rb[k] ^= rx[rA & 1];
                }
                A += (nthr - 1) * BLOCKSIZE;
                x += (nthr - 1) * BLOCKSIZE;
        }
        for(i = 0; i < 32; i++) atomic_xor(&b[i], rb[i]);
}
```

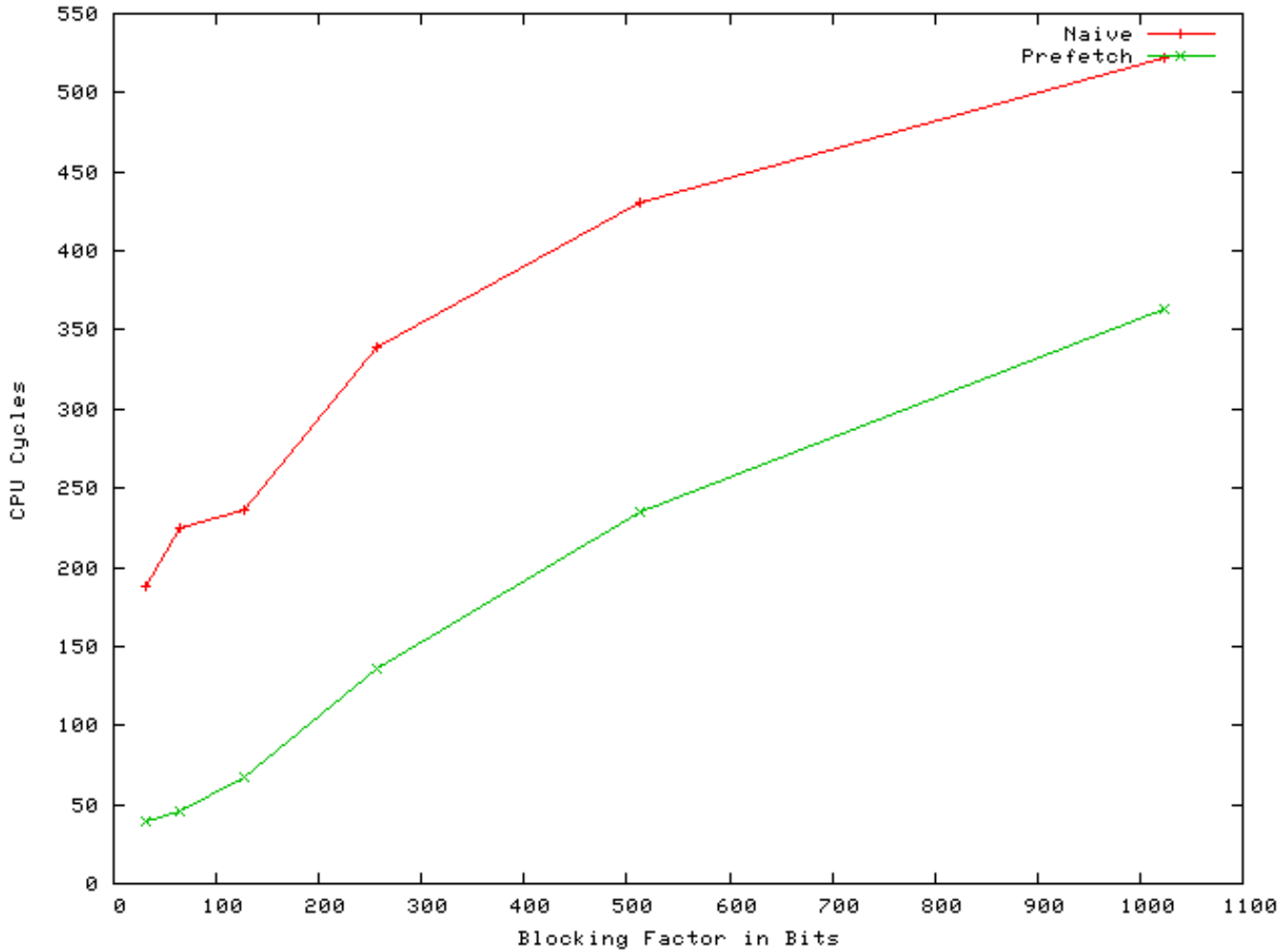
Sparse Op. Changes

- Prefetch matrix to be used in (**current step + offset1**)
- Prefetch X vector data to be used in (**current step + offset2**)
  - offset2 = ~(offset1 / 2)
- Interleave work by L1 line size similar to dense operations
- Group sparse rows into **Z** interleaved sets by length to take advantage of possible spatial locality in column offsets
  - A “good” Z value depends heavily on filtering and N
- Set padding column offsets to **N** and **X[N]** = 0
  - This will not affect calculations and leaves the inner loop branch free

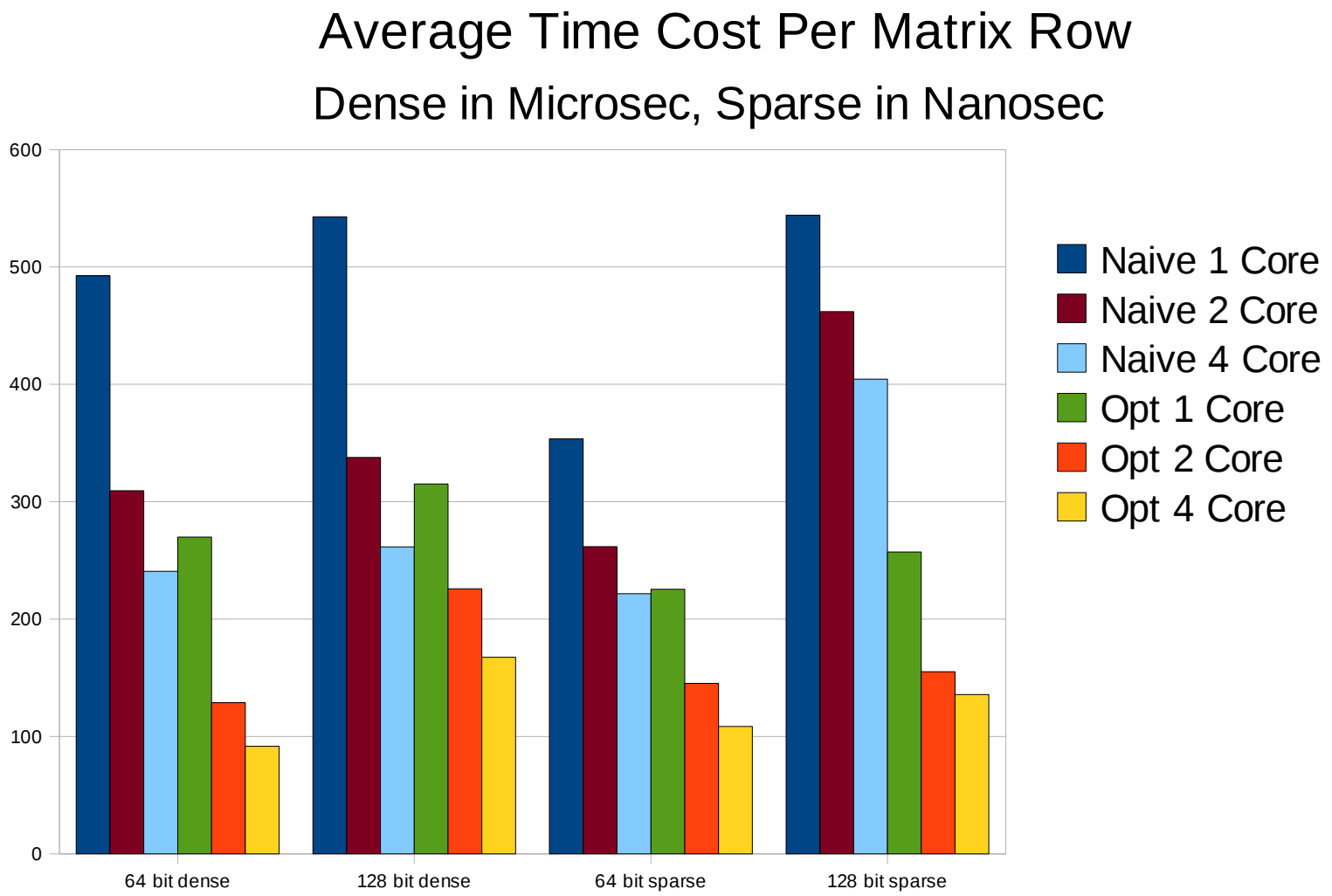
Opt. Sparse Operation

```
#define BLOCKSIZE (L1_DCACHE_LINESIZE / sizeof(uint32))
void mul_sparse_256xN_Nx32(uint32 *b, uint32 *A, uint32 *x,
        uint32 rlen, uint32 throff, uint32 nthr)
{
        uint32 i, j;
        i = (throff * BLOCKSIZE); A += (throff * BLOCKSIZE);
        for(; i < rlen; i += (nthr * BLOCKSIZE)) {
                for(j = 0; j < BLOCKSIZE; j++) {
                        prefetcht0(A + (nthr * BLOCKSIZE * 32));
                        prefetcht0(x + A[nthr * BLOCKSIZE * 16]);
                        b[(i + j) & 0xff] ^= x[*(A++)];
                }
                A += (nthr - 1) * BLOCKSIZE;
        }
}
```

Prefetch & Sparse Op.



Opt. Results – 1m^2 Matrix

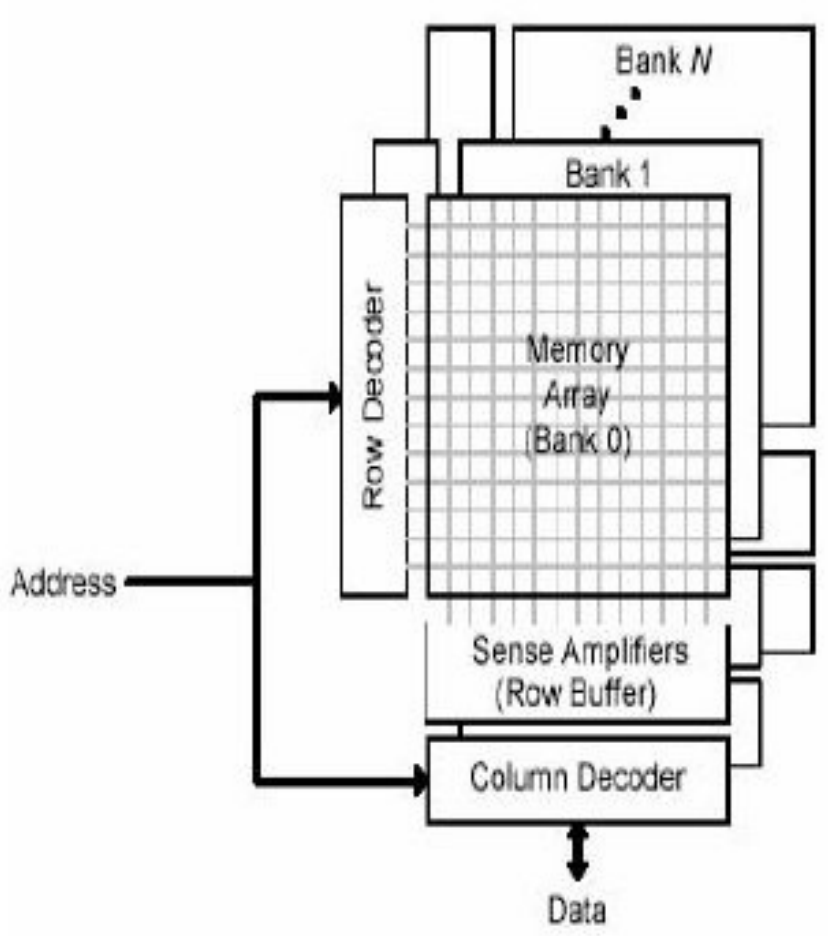


Memory DIMM to MCH

- 64 bit data bus per DIMM bank
  - Dual channel = 2 DIMM banks
- Most modern desktop x86 boards have 2 DIMM banks per CPU
- Throughput highly dependent
  - CPU and MCH access scheduling
  - Component timings with respect to their frequency

Memory Component Level

- Chips are addressed in a 3-D matrix like fashion by bank, row, column
- Timings described in notation of tCAS - tRCD - tRP - tRAS – tCR
  - tCAS = Column Address Strobe (CAS) Latency
  - tRCD = RAS to CAS delay
  - tRP = Row Precharge
  - **tRAS = RowAddress Strobe**
  - tCR = Command Rate
- Other constraints on memory components typically not advertised by DIMM vendors

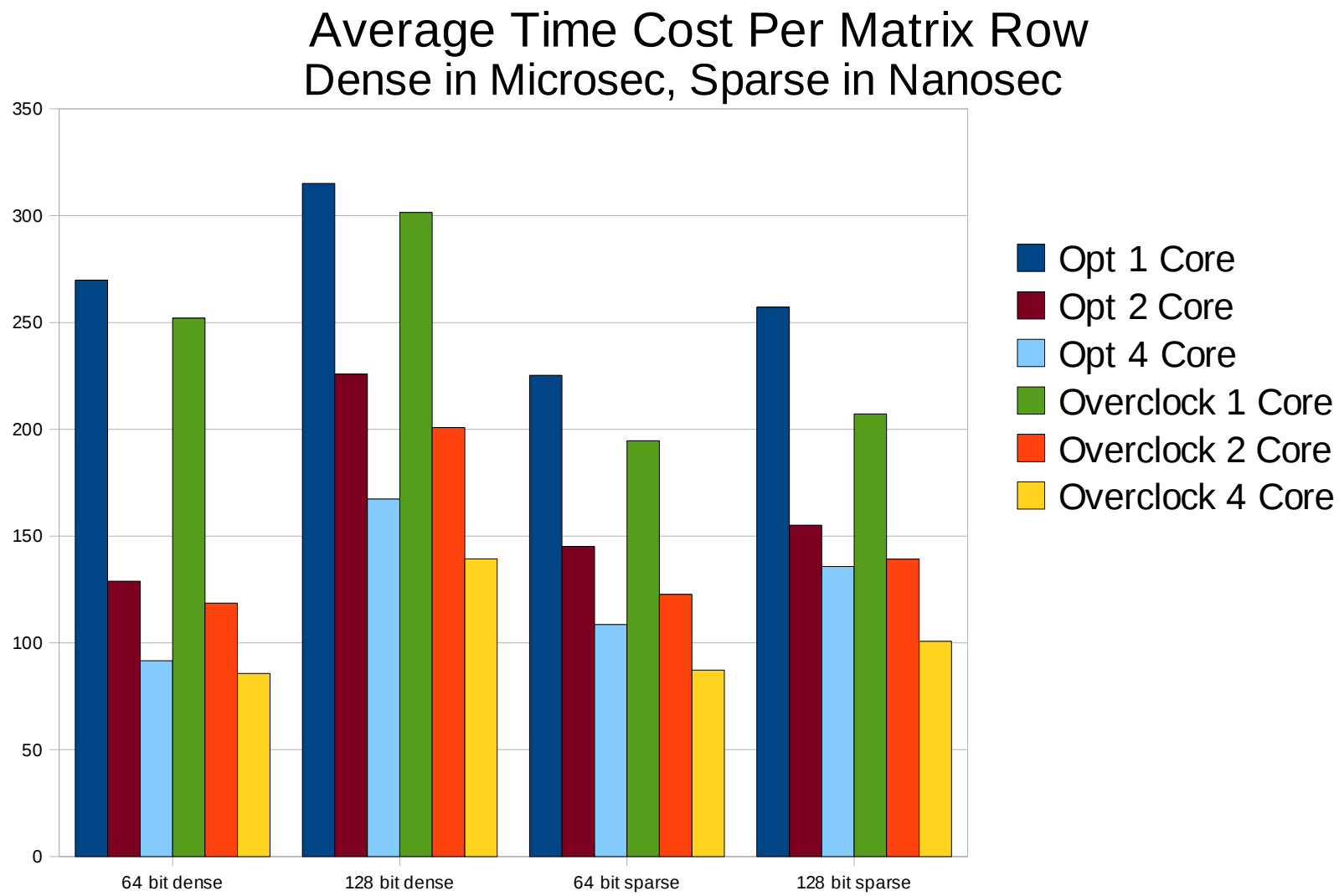


Overclocking Before & After

- FSB - 1333mhz
- CPU - 2.66ghz
- Memory – 1833mhz @1.95V
- Mfg Default Timings
  - 8-8-8-27-1
  - RAS delay 14.72ns
  - 4 clock read to read delay

- FSB - 1600mhz
- CPU – 2.60ghz
- Memory – 1800mhz @2.12V
- New Timings
  - 7-6-6-22-1
  - RAS delay 12.22ns
  - 3 clock read to read delay
- Required extra fan and two zip ties to pass Memtest

Overclocking – 1m^2 Matrix



nVidia CUDA

GPU Myths

- Difficult to program
  - Its just a different programming model, probably more akin to MasPar than x86
  - C and assembler interfaces
- Inaccurate
  - Its hard to get address generation and XOR wrong
  - IEEE 754 or better double precision FP operations
  - 64-bit integer, for the most part same speed as FP equivalents

NVidia GTX280 vs Intel Q9450

- 512 bit data bus
- 1GB GDDR3 (dual port) 2.2ghz memory
- 141GB/s memory bandwidth
- 16kb cache-like shared memory & 16kb texture cache per block
- 240x 1.3ghz cores

- 128 bit data bus
- 8GB DDR3 (single port) 1.833ghz
- 12.1GB/s memory bandwidth
- 12mb cache (2x 6mb)
- 4x 2.66ghz cores

GPU Execution

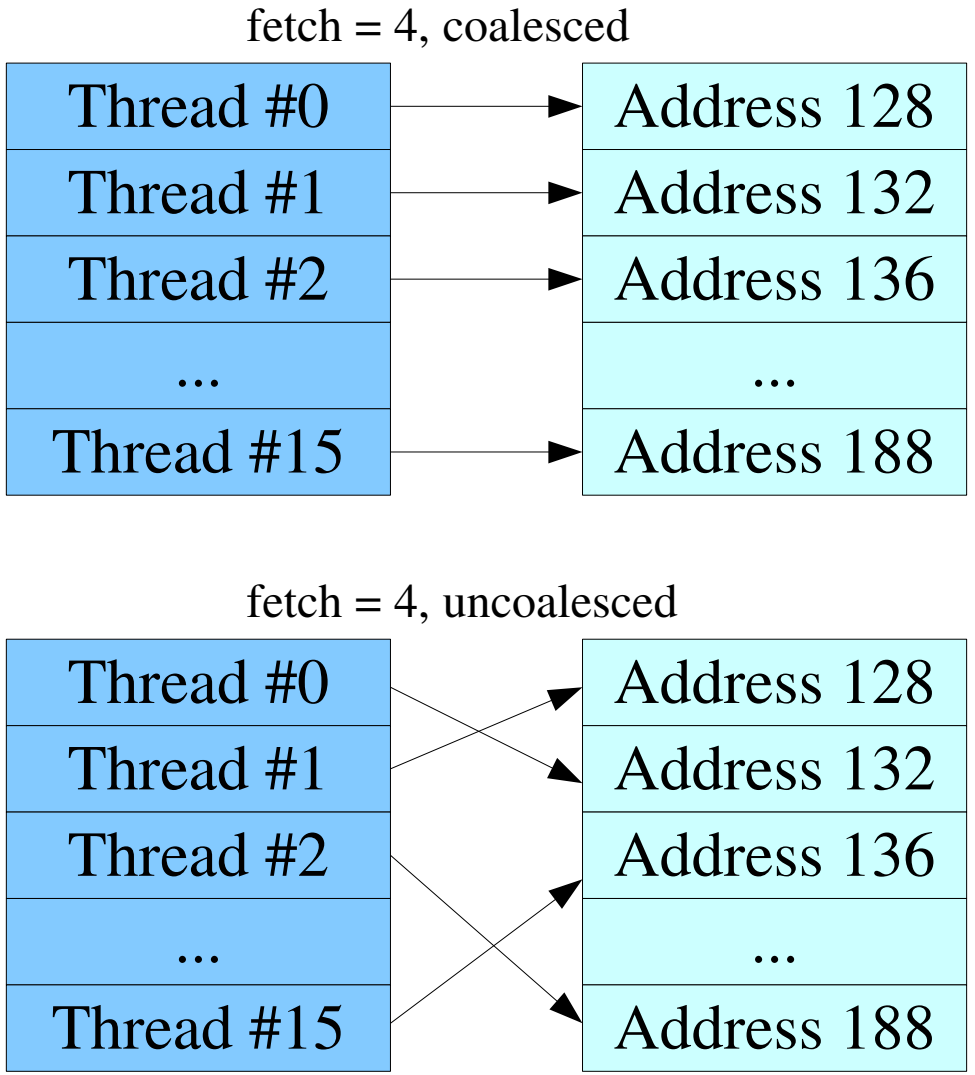
- Code, called kernels, are launched in 3-D groups of blocks
- Blocks are composed of 3-D groups of threads
- Blocks are scheduled on multi-processors in units of threads called warps
  - Warp size is a constant 32

GPU Execution (cont.)

- Groups of 16 threads called “half-warps” are scheduled once per clock cycle
- Half-warps share an instruction pipeline, but have independent registers
- If one thread in a half warp diverges by taking a branch either:
  - It remains idle until all other threads in half warp converge
  - Other threads remain idle until the divergent thread converges

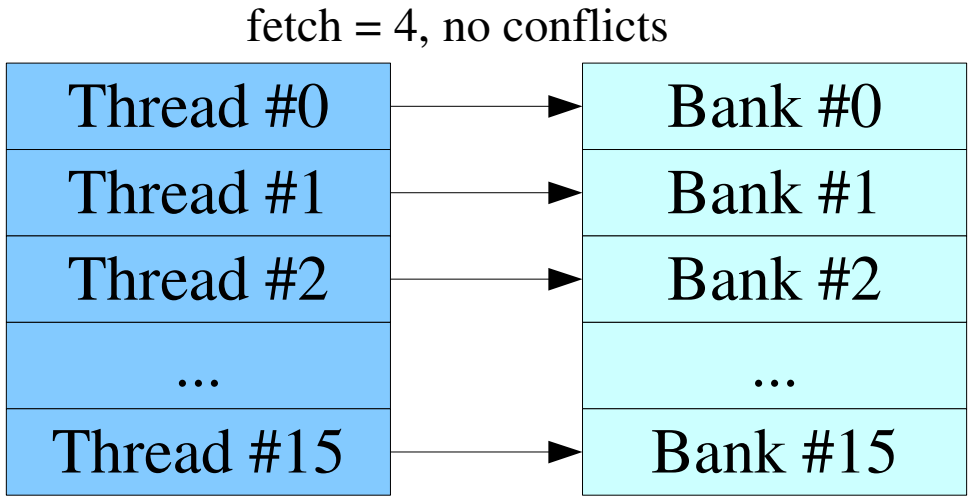
Global Memory

- Read/write coalescing requires:
  - Fetch size per thread to be 4, 8, or 16 bytes
  - Not all threads must participate
  - (thread #n * fetchsize) = addr (mod 16 * fetchsize)
- Not cached, requires 100 to 200 cycles



Shared Memory

- Shared per thread block
- Register speed, 16kb per thread block
- Divided into 16 banks
  - Address % 16 = bank #
  - Threads within a half-warp must maintain a 1:1 bank access mapping
  - If >= 2 threads read the same address in a bank, it is broadcast without penalty



Texture Memory

- 16kb cache
  - Cache hit = similar cost as shared memory access
  - Cache miss = same as global memory access
- Read-only
- Not subject to memory access pattern requirements
  - Perfect for the random read requirements of sparse operations

GPU Dense Operation

```
__global__ void mul_dense_32xN_Nx32_kernel(uint *b, uint *A, uint *x) {
        uint idx, i, j, rA, rx[2];
        __shared__ uint sh_b[512];
        idx = (blockIdx.x << 9) + threadIdx.x;
        sh_b[threadIdx.x] = 0;
        rA = A[idx];
        rx = { 0, x[idx] };
        j = threadIdx.x;
        for(i = 0, j = threadIdx.x & 0x1f; i < 32; i++, j = (j + 1) & 0x1f)
                sh_b[(threadIdx.x & ~0x1f) | j] ^= rx[(rA >> j) & 1];
        for(i = 256; i >= 32; i >>= 1) {
                if(threadIdx.x < i)
                        sh_b[threadIdx.x] ^= sh_b[threadIdx.x + i];
                __syncthreads();
        }
        if(threadIdx.x < 32)
                atomicXor(&b[threadIdx.x], sh_b[threadIdx.x]);
}
```

GPU Sparse Operation

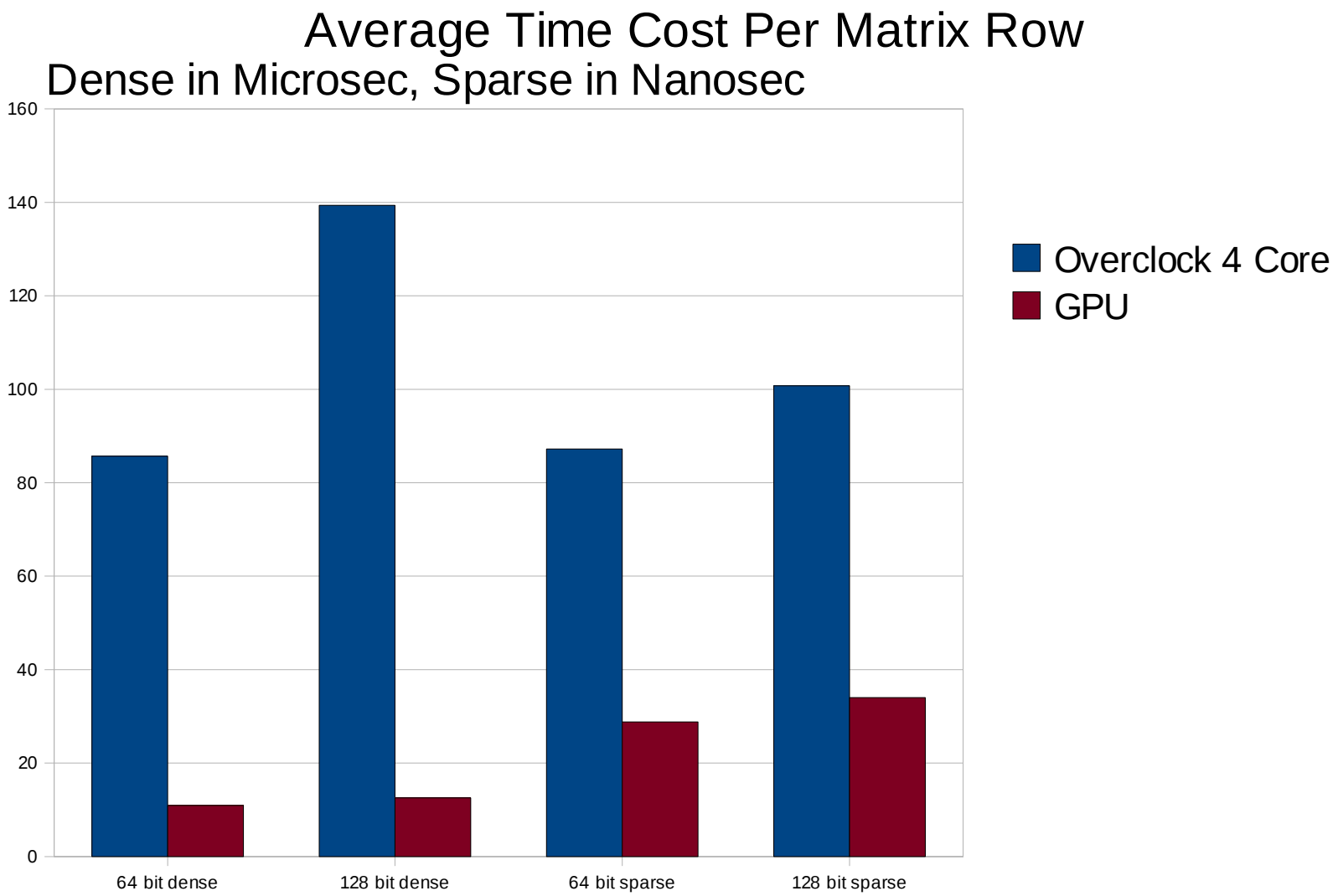
```
texture<uint, 1, cudaReadModeElementType> x32Tex;

__global__ void mul_sparse_256xN_Nx32_kernel(uint *b, uint *A, uint *Ahdr) {
        uint rb;
        uint2 hdr;
        __shared__ uint2 sh_hdr;

        if(threadIdx.x == 0)
                sh_hdr = make_uint2(Ahdr[blockIdx.x], Ahdr[blockIdx.x + 1]);
        __syncthreads();

        rb = 0;
        for(hdr = sh_hdr; hdr.x < hdr.y; hdr.x += 256)
                rb ^= tex1Dfetch(x32Tex, A[hdr.x + threadIdx.x]);
        b[(blockIdx.x << 8) + threadIdx.x] = rb;
}
```

Opt. Results – 1m^2 Matrix



Parallel Operations

Large Matrices

- Problems
  - 1GB on GPU is easily exhausted at ~c145
  - 8GB on desktops is slower but not enough storage
  - 256GB on quad CPU machines not enough bandwidth
    - 4x 128-bit data bus = 512-bit @~1ghz clock
    - Typically 8 DIMMs per DIMM bank @667mhz

Partitioning Schemes

- Mesh
  - N x N grid of sub-matrices
  - Partitioning algorithm execution time vs speed-up of matrix vector product diminishes as matrix size increases (Slide 7)
- Row
  - Each node consumes row sets from the matrix round robin ordered densest to lightest
  - Each node has approximately the same memory requirements and number of random accesses

Communication – N x N

- Mesh – Peer to Peer
  - Lower left node completes computation at cycle N^2
- Parallel Column Mesh
  - Controller transmits via a parallel bus that runs top to bottom along the column
  - Left column completes computation at cycle N
- Parallel Column, Switched Row
  - Each node has direct communication with reciever
  - Row receiver must accumulate results

Communication – By Row

- Most communication buses provide broadcast or multi-cast facilities
  - Ethernet, PCIe 2.0 AS
- Send entirety of current vector via broadcast
  - Same as cost of sending it out in pieces to multiple hosts
- Results require no reassembly other than maintaining ordering
  - Recv or memcpy to appropriate offset

Proposed Solution

- RSA-768 matrix speculations
  - 300m x 300m, 50b non-zeros
  - Assuming none of these are “dense”
    - 186.26GB for matrix with 32-bit offsets
    - 93.13GB for matrix with 16-bit offsets
  - If n is 512 and m is 1024, each temp storage
    - 4.47GB if 128-bits are computed at once
    - 2.24GB if 64-bits are computed at once
    - Each system requires 1 + 1/num_total_systems of these

Proposed Solution (cont.)

- Machine with cost ~25k Euro:
  - Asus P5E64 WS Evolution
  - 8GB DDR3
  - 4x nVidia Tesla S1070
  - PCIe 1x dual gigabit ethernet
- Four machine yield:
  - 256GB GDDR3
  - 4 gbps communication

P5E64 WS Evolution

Future Work

- Calculate timing estimates for RSA-768 sized matrix
- Rewrite Berlekamp-Massey to use sub-quadratic algorithms described by Thomé
- Roll x86-64 and CUDA implementations into a redistributable library and standalone solver
  - Currently waiting on vendor supported fixes to CUDA
    - Remove direct to driver hacks currently used and legally not distributable