Fast Feedforward Neural Networks with CUDA and OpenMP+SSE

Elliott Forney

May, 2010

Brief Neural Network Review

Status

Micro-benchmarks

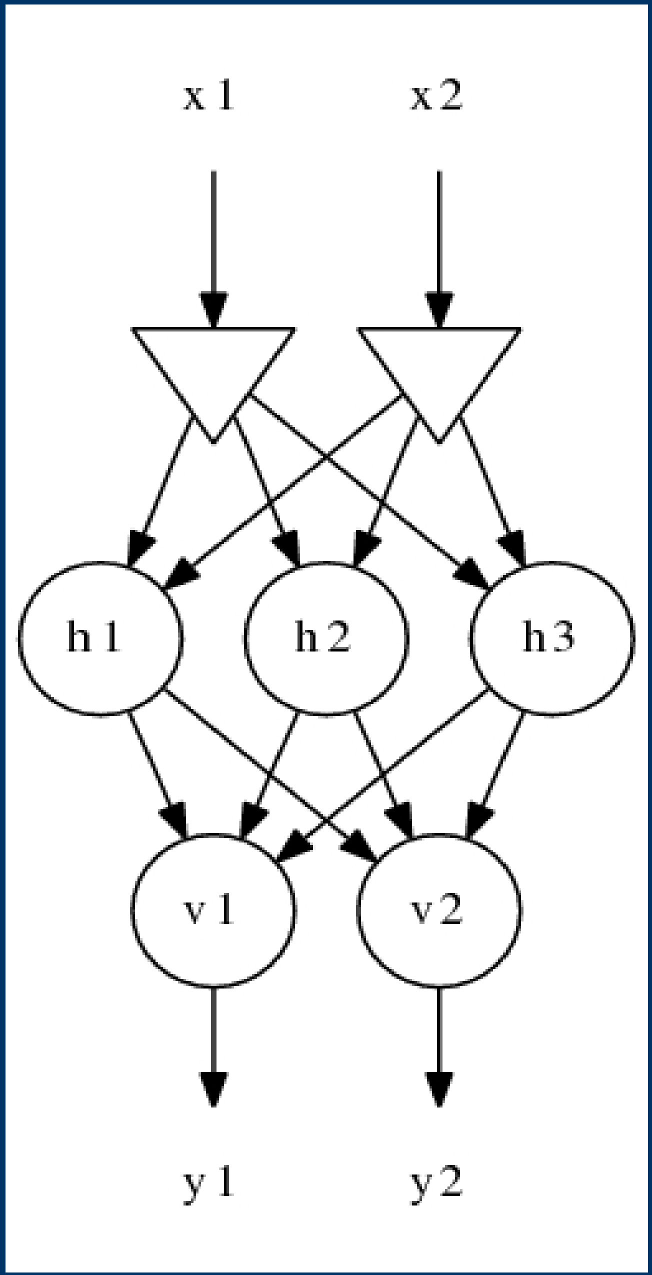
Macro-benchmarks

Conclusions

Future Improvements

Neural Network Review

- Neural networks are trainable function approximators
- Neural networks “learn” to map inputs x to outputs y by adjusting connection strengths between units h and v
- Two layer network is a universal function approximator, with potentially infinite number of hidden units
- If we collect our samples into a matrix then each training pass combines the errors from each sample: batch training
- Here, weights are updated using “Steepest” Gradient Descent



Neural Network Review

n : number of training samples

m : number of network inputs

p : number of hidden units

q : number of network outputs

X : input matrix m x n

Y : output matrix q x n

T : target matrix q x n

H : hidden weight matrix p x m

V : visible weight matrix q x p

l : learning rate parameter

ϵ : desired accuracy

$*$: component-wise multiplication

$tanh$: component-wise hyperbolic tangent

M^T : transpose of M

M^+ : add a row of 1's to M

M^- : remove last column of M

Forward pass:

$$Y = V(tanh(HX^+)^+)$$

Squared Error:

$$E = (Y - T)^2$$

Gradient for visible layer:

$$\nabla_V E = \nabla_Y E \cdot \nabla_V Y = 2(Y - T)(tanh(HX^+)^+)^T$$

Gradient for hidden layer:

$$\nabla_H E = \nabla_Y E \cdot \nabla_H Y = (((V^-)^T 2(Y - T)) * (\underline{1} - tanh^2(HX^+)))(X^+)^T$$

Update rules:

$$V_{t+1} \leftarrow V_t + \nabla_V E \cdot l$$

$$H_{t+1} \leftarrow H_t + \nabla_H E \cdot l$$

Terminate when:

$$\frac{\Sigma_{i=0}^{q} \Sigma_{j=0}^{n} E_{i,j}}{n * q} < \epsilon$$

Neural Network Review

- Run time of both forward and backward pass are dominated by matrix multiplication: O(qpn + pmn)

- Weight update is asymptotically squished: O(pm + qp)

- In practice, q is approximately m and n >> m and p

- Complexity grows linearly as number of samples increases alone

- While improving all operations will help for “smaller” problems, matrix multiply dominates asymptotically, i.e. for arbitrarily large problem sizes

Status

- It works!
- Both CPU and GPU implementations
- No comparison with optimized 3rd party versions... makes it a bit of a “straw man”
- Tested with XOR and noisy sinewave
- On these problems, precision doesn’t “appear” to be an issue

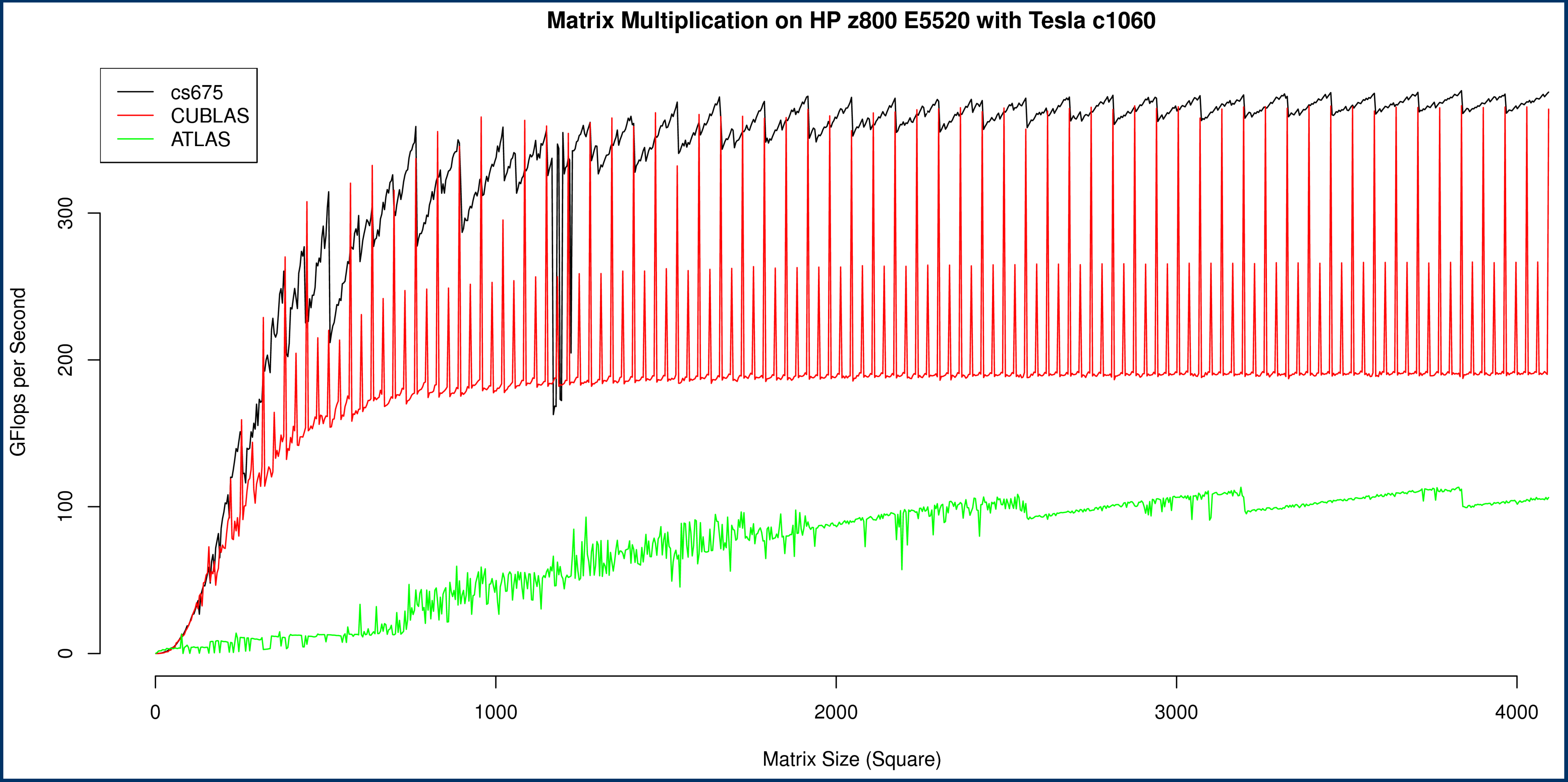
```
RMSE: 0.000020
RMSE: 0.000019
RMSE: 0.000019
RMSE: 0.000018
RMSE: 0.000018
RMSE: 0.000017
RMSE: 0.000017
RMSE: 0.000016
RMSE: 0.000016
RMSE: 0.000015
RMSE: 0.000015
RMSE: 0.000014
RMSE: 0.000014
RMSE: 0.000013
RMSE: 0.000013
RMSE: 0.000013
RMSE: 0.000012
RMSE: 0.000012
RMSE: 0.000011
RMSE: 0.000011
RMSE: 0.000011
RMSE: 0.000010
RMSE: 0.000010
RMSE: 0.000010
RMSE: 0.000010
RMSE: 0.000009
RMSE: 0.000009
RMSE: 0.000009
RMSE: 0.000009
RMSE: 0.000008
RMSE: 0.000008
RMSE: 0.000008
RMSE: 0.000007
RMSE: 0.000007
RMSE: 0.000007
inputs:
0.000000 1.000000 0.000000 0.000000 1.000000 0.000000 1.000000 1.000000 1.000000 1.000000
0.000000 1.000000 0.000000 1.000000 1.000000 0.000000 0.000000 0.000000 1.000000 1.000000
targets:
0.010000 0.010000 0.010000 0.990000 0.010000 0.010000 0.990000 0.990000 0.010000 0.010000
outputs:
0.010004 0.010005 0.010004 0.989987 0.010005 0.010004 0.989992 0.989992 0.010005 0.010005
platte:~/courses/cs675/badger$
```

Status

- My Implementation consists of a number of small kernels:
- ✓ Matrix multiply – ATLAS & CUDA
- ✓ Matrix transpose – SSE+OMP & CUDA
- ✓ Matrix-scalar multiply – SSE+OMP & CUDA
- ✓ Pointwise multiply, add, subtract – SSE+OMP & CUDA
- ✓ Multiply-Hyperbolic tangent – ATLAS + OPM & CUDA
- ✓ Apply derivative of hyperbolic tangent – SSE+OMP & CUDA
- ✓ Add/remove bias weights – Leave extra padding on CPU & GPU
- ✗ Summation / reduction – not done, just run for fixed iterations

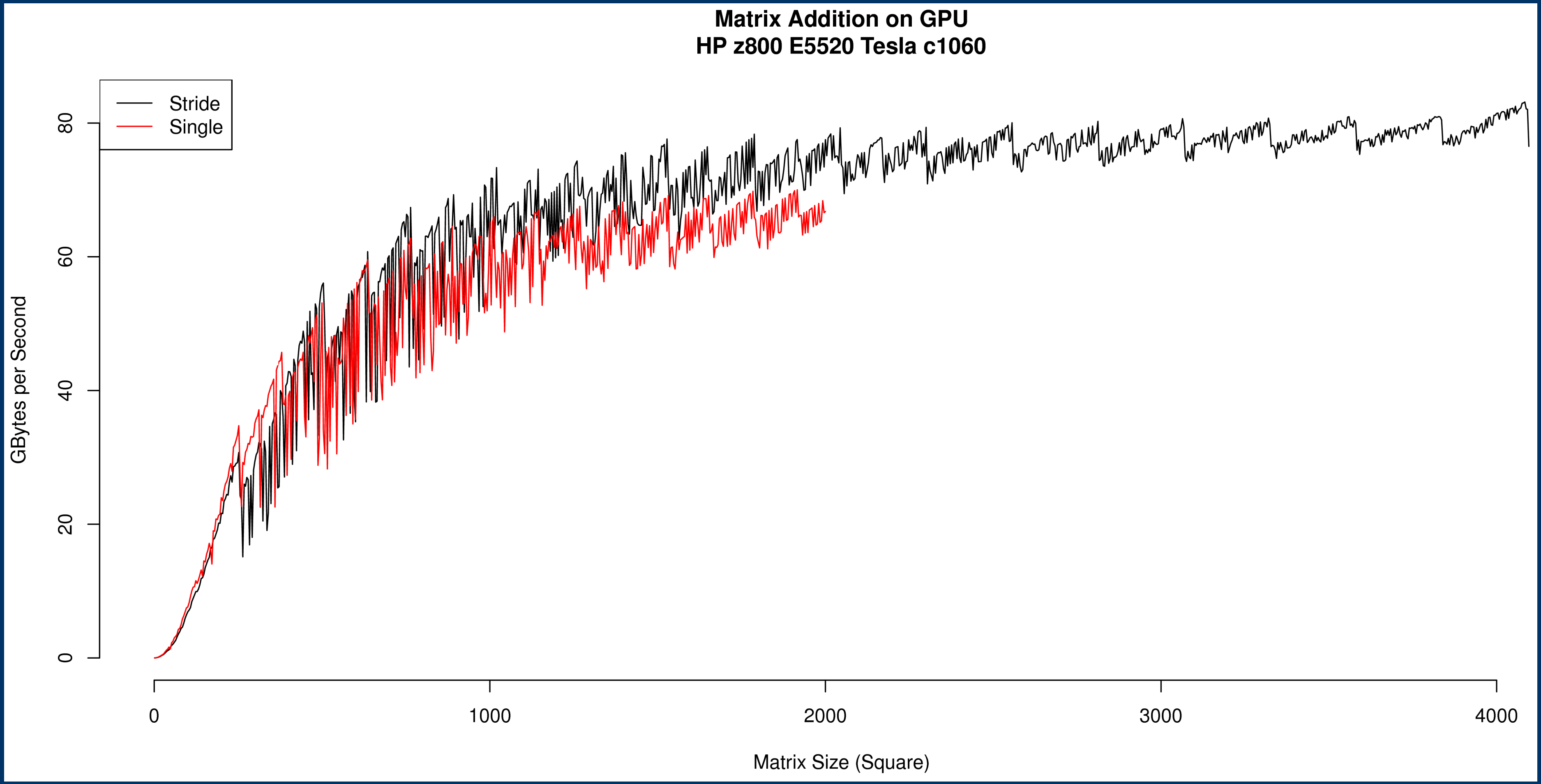
Micro-benchmarks

- CUDA matrix multiply beats CUBLAS for most matrix sizes
- However, zero padding makes life difficult.
- For CPU, ATLAS is hard to beat so we just use that... for now



Micro-benchmarks

- Two paradigms for pointwise operations in CUDA
- For small matrices, treat as vector and assign one thread per component
- For large matrices, use 2d grid and virtualize down columns

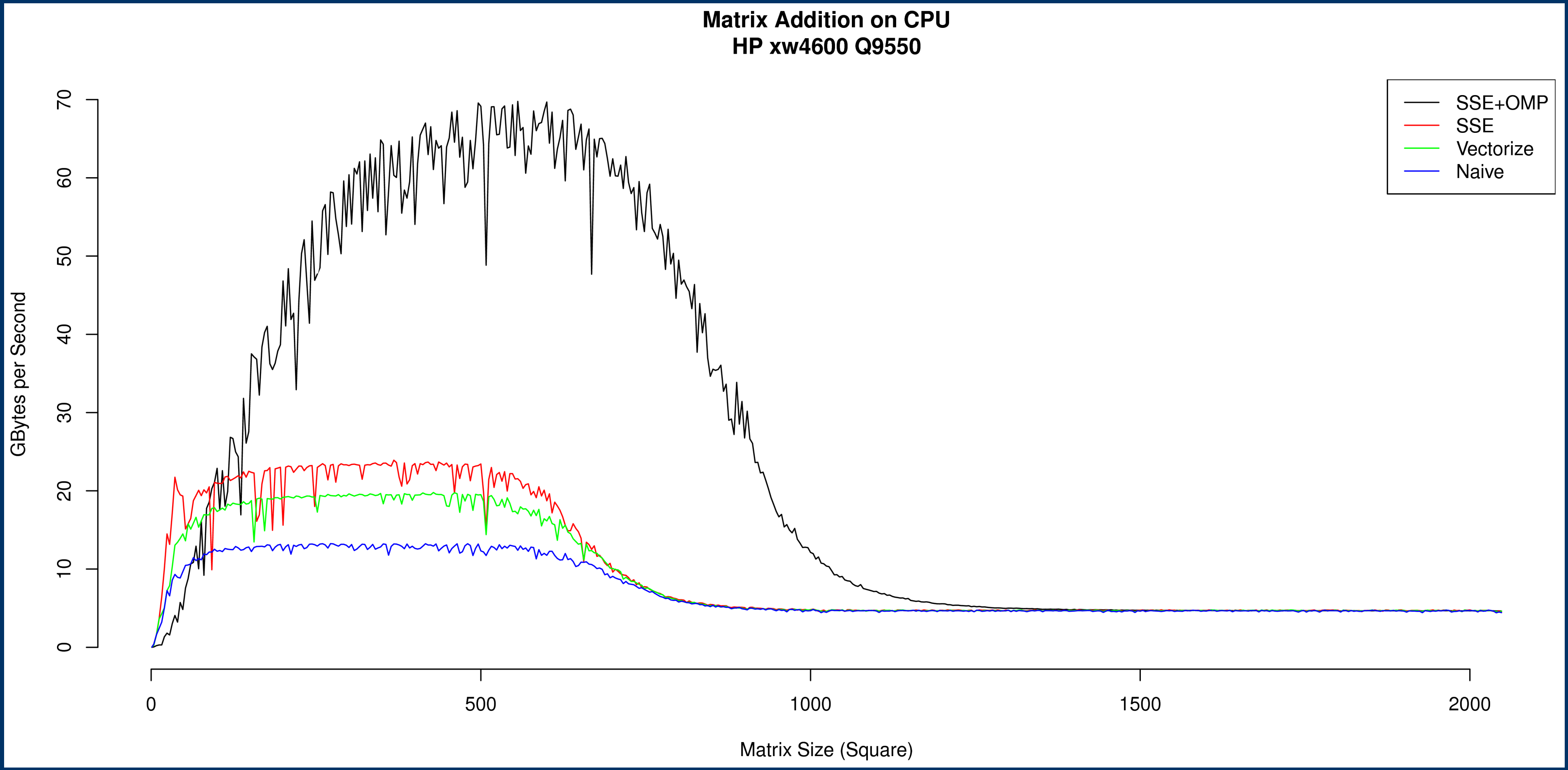


Micro-benchmarks

```
// addition kernel for small matrices
__global__ void add_small_kern(float *a, float *b, float *c, unsigned n)
{
  // unique id for each thread 0, ..., (nthreads-1)
  const unsigned id = threadIdx.x + blockIdx.x * blockDim.x;

  // if inside matrix
  if (id < n)
    // sum one value
    a[id] = b[id] + c[id];
}

// addition kernel for big matrices
__global__ void add_big_kern(float *a, float *b, float *c, unsigned n, unsigned stride)
{
  unsigned i;

  // unique id for each block, strided according to stripe size
  const unsigned block_index = blockIdx.x + blockIdx.y * gridDim.x * add_big_stripe;

  // unique id for each thread
  const unsigned id = threadIdx.x + block_index * blockDim.x;

  // each thread sums down a column stripe times
  #pragma unroll
  for (i = id; i < id+add_big_stripe*stride; i += stride)
    if (i < n)  // if inside matrix
      a[i] = b[i] + c[i]; // sum value
}
```

Micro-benchmarks

- Two paradigms for pointwise operations on CPU as well
- OpenMP hurts for small matrices and helps for mid-sized ones
- Overhead of thread launch vs multi-core & cache



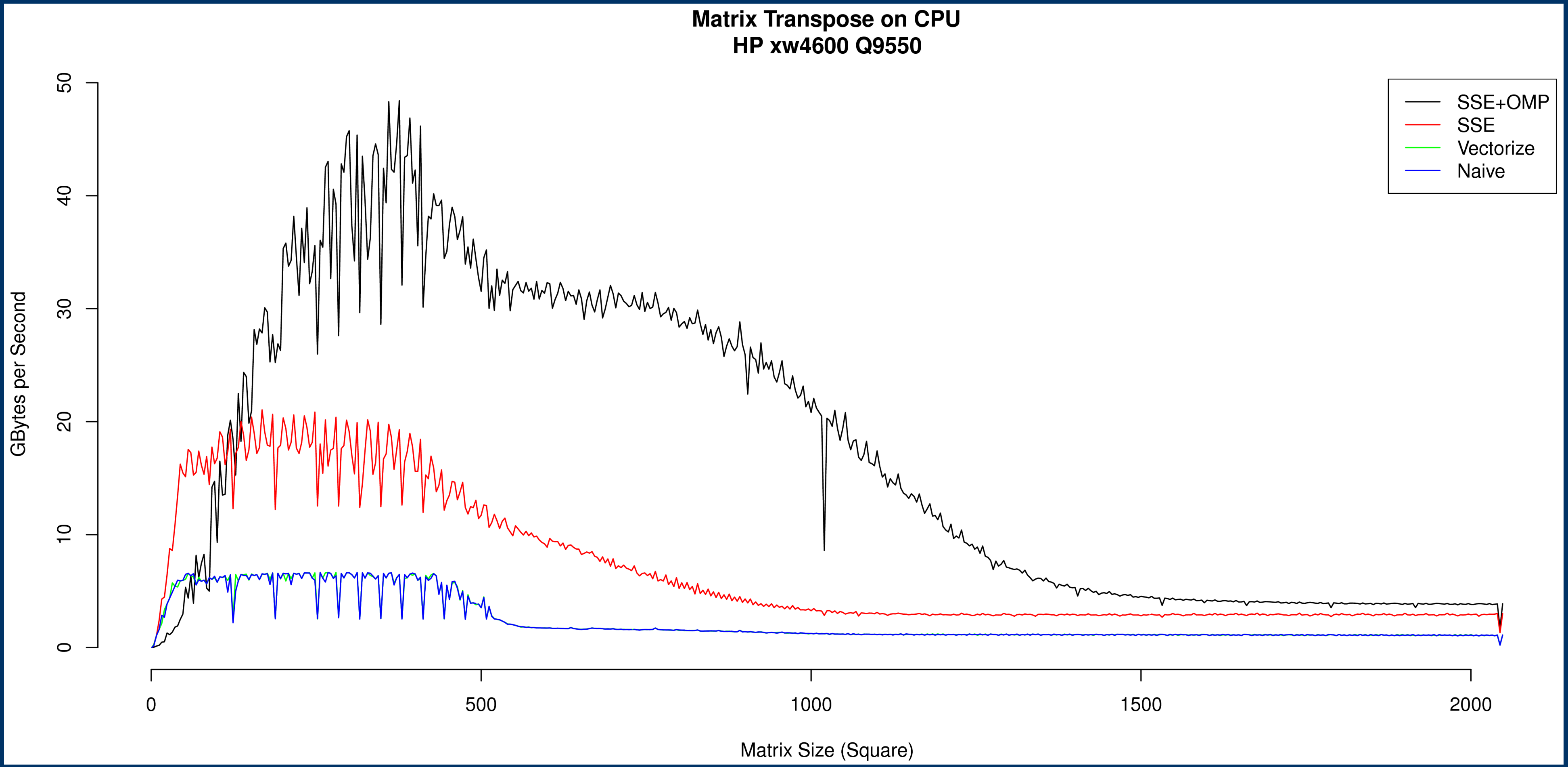
Micro-benchmarks

```c
// addition kernel for small matrices
void add_small(float *a, float *b, float *c, unsigned n)
{
  // pointer to last destination
  const float *a_end = a+n;

  // loop through each value in destination
  while (a < a_end)
  {
    // four values from b and c into sse registers
    __m128 mm_v1 = _mm_load_ps(b);
    __m128 mm_v2 = _mm_load_ps(c);

    // add our sse vectors
    mm_v1 = _mm_add_ps(mm_v1, mm_v2);

    // store result into a
    _mm_store_ps(a, mm_v1);

    // increment pointers by 4
    a += 4; b += 4; c += 4;
  }
}
```

```c
// addition kernel for big matrices
void add_big(float *a, float *b, float *c, unsigned n)
{
  unsigned i;

  // loop through a, b and c by 4 in parallel
  #pragma omp parallel for
  for (i = 0; i < n; i += 4)
  {
    // load four values into sse registers
    __m128 mm_v1 = _mm_load_ps(b+i);
    __m128 mm_v2 = _mm_load_ps(c+i);

    // add our sse vectors
    mm_v1 = _mm_add_ps(mm_v1, mm_v2);

    // store result into a
    _mm_store_ps(a+i, mm_v1);
  }
}
```

Micro-benchmarks

- CPU Transpose below, tradeoff similar to addition
- CUDA transpose follows follows principals from NVIDIA paper
- 75 GbyteS, roughly 5 GbyteS improvement by tweaking tile size



Micro-benchmarks

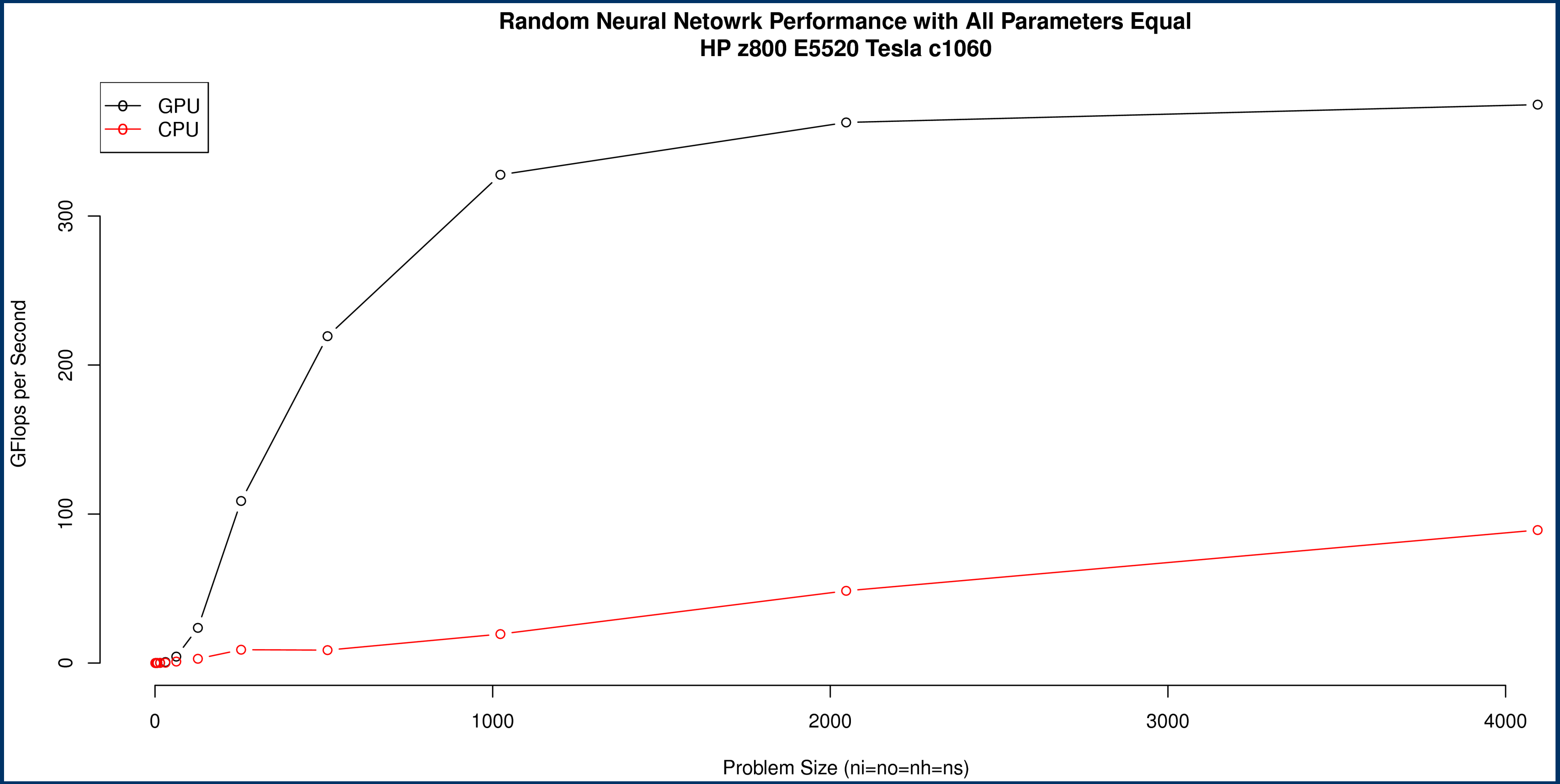
```
// transpose kernel a = b^T
__global__ void trans_kern(float *a, float *b, unsigned nrow,
                           unsigned astride, unsigned bstride)
{
  unsigned i, blockIdx_x, blockIdx_y;

  if (nrow == astride) { // this block borrow from CUDA SDK
    blockIdx_y = blockIdx.x; // Thanks!
    blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
  } else {
    const unsigned bid = blockIdx.x + gridDim.x*blockIdx.y;
    blockIdx_y = bid%gridDim.y;
    blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
  }

  const unsigned tile_r_stripe = trans_tile_r * trans_stripe;
  const unsigned tid_y_stripe  = threadIdx.y  * trans_stripe;

  const unsigned block_row = blockIdx_y * tile_r_stripe;
  const unsigned block_col = blockIdx_x * trans_tile_c;

  unsigned row  = block_col + tid_y_stripe;
  unsigned col  = block_row + threadIdx.x;
  unsigned base = row*bstride + col;

  __shared__ float tile[trans_tile_c][tile_r_stripe+1];

  #pragma unroll
  for (i = 0; i < trans_stripe; ++i)
    tile[threadIdx.x][tid_y_stripe+i] = b[base+i*bstride];

  __syncthreads();

  row  = block_row + tid_y_stripe;
  col  = block_col + threadIdx.x;
  base = row*astride + col;

  #pragma unroll
  for (i = 0; i < trans_stripe; ++i)
    a[base+i*astride] = tile[tid_y_stripe+i][threadIdx.x];
}
```

Micro-benchmarks

```
#pragma omp parallel for private(c)
for (r = 0; r < a.r; r += 4)
  for (c = 0; c < a.c; c += 4)
  {
    // load 4x4 tile
    float *base = bdata + c*bstride+r;
    __m128 mm_v1 = _mm_load_ps(base            );
    __m128 mm_v2 = _mm_load_ps(base +   bstride);
    __m128 mm_v3 = _mm_load_ps(base + 2*bstride);
    __m128 mm_v4 = _mm_load_ps(base + 3*bstride);

    // transpose 4x4 tile
    _MM_TRANSPOSE4_PS(mm_v1, mm_v2, mm_v3, mm_v4);

    // store 4x4 tile back into a
    base = adata + r*astride+c;
    _mm_store_ps(base,             mm_v1);
    _mm_store_ps(base +   astride, mm_v2);
    _mm_store_ps(base + 2*astride, mm_v3);
    _mm_store_ps(base + 3*astride, mm_v4);
  }
```

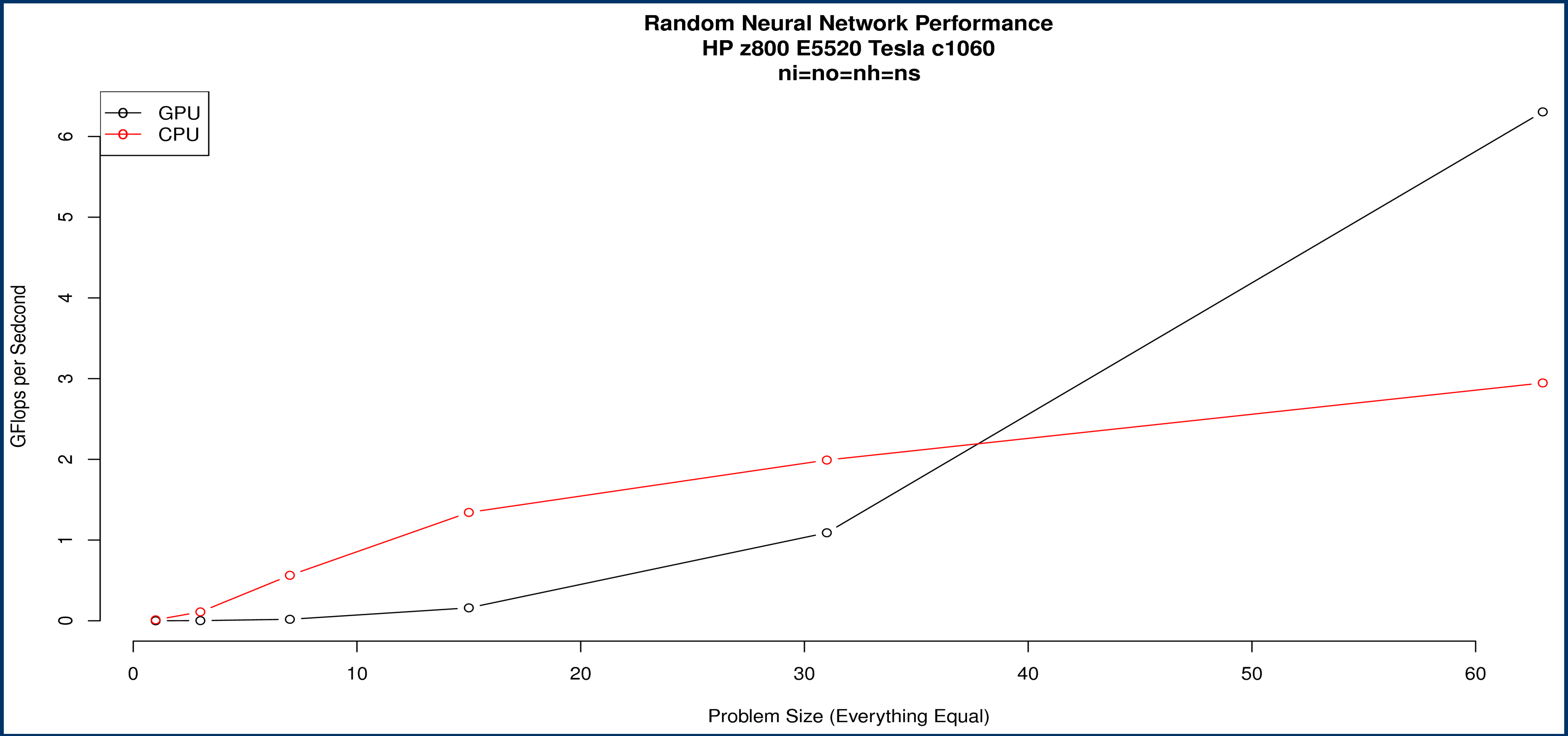
Macro-benchmarks

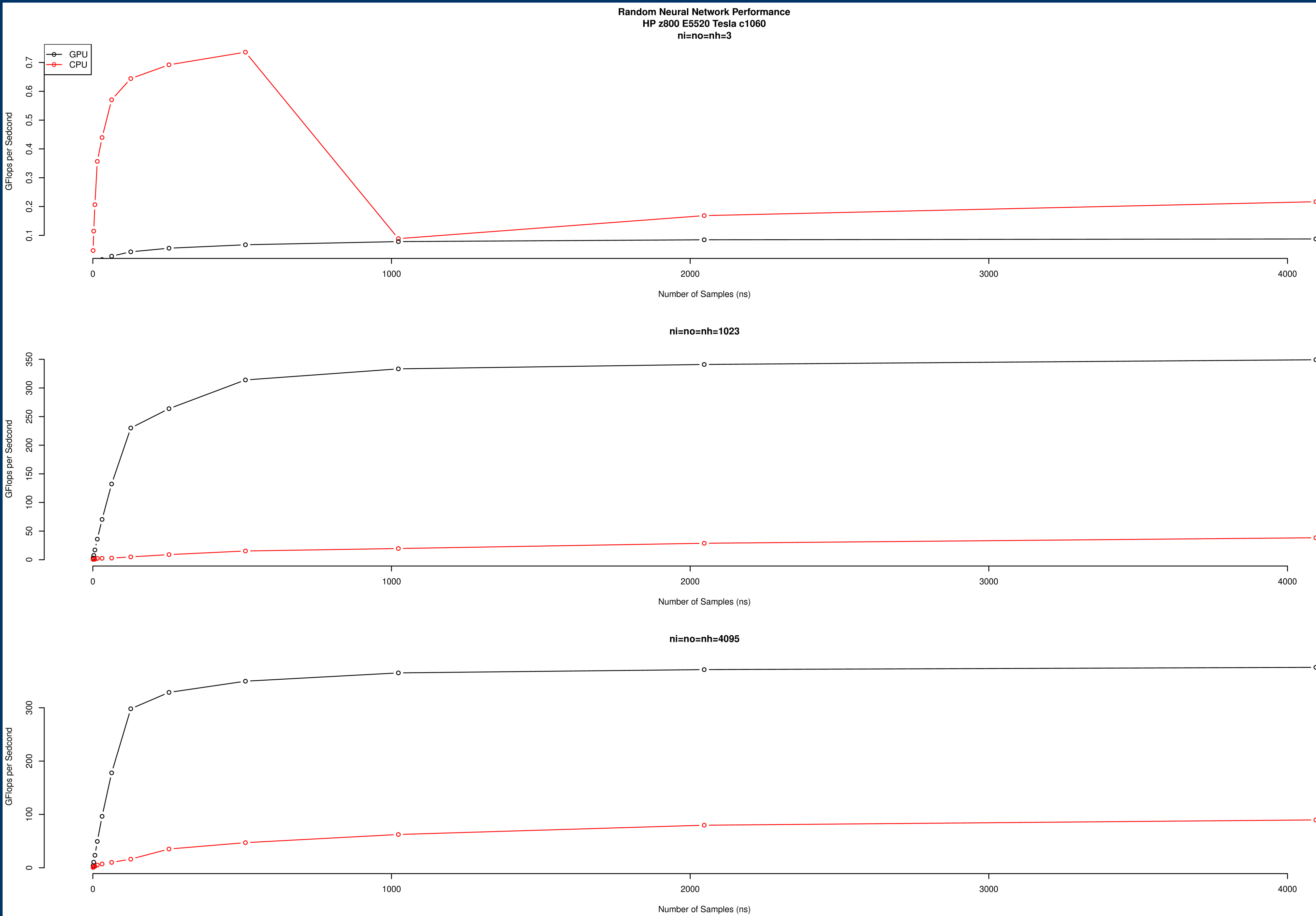
- Use random inputs and targets
- Let ni=no=nh=ns and vary 1 3 7 15 31 63 127 255 511 1023 2047 4095
- CPU version gets up to 90 GflopS, GPU version 375 GflopS, 4x speedup



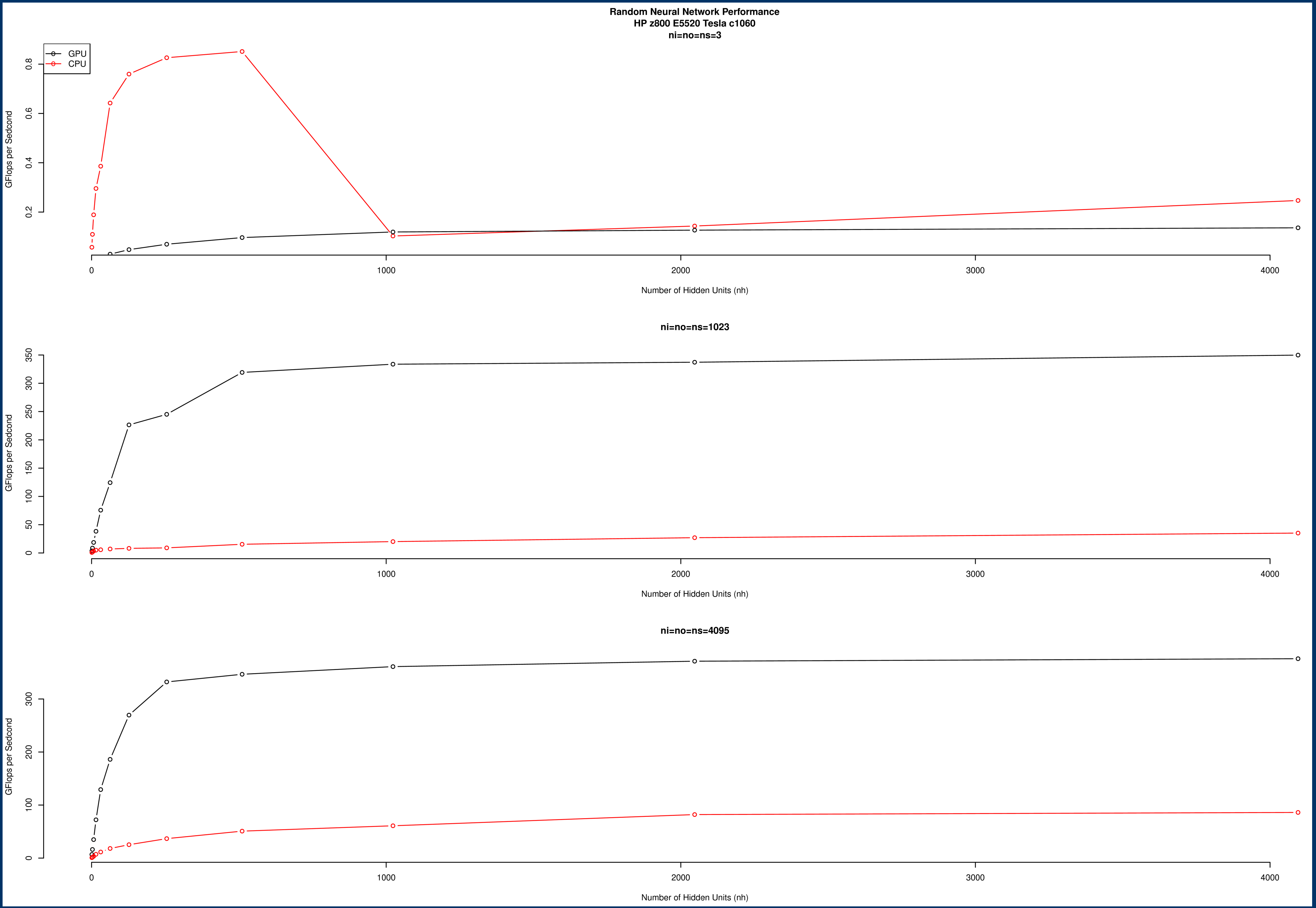
Macro-benchmarks

- Same image as previous but zoomed in on small problems
- CPU version beats GPU for problems smaller than about 40
- Smaller padding, not enough thread blocks, transfer overhead





Random Neural Network Performance
HP z800 E5520 Tesla c1060
ni=no=nh=3
GPU
CPU
GFlops per Sedcond
Number of Samples (ns)
ni=no=nh=1023
GFlops per Sedcond
Number of Samples (ns)
ni=no=nh=4095
GFlops per Sedcond
Number of Samples (ns)


Random Neural Network Performance
HP z800 E5520 Tesla c1060
ni=no=ns=3
GPU
CPU
GFlops per Sedcond
Number of Hidden Units (nh)
ni=no=ns=1023
GFlops per Sedcond
Number of Hidden Units (nh)
ni=no=ns=4095
GFlops per Sedcond
Number of Hidden Units (nh)

Conclusions

- For large problems, CUDA can provide up to a 4x speedup
- For smaller problems, CPU version still beats CUDA, even for some long non-square matrices, i.e. many samples or hidden units with few inputs & outputs
- Working in CUDA is “relatively” straight forward but in some ways can be more cumbersome than SSE & OMP
- SSE and OpenMP are fairly easy and can provide nice speedups, especially but not exclusively on compute bound tasks

Future Improvements

- Test performance on more real-world problems!
- Better weight updates, SCG, Rprop, Alopex
- Fused multiply-transpose & transpose-multiply
- Reduction to compute sum error measures
- Parallel random number generation for weight initialization
- Autotuning small/big kernel boundaries
- More microbenchmarks
- Clean up code and interface
- Better error checking and handling

Yay, summertime!