Sparse Training Data

Tutorial of Parameter Server

Mu Li

CSD@CMU & IDL@Baidu

muli@cs.cmu.edu

High-dimensional data are sparse

- Why high dimension?
  - make the classifier's job easier
  - linear method is often good enough
- Why sparse?
  - easy to storage (only store non-zero entries)
  - affordable computation cost
- Key difference to dense data when using
  - require a lot of random read/write

Store and Computing

Compressed storage

	0	1	2
0		10	20
1	3		
2		87	
3	57		

- Sparse row-major:

  offset = [0, 2, 3, 4, 5]
  index = [1, 2, 0, 1, 0]
  value = | 10 | 20 | 3 | 87 | 57 |

- Sparse column-major:

  offset = [0, 2, 4, 5]
  index = [1, 3, 0, 2, 0]
  value = | 3 | 57 | 10 | 87 | 20 |

- Access a(i,j) under row-major:

k = **binary_search**(index[offset[i]], index[offset[i+1]], j)
return valid(k) ? value(offset[i]+k) : 0

$y = Ax$

all offset, index, and value are read sequentially

- Sample C++ codes:

```
// matrix-vector multiplication y = A * x
voi                                t V* x const, V* y) const {
  if
                                   ); ++i) {  // i-th row

                                   j < offset[i+1]; ++j)
        y_i += x[index[j]] * value[j];

    }
  } e
    n                              );
    fo                                ); ++i) {  // i-th column
      V x_i = x[i];
      for (size_t j = offset[i]; j < offset[i+1]; ++j)
        y[index[j]] += x_i * value[j];
    }
  }
}
```

write y sequentially, but read x in random

read x sequentially, but write y in random

Numbers Everyone Should Know

L1 cache reference	10 times	0.5 ns
Branch mispredict		5 ns
L2 cache reference	10 times	7 ns
Mutex lock/unlock		100 ns
Main memory reference		100 ns
Compress 1K bytes with Zippy		10,000 ns
Send 2K bytes over 1 Gbps network		20,000 ns
Read 1 MB sequentially from memory		250,000 ns
Round trip within same datacenter		500,000 ns
Disk seek		10,000,000 ns
Read 1 MB sequentially from network		10,000,000 ns
Read 1 MB sequentially from disk		30,000,000 ns
Send packet CA->Netherlands->CA		150,000,000 ns

slides by Jeff Dean

Cost of $y = Ax$

- The computation cost is *O(nnz(A))*
- The random access dominates the cost:

$$\approx \text{L2-cache-reference}(\text{nnz}(A))$$

- In theory: process 1.4e8 nnz entries per second
- In reality: 8.4e7 nnz entries per second
  - 4.3M x 17.4M sparse matrix
  - mac mbp, Intel i7 2.3GHz cpu
  - single thread

Real data

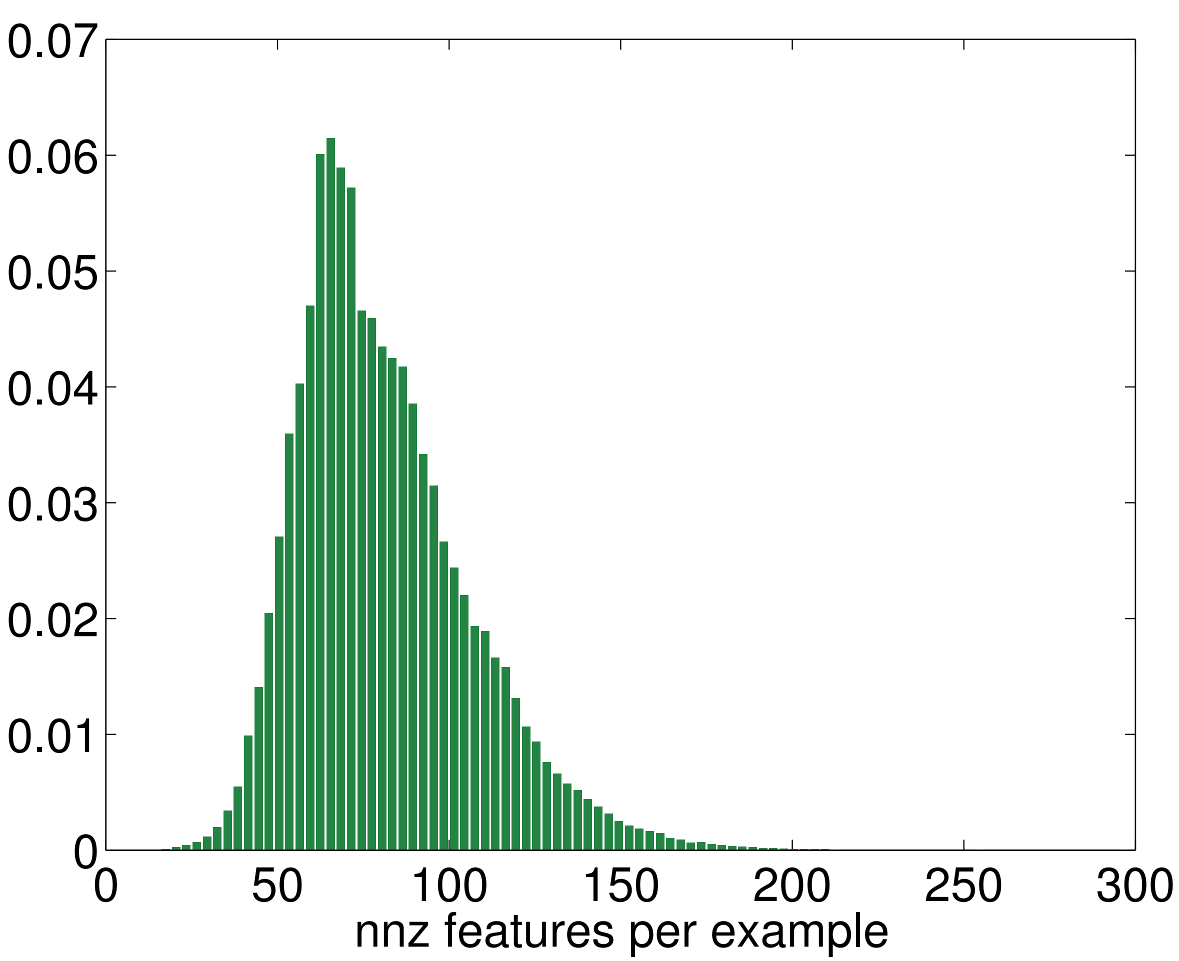
Product	Examples	Training data	Features per example
A	59.9B	2.00TB	54.9
B	7.6B	0.71TB	94.9
C	197.5B	15.54TB	77.7
D	129.1B	17.24TB	100.57

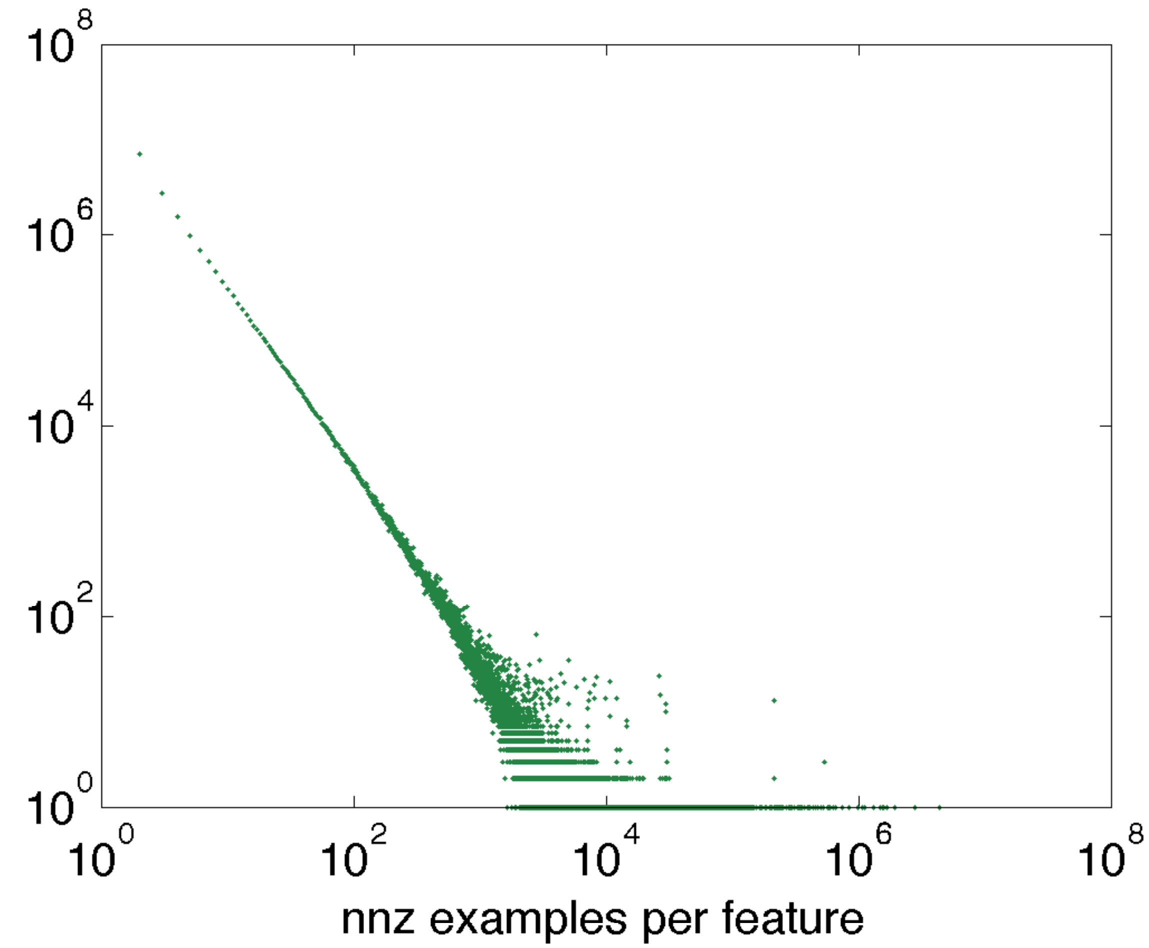
from sibyl

- ≈100 features per example is reasonable
  - ≈feature groups
  - for y=Ax, process 1e6 examples per second
  - for linear method, 1000 cores, 100 billion examples, 100 iterations, finish in 3h in ideal

$$10^{11} \text{ examples} \times 100 \text{ iterations}/1000 \text{ cores}/10^6 = 1000 \text{ second}$$

Patterns of Sparsity



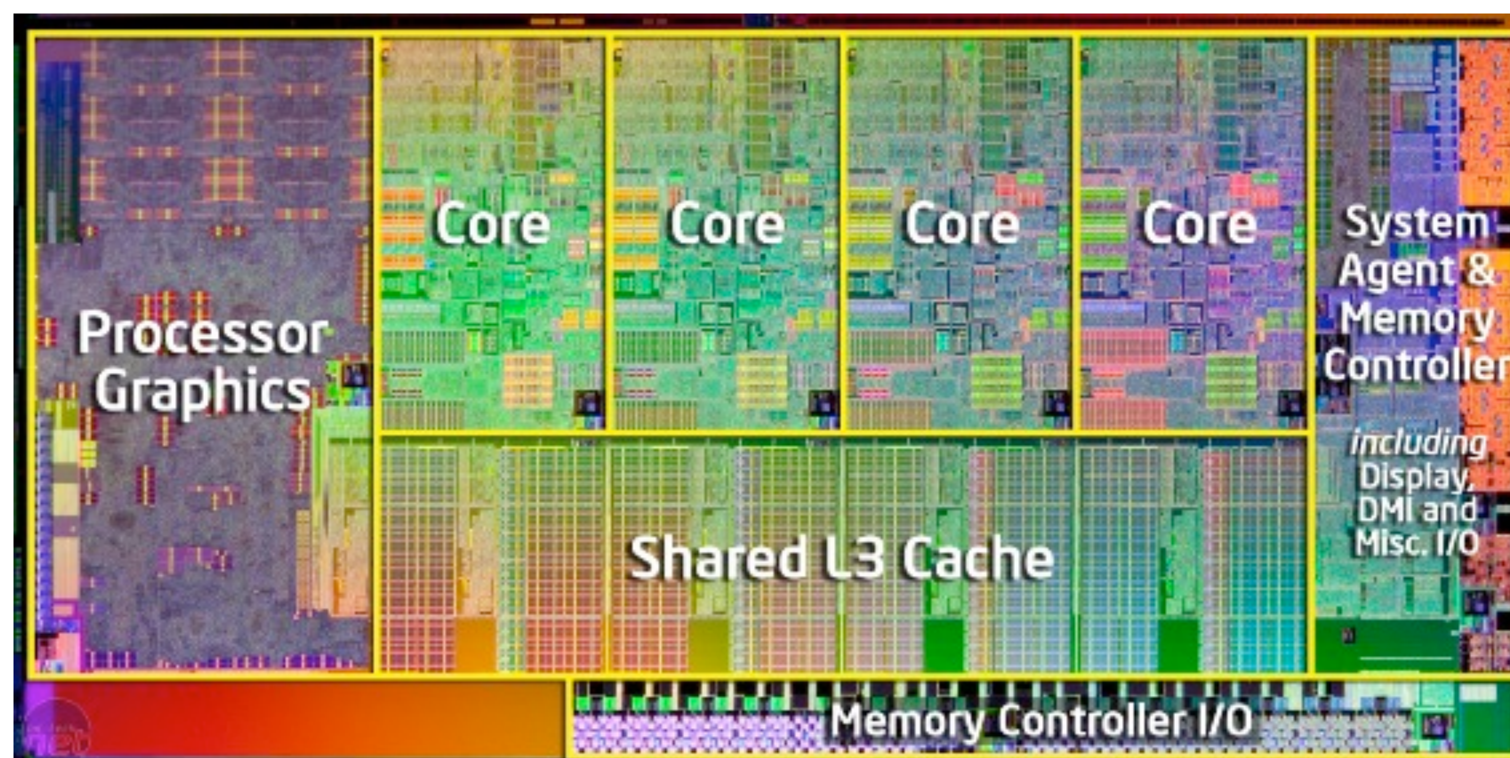


- non-zero entries are distributed irregularly on features
  - imbalanced workload partition
  - ill conditional number

Multi-thread Implementation

CPU

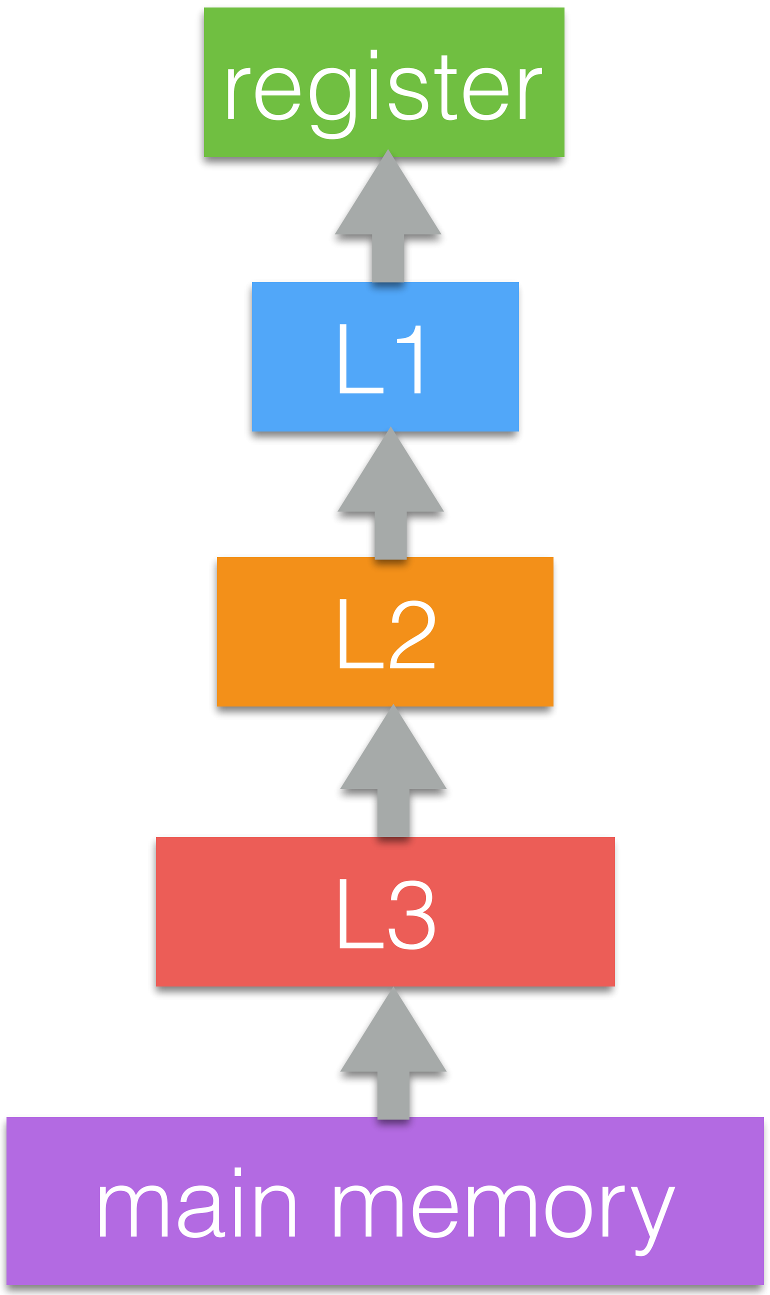
- Multiple cores (4-8)
- Multiple sockets (1-4)
- 2-4 GHz clock
- Memory interface 20-40GB/s
- Internal bandwidth >100GB/s



Benefits of multi-thread

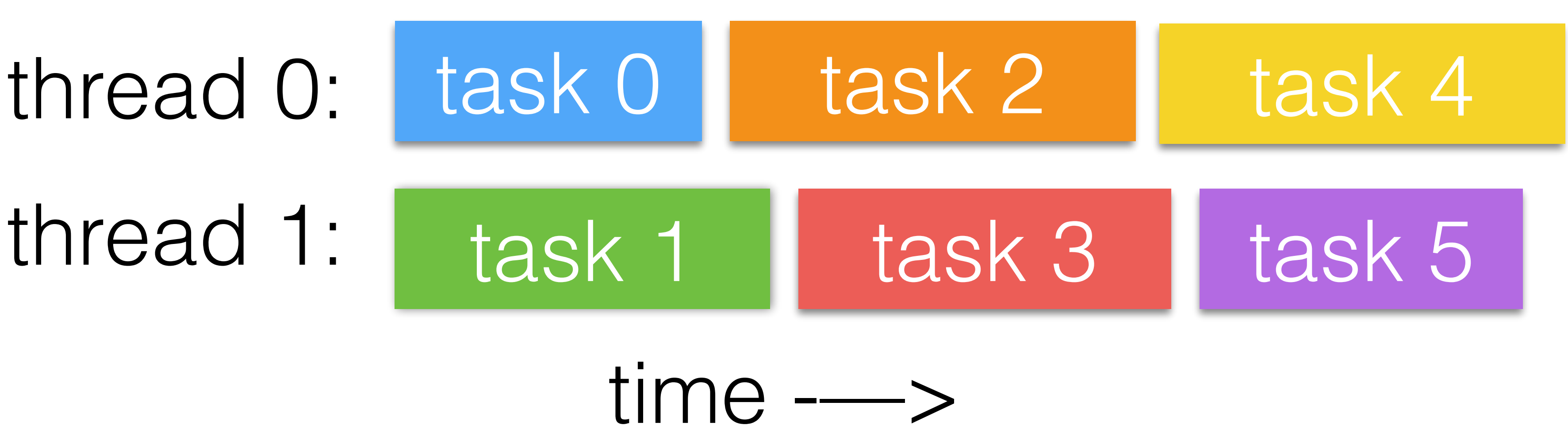
- Use more computation units
  - float point units
- Hide the memory latency
  - run something else when the data are not ready

fetch data from memory
~100 cycles



Using ThreadPool

- A pool of threads, each one keeps fetching and executing unfinished tasks
- Create a pool with n threads: ThreadPool pool(n)
- Add a task into the pool: pool.add(task)
- Start executing: pool.startWorkers()



Multi-threaded $y = Ax$

- Assume row major
- Compute a segment of y

```
void rangeTimes(SizeR row_range, const V* const x, V* y) const;
```

- Divide y into several segments, each one is assigned to a thread

```
ThreadPool pool(num_threads);
int num_tasks = rowMajor() ? num_threads * 10 : num_threads;
for (int i = 0; i < num_tasks; ++i) {
  pool.add([this, x, y, row_range, num_tasks, i](){
      rangeTimes(row_range.evenDivide(num_tasks, i), x, y);
    });
}
pool.startWorkers();
```

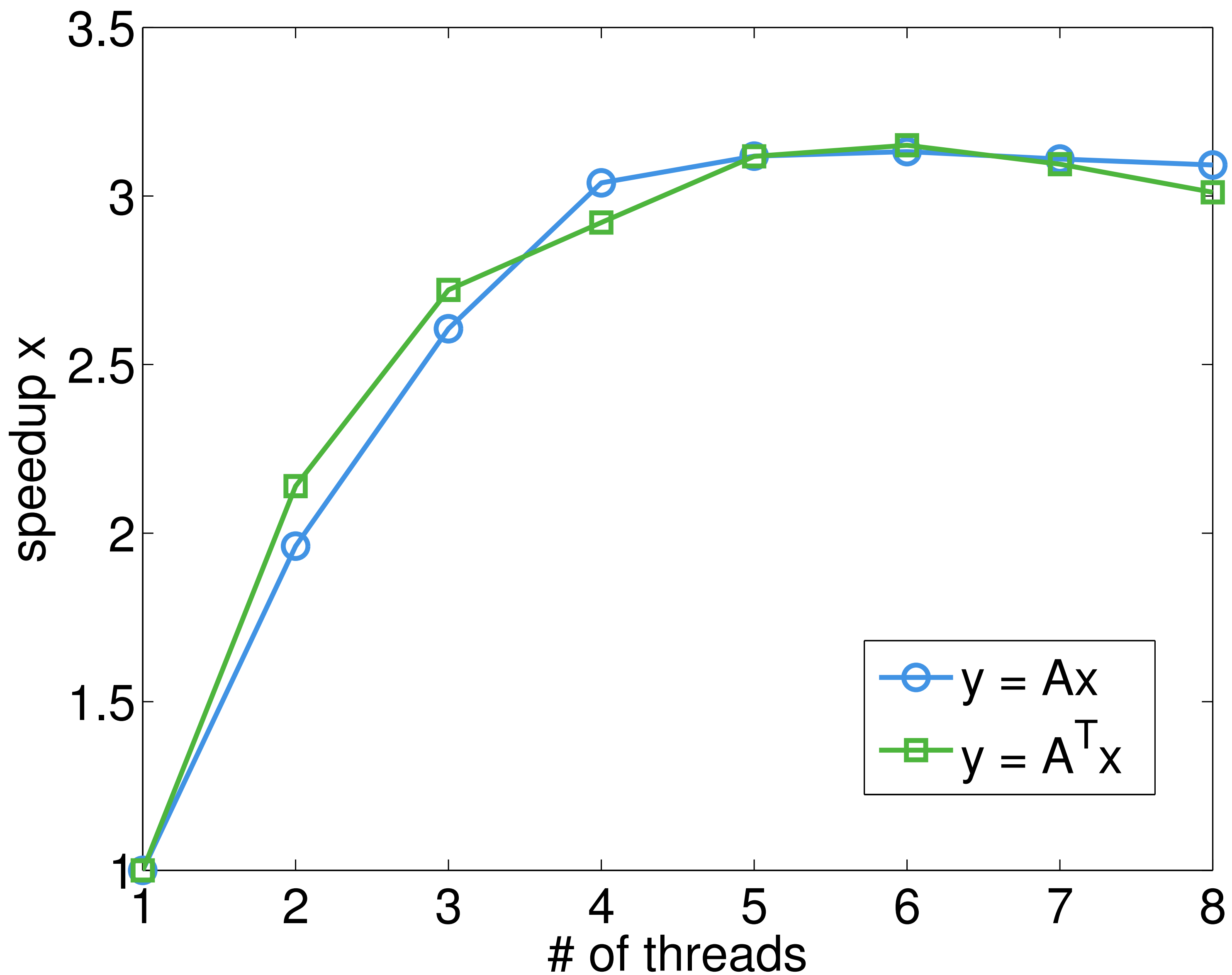
c++11 lambda functions

How about column-major?

- Equivalence to $x = A^T y$ for row-major A
- multi-threads concurrently write the same y
- Several possible solutions:
  - convert into a row-major matrix first
  - lock y[i] (or a segment) before write it
  - each thread writes only a segment of y

Experiments

- data: CTRa
- row major,
  4.3M rows,
  17.4M columns,
  354M nnz entries
- MBP Pro,
  Intel i7 2.3GHz,
  4 cores,
  8 hyper-threads



Coding Practice

- Implement $x = A^T y$
- You can reuse the codes at
  - https://github.com/mli/mlss14_a
- CTRa in binary format is provided
  - CTRa_X.index: 354700138 uint32
  - CTRa_X.offset: 4349786 uint64
  - CTRa_X.info: information
  - 0-1 values, so ignore the value

Row major or column major

- No big difference for individual and whole access
  - choose the one how data are stored
- Use row major when need read individual rows
  - SGD, minibatch SGD, online learning
- Use column major when need read columns
  - (block) Coordinate descent
- Converting cost 2*nnz(A) random access

More

- Other operations? BLAS, LAPACK
  - Timothy A. Davis, Direct Methods for Sparse Linear Systems, SIAM, 2006
- Existing packages:
  - SuiteSparse, Eigen3…
  - Use them as much as possible
  - however, problem-specific optimizations may improve the performance a lot, we will see later

Eigen3

- Easy to install: all header files, just copy to a proper place
- Not easy to read: a lot of templates
- Good performance on dense data
- Somewhat convenient to use
- Jeff Dean is using it…

Bug 613 - Bug in internal::psqrt SSE implementation

Status: RESOLVED FIXED	**Reported:** 2013-06-13 17:54 UTC by Jeff Dean
Product: Eigen	**Modified:** 2013-06-14 09:52 UTC (History)
Component: Core - general	

add subtract	`mat3 = mat1 + mat2;` `mat3 = mat1 - mat2;`	`mat3 += mat1;` `mat3 -= mat1;`
scalar product	`mat3 = mat1 * s1;` `mat3 = mat1 / s1;`	`mat3 *= s1;` `mat3 /= s1;`

matrix/vector products *	`col2 = mat1 * col1;`
	`row2 = row1 * mat1;`
	`mat3 = mat1 * mat2;`
		
		`row1 *= mat1;`
		`mat3 *= mat1;`

transposition adjoint *	`mat1 = mat2.transpose();` `mat1 = mat2.adjoint();`	`mat1.transposeInP` `mat1.adjointInPla`

dot product
inner product *
	`scalar = vec1.dot(vec2);`
	`scalar = col1.adjoint() * col2;`
	`scalar = (col1.adjoint() * col2).value();`	

outer product *	`mat = col1 * col2.transpose();`	