In-Memory Data Compression for Sparse Matrix on GPU & CPU

Dr. Orion Sky Lawlor
lawlor@alaska.edu
U. Alaska Fairbanks

IA^3 workshop, SC13
2013-11-17

Why Compress Memory?

“Communication dominates energy”

— Keckler, Dally, Khailany, Garland, and Glasco, 2011

“Minimizing data transport energy, rather than arithmetic logic unit (ALU) energy, is the real challenge.”

— Josep Torrellas, 2009

“At scale arithmetic is free, and communication is expensive.”

— me, now

Modern CPU Storage Hierarchy

Level	*Cost*	*Controller*
Operand ports	sub-ns	Superscalar scheduler
Register file	0.3ns	Rename hardware
L1 Cache	1 ns	Load-store unit
L2 Cache	2 ns	Per-Core Cache Manager
L3 Cache	10 ns	Multicore Cache Manager
Socket RAM	30 ns	OS Memory Allocator
NUMA RAM	60 ns	OS Memory Allocator
LAN / Cluster	80 us	MPI, hadoop
WAN / Cloud	10 ms	OS TCP stack

Disclaimer: costs are quoted to zero significant digits

Tradeoffs: Compressing Memory

- **Compressed data uses less <u>bandwidth</u> between levels**
  - **Linear speedup, e.g., save 20% storage, save 20% bandwidth**
- **Compressed data might <u>fit</u> in a higher level**
  - **Nonlinear gains possible, e.g., fit in cache, get 3x speedup**
- **To be useful, benefits must outweigh encoding and decoding <u>costs</u>**
  - **Good implementation matters!**

Sparse Matrix

- **Matrix is a modern *lingua franca***
  - **Computer scientists, application scientists, mathematicians**
- **Dense matrix is wildly inefficient**
  - **Mesh-derived matrix is sparse**

    E.g., 5 nonzeros/row, 10^6 cols!
- **"Sparse Matrix" skips over zeros**
  - **Only store or compute nonzeros**

  E.g., 10^6 x 10^6 matrix in 10^8 bytes
- **See Intel MKL, NVIDIA CUSPARSE**

Compressed Sparse Row: CSR
(Prior Work)

Prior: Compressed Sparse Row

For example, we will encode this 3×3 matrix in CSR form.

$$\begin{bmatrix} 9 & 5 & 0 \\ 0 & 8 & 0 \\ 6 & 0 & 7 \end{bmatrix}$$

Start with a matrix

Prior: Compressed Sparse Row

For example, we will encode this 3×3 matrix in CSR form.

$$\begin{bmatrix} 9 & 5 & 0 \\ 0 & 8 & 0 \\ 6 & 0 & 7 \end{bmatrix}$$

In zero-based CSR format the three arrays will contain these values.

val= $\begin{bmatrix} 9 & 5 & 8 & 6 & 7 \end{bmatrix}$ matrix nonzero values (real)

Write down the nonzeros or "values"

Prior: Compressed Sparse Row

For example, we will encode this 3×3 matrix in CSR form.

$$\begin{bmatrix} 9 & 5 & 0 \\ 0 & 8 & 0 \\ 6 & 0 & 7 \end{bmatrix}$$

In zero-based CSR format the three arrays will contain these values.

val= $\begin{bmatrix} 9 & 5 & 8 & 6 & 7 \end{bmatrix}$ matrix nonzero values (real)

col= $\begin{bmatrix} 0 & 1 & 1 & 0 & 2 \end{bmatrix}$ corresponding column index (int)

Write down the column numbers

Prior: Compressed Sparse Row

For example, we will encode this 3×3 matrix in CSR form.

$$\begin{bmatrix} 9 & 5 & 0 \\ 0 & 8 & 0 \\ 6 & 0 & 7 \end{bmatrix}$$

In zero-based CSR format the three arrays will contain these values.

val= $\begin{bmatrix} 9 & 5 & 8 & 6 & 7 \end{bmatrix}$ matrix nonzero values (real)

col= $\begin{bmatrix} 0 & 1 & 1 & 0 & 2 \end{bmatrix}$ corresponding column index (int)

start= $\begin{bmatrix} 0 & 2 & 3 & 5 \end{bmatrix}$ each row's first index in arrays above

Start column for each row

Prior: CSR Matrix-Vector Product

```
void csr_product(const real *src,real *dest)
{

    for (int r=0;r<max_row;++r) {
        real sum=0.0;
        for (int i=start[r];i<start[r+1];++i)
            sum+=vals[i]*src[ cols[i] ];
        dest[r]=sum;
    }
}
```

Prior: CSR Matrix-Vector Product

```
void csr_product(const real *src,real *dest)
{

    for (int r=0;r<max_row;++r) {
        real sum=0.0;
        for (int i=start[r];i<start[r+1];++i)
            sum+=vals[i]*src[ cols[i] ];
        dest[r]=sum;
    }
}
```

Prior: CSR on Multicore

```
void csr_product(const real *src,real *dest)
{

    for (int r=0;r<max_row;++r) {
        real sum=0.0;
        for (int i=start[r];i<start[r+1];++i)
            sum+=vals[i]*src[ cols[i] ];
        dest[r]=sum;
    }
}
```

- **SIMD (SSE/AVX) doesn't help at all due to the vector <u>gather</u> cost**

Prior: CSR on Multicore

```
void csr_product(const real *src,real *dest)
{
#pragma omp parallel for schedule(dynamic,32)
    for (int r=0;r<max_row;++r) {
        real sum=0.0;
        for (int i=start[r];i<start[r+1];++i)
            sum+=vals[i]*src[ cols[i] ];
        dest[r]=sum;
    }
}
```

- **OpenMP helps, but scales poorly: "irregular memory read" rate limited**

Compressed Column Index: CCI
(new work)

New: Compressed Column Index

- Compress the “cols” array only
  - Rationale: “vals” is application data, “start” is already small
- Store each column index as a variable bit-width jump instead of “int”
  - Reduce 32 bits to 2-3 bits
- Compression must be lossless
- Most matrix column indexes shrink by over 90%!

E.g. Compressed Column Index

- **Initial size: 5 ints, 160 bits**

$$\text{col}=\begin{bmatrix} 0 & 1 & 1 & 0 & 2 \end{bmatrix}$$

- **In fixed-length binary code, 10 bits**

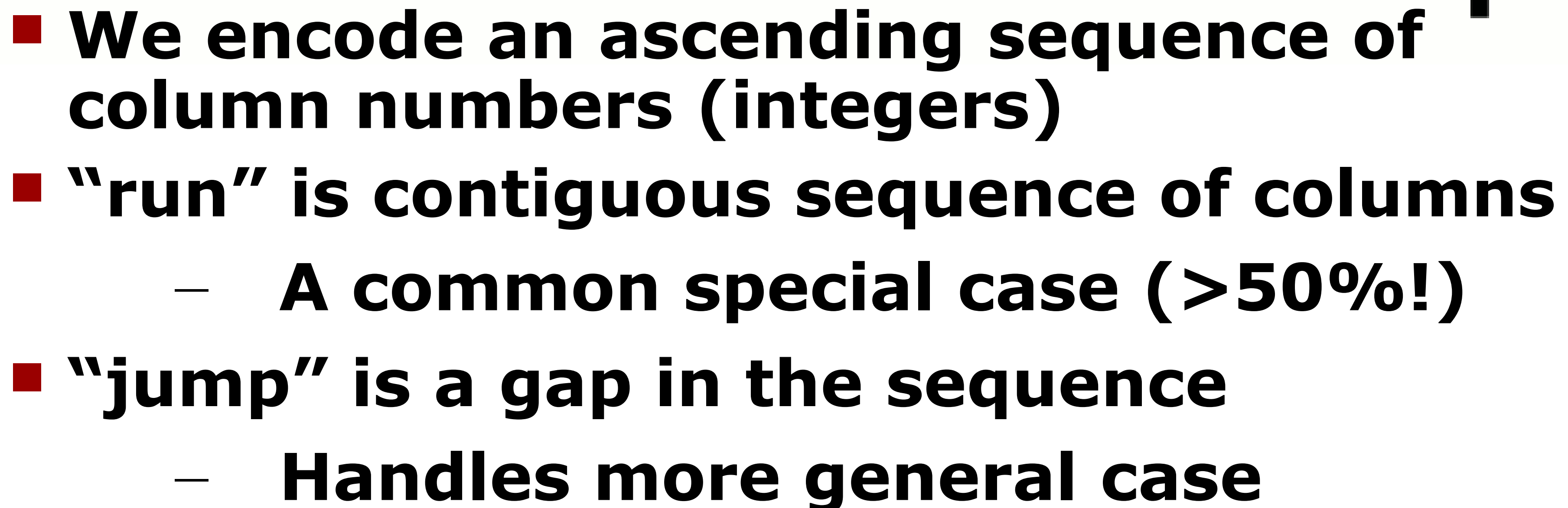
00 01 01 00 10

- **In variable-length code, 9 bits**

1 01 01 1 001

Actual Variable-Length Code

subsequent compressed data	4-bit run	0
	5-bit jump	001
	15-bit jump	011
	20-bit jump	101
	29-bit jump	111



- **We encode an ascending sequence of column numbers (integers)**
- **"run" is contiguous sequence of columns**
  - **A common special case (>50%!)**
- **"jump" is a gap in the sequence**
  - **Handles more general case**

Compressio

Can you make it run fast?

Because if it's not fast, it's useless.

Running Fast: Avoid Branches

- **Typical compression implementation uses sequence of branch instructions:**
  - **Is it the 1-bit code?**
  - **Or is it this 3-bit code?**
  - **How about this 3-bit code?**
- **Switch uses a branch table**

- **This is very hard on branch predictors**

"If those branches are predictable, by definition your compression is bad"

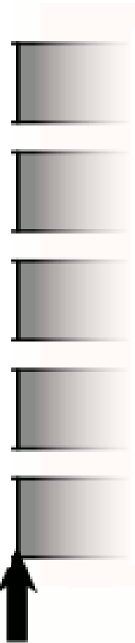
--Chris Hartman, U. Alaska

Running Fast: Use Decode Table

- Avoid branches using decode table
  - Always look up 3 bits
  - Index into 8-entry table
  - Add duplicates for shorter codes
  - See [Edfors et al 1993]
- Can shrink the decode table by splitting opcode from data
  - Table stores a bit mask to extract data portion
- Result: branch-free decoding

Running Fast: Decode Table

```
int code=fetch_compressed_bits(bit_index);
```



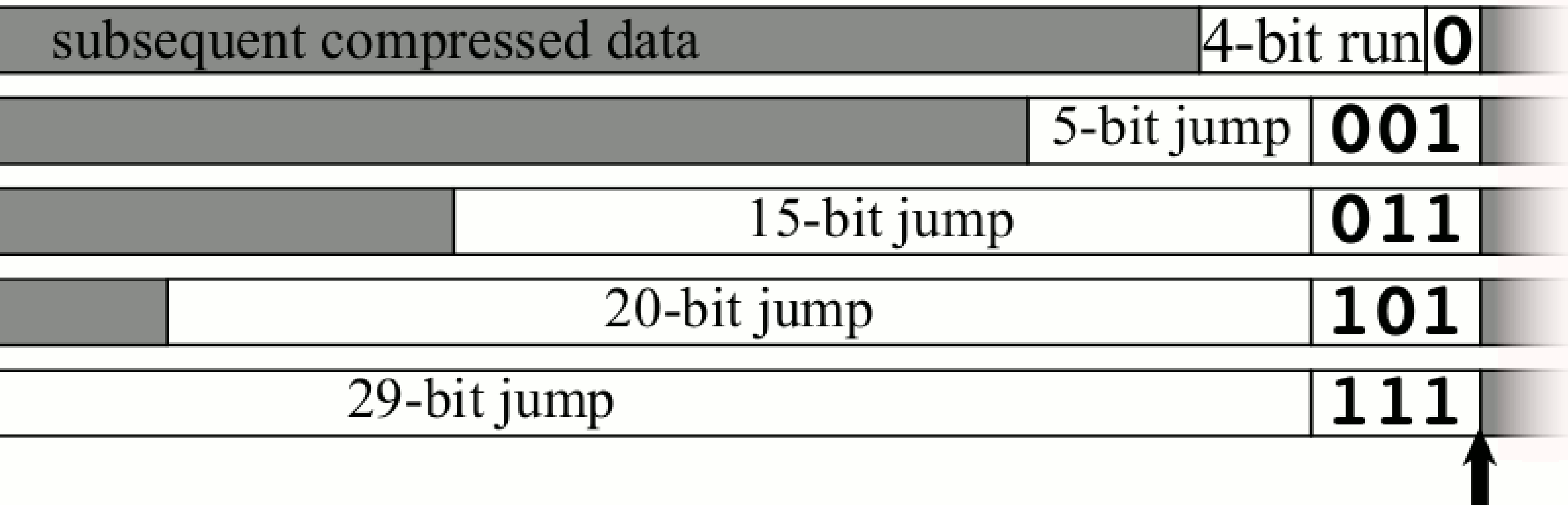
Running Fast: Decode Table

```
int code=fetch_compressed_bits(bit_index);
const table_entry_t &T=table[code&opcode_mask];
```

t run 0
001
011
101
111

Running Fast: Decode Table

```
int code=fetch_compressed_bits(bit_index);
const table_entry_t &T=table[code&opcode_mask];
bit_index+=T.count; // move data pointer
column+=T.mask & (code>>T.shift);  // extract column
```

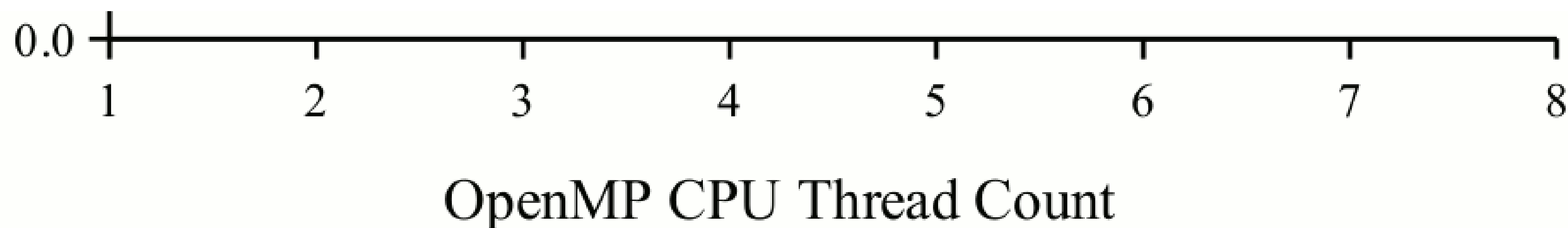


Multicore Performance

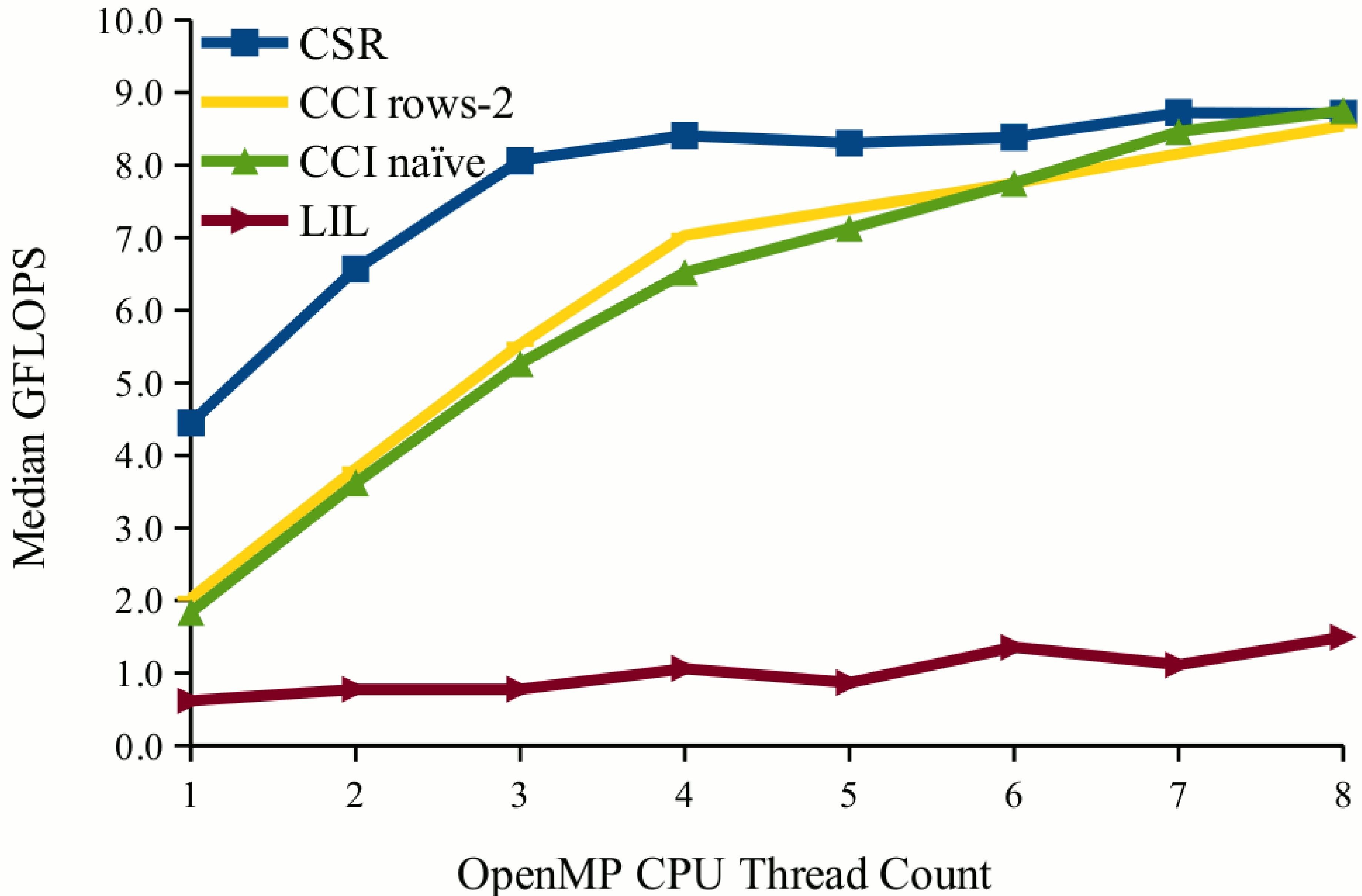
General Multicore Scaling Rule

Arithmetic Limited

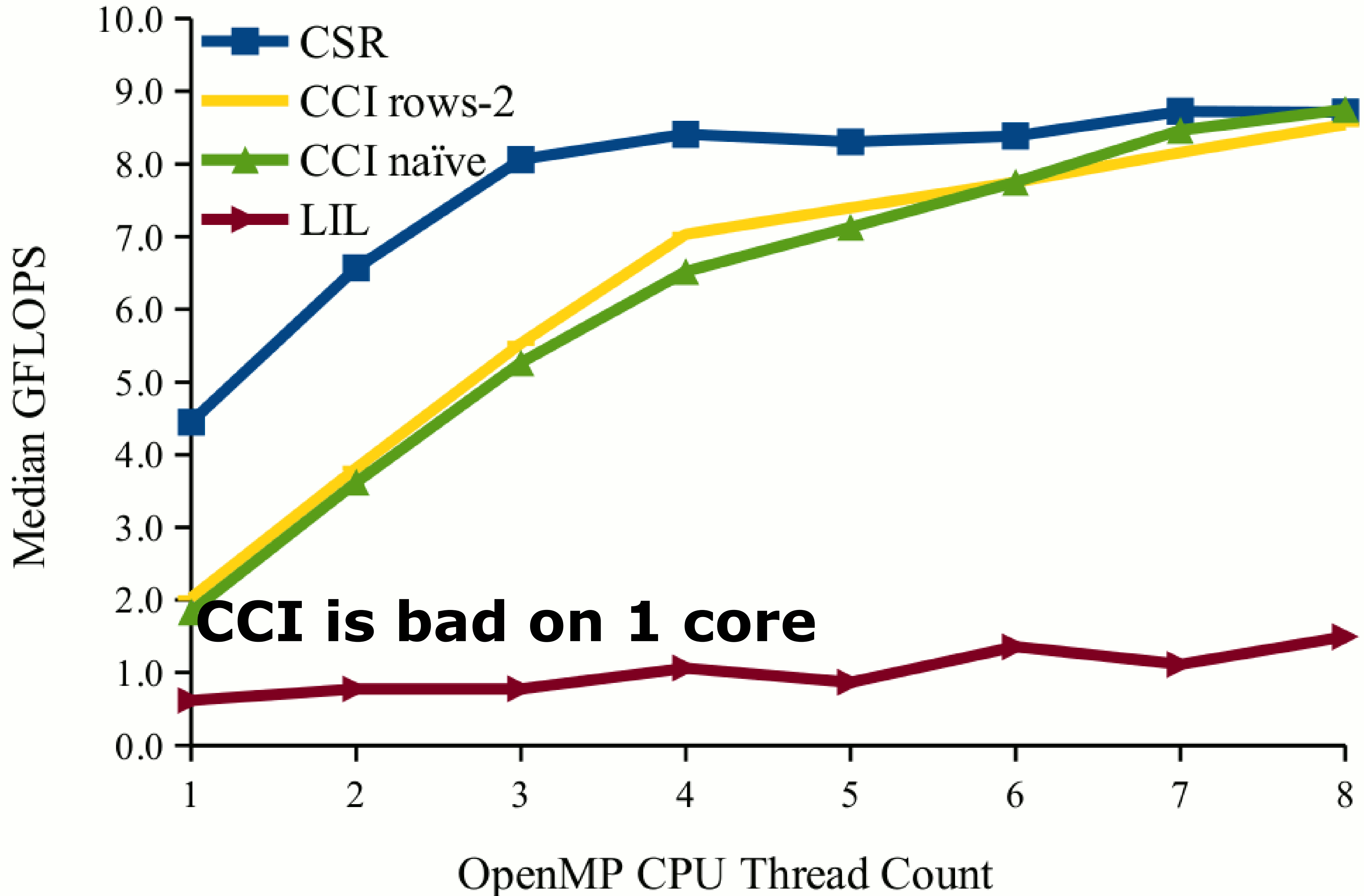
Bandwidth Limited



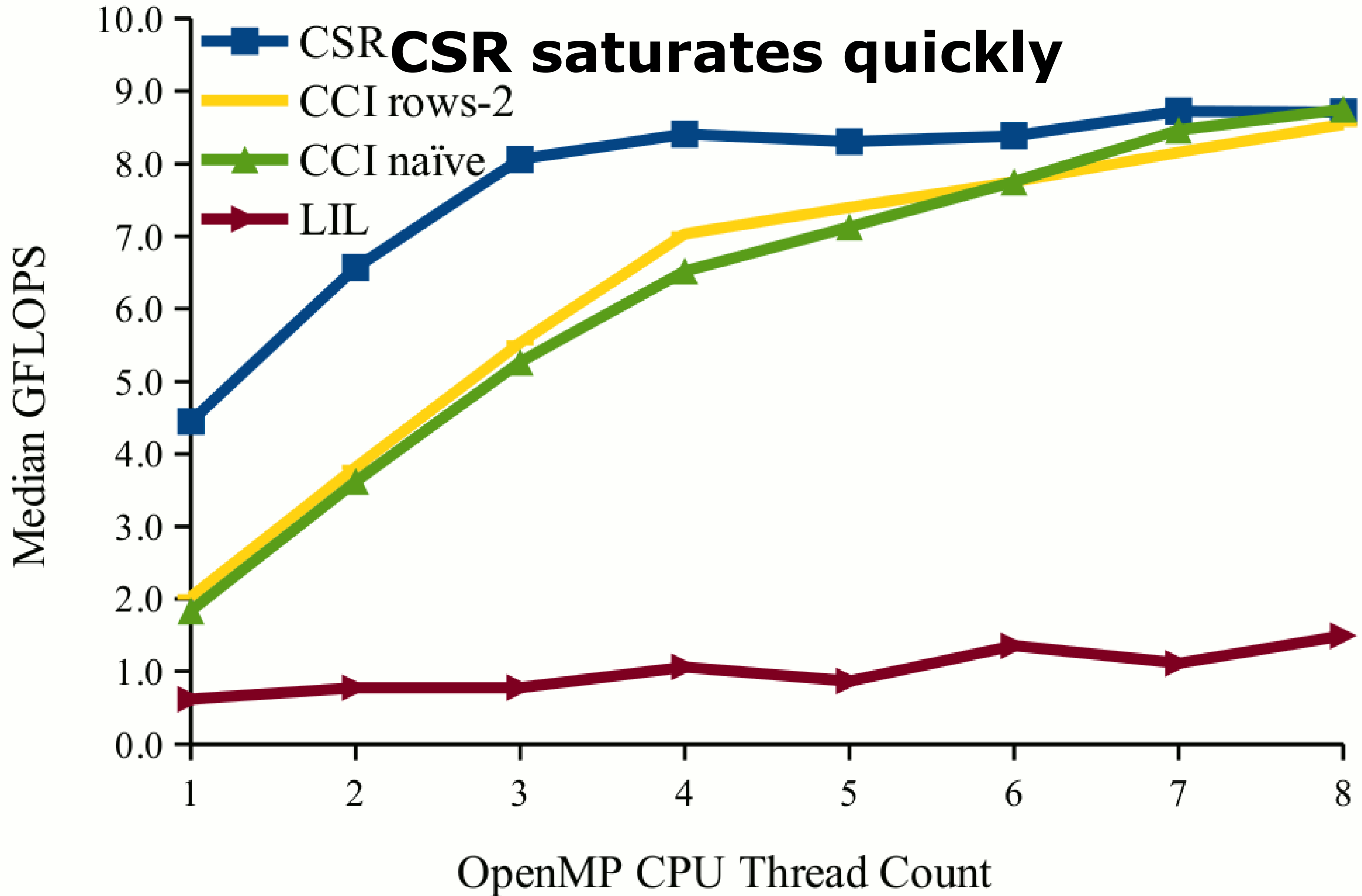
Sparse Matrix Multicore Scaling



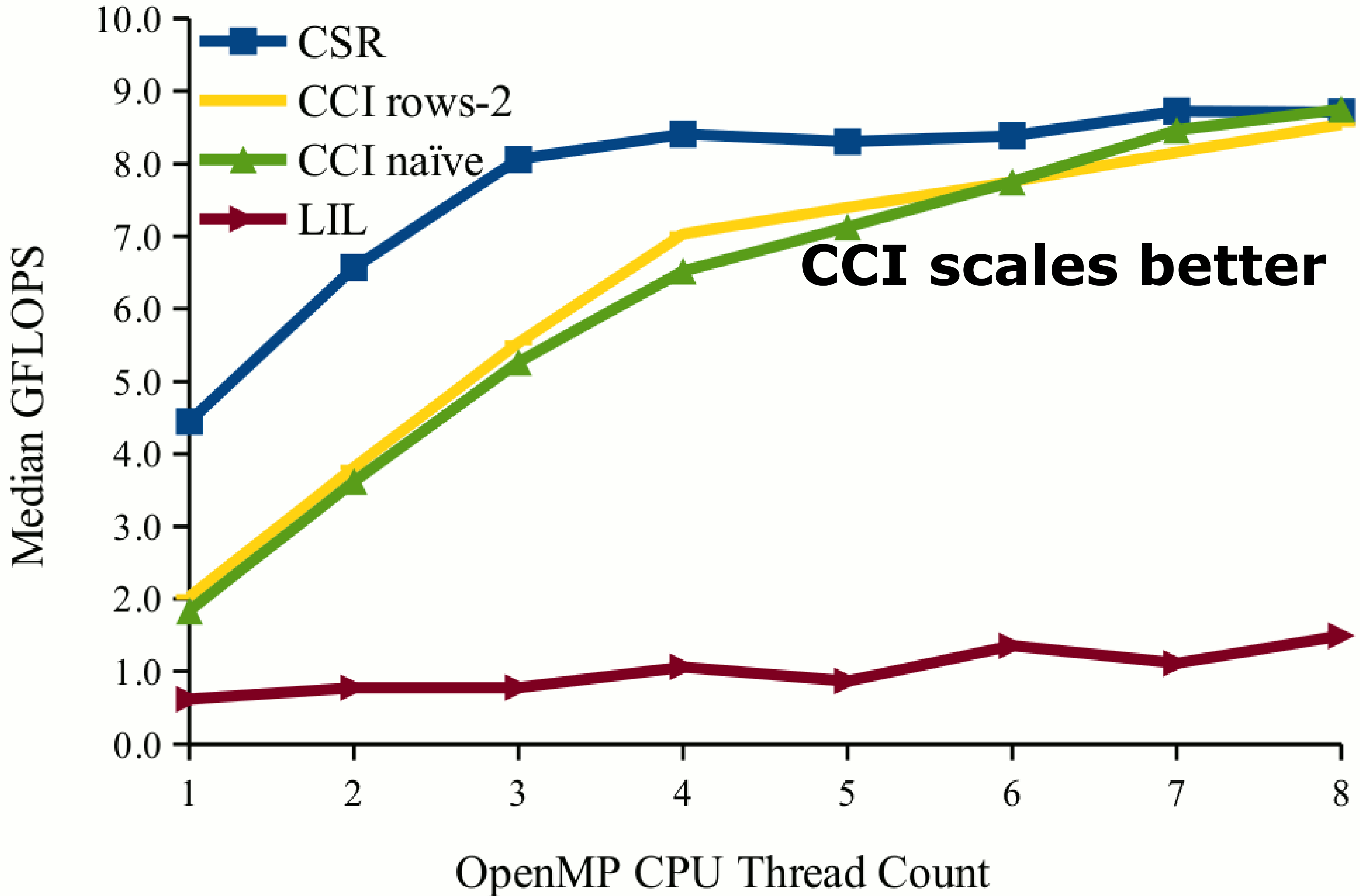
Sparse Matrix Multicore Scaling



Sparse Matrix Multicore Scaling



Sparse Matrix Multicore Scaling



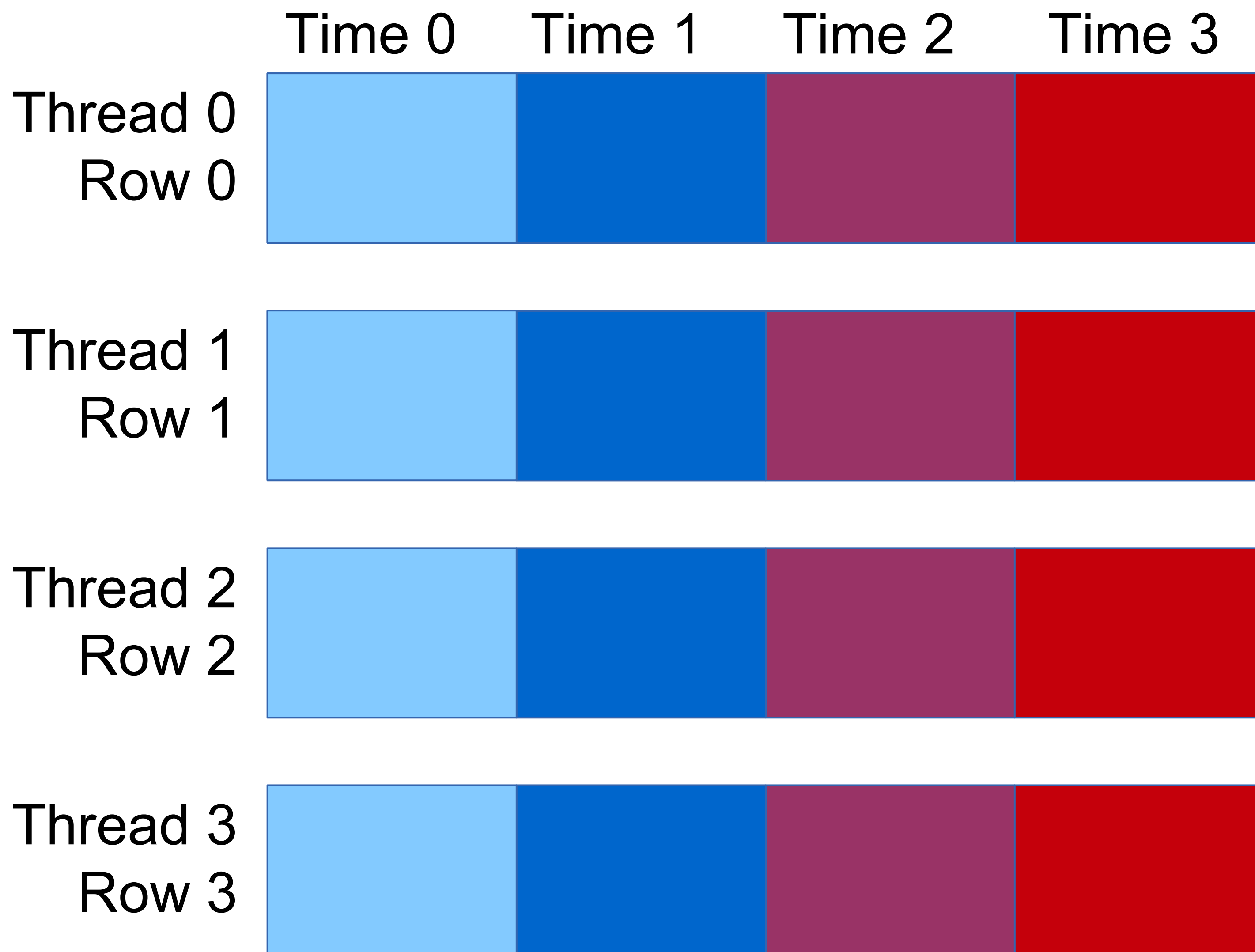
GPU Performance

Prior: CSR on GPU, in CUDA

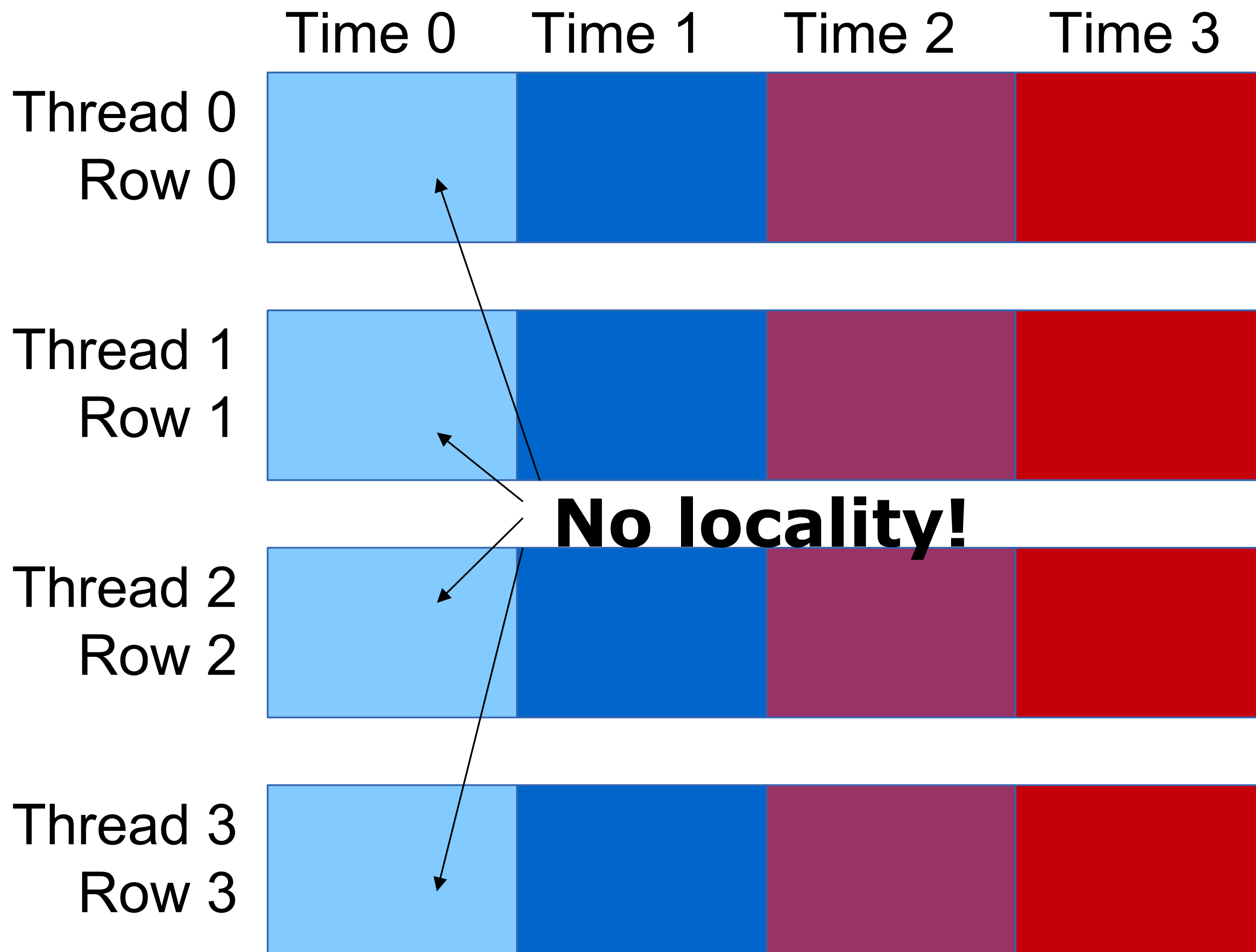
```
__global__ void csr_kernel(const real *src,real *dest,
     const real *vals,const int *cols,const int *start)
{
  int r=threadIdx.x+blockIdx.x*blockDim.x;
  if (r>=max_row) return;
  real sum=0.0;
  for (int i=start[r];i<start[r+1];++i)
    sum+=vals[i]*src[ cols[i] ];
  dest[r]=sum;
}
```

- **Naive thread-per-row CUDA CSR implementation is about 2GF**
  - **Slower than a single CPU core!**

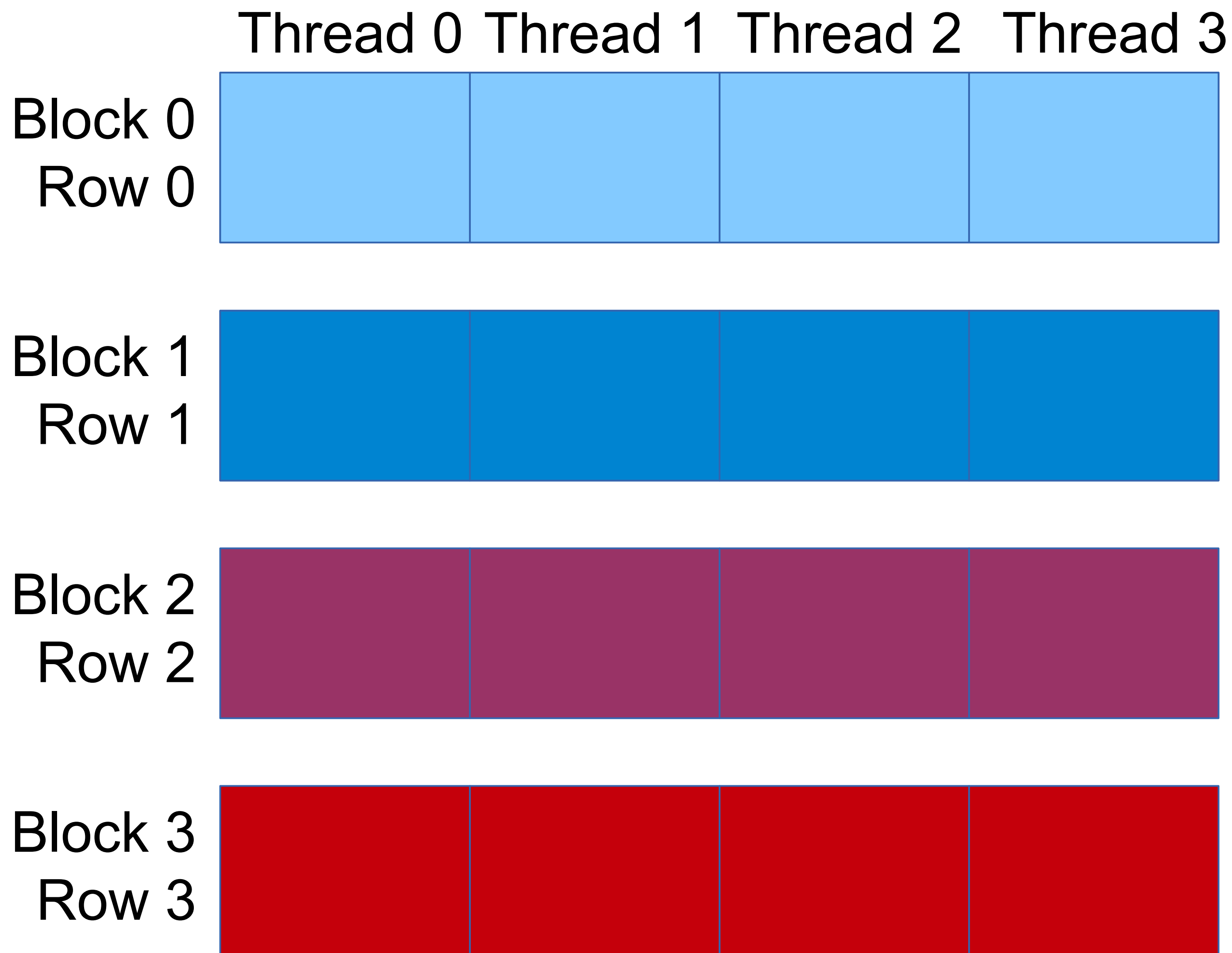
CSR on GPU: The Thread Problem



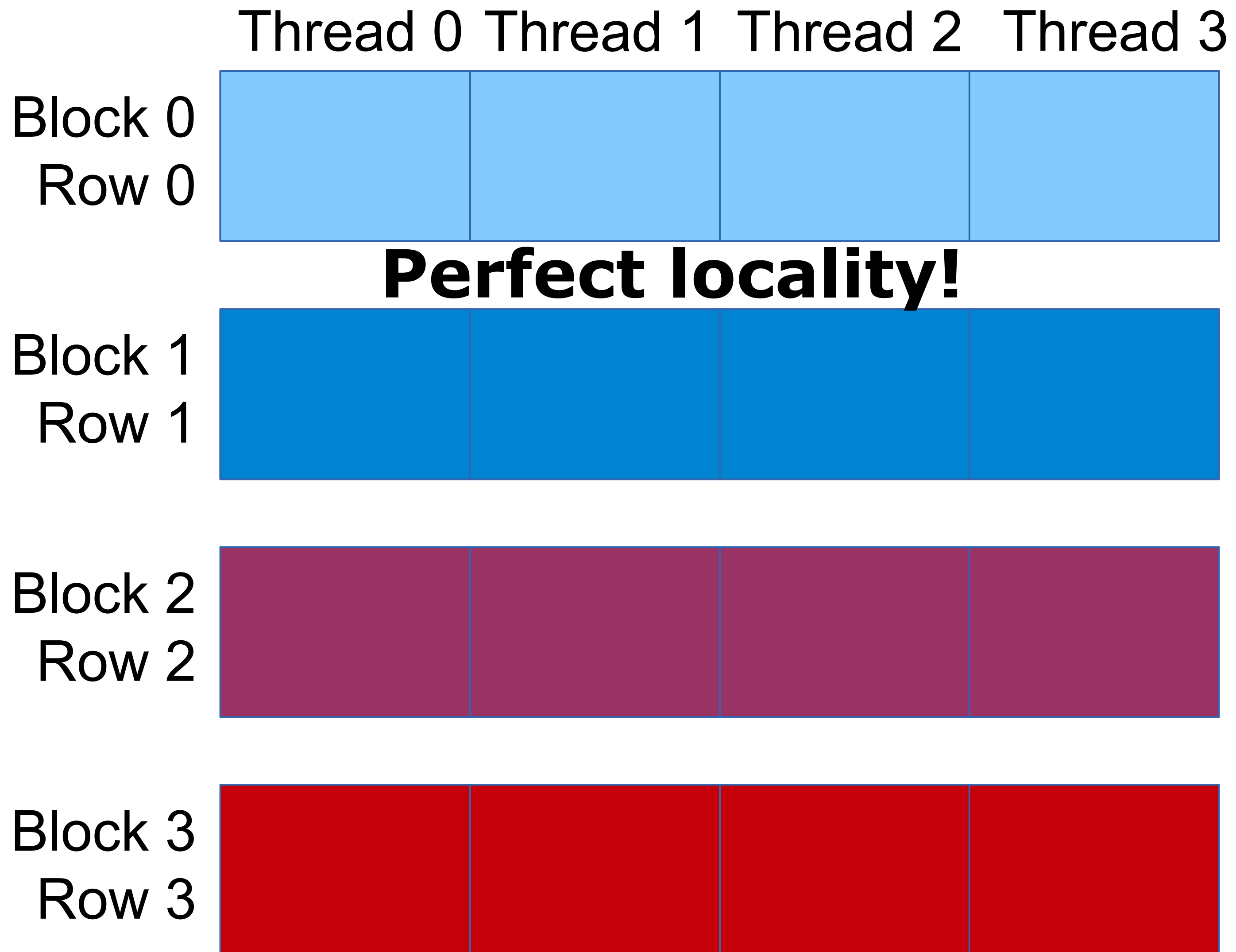
CSR on GPU: The Thread Problem



CSR on GPU: The “Stride” Fix



CSR on GPU: The “Stride” Fix

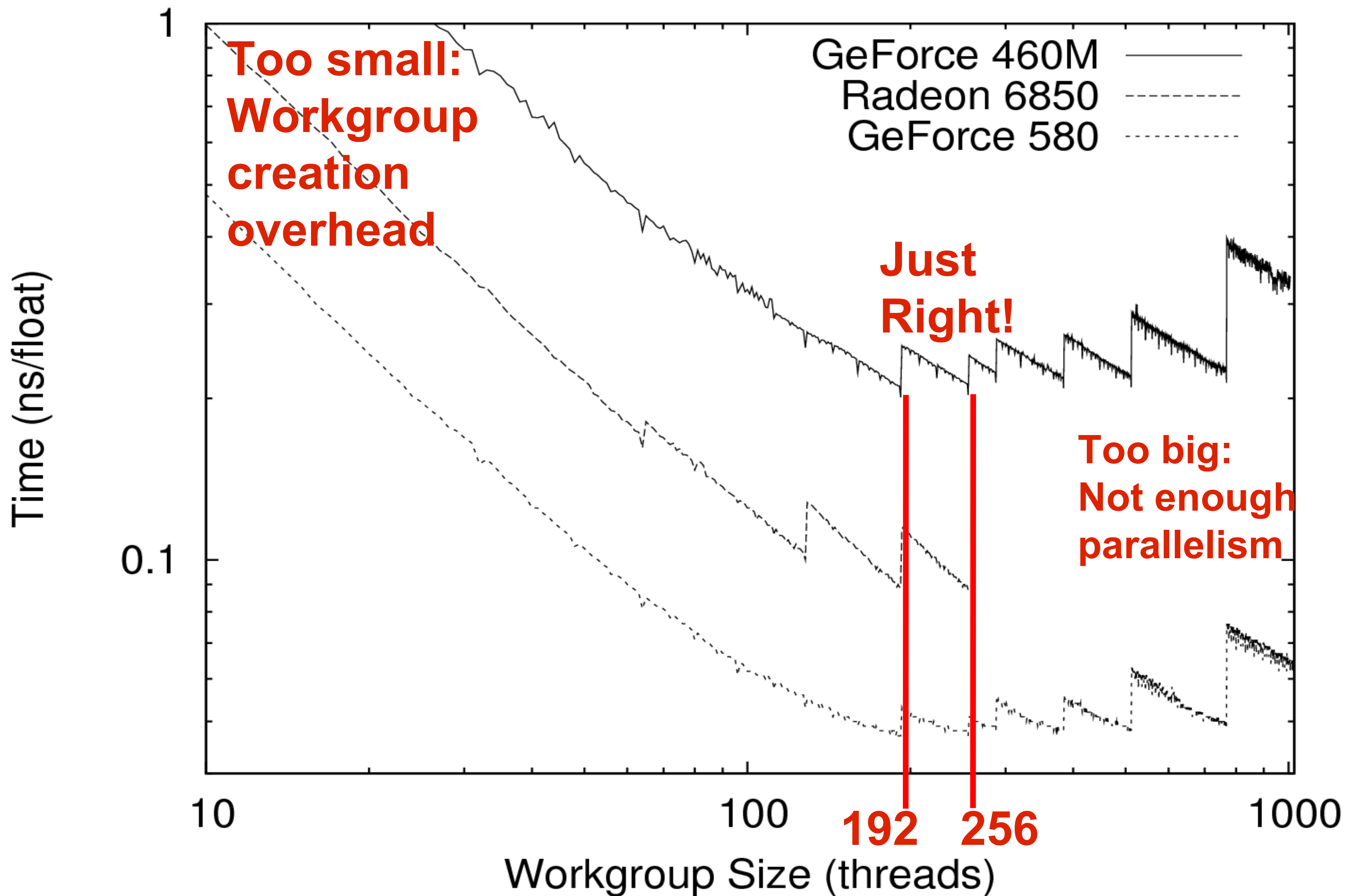


CSR on GPU: "Stride" Code

```
__global__ void csr_kernel(const real *src,real *dest,
     const real *vals,const int *cols,const int *start)
{
  int thread=threadIdx.x+blockIdx.x*blockDim.x;
  int r=thread/stride;
  if (r>=max_row) return;
  real sum=0.0;
  for (int i=start[r]+(thread%stride);i<start[r+1];i+=stride)
    sum+=vals[i]*src[ cols[i] ];
  shared_reduce(&sum,stride);
  dest[r]=sum;
}
```

- **Bell and Garland [2008] technique**
  - **In practice, stride=8 threads/row**

GPU: Workgroup Size vs Time



Conclusions

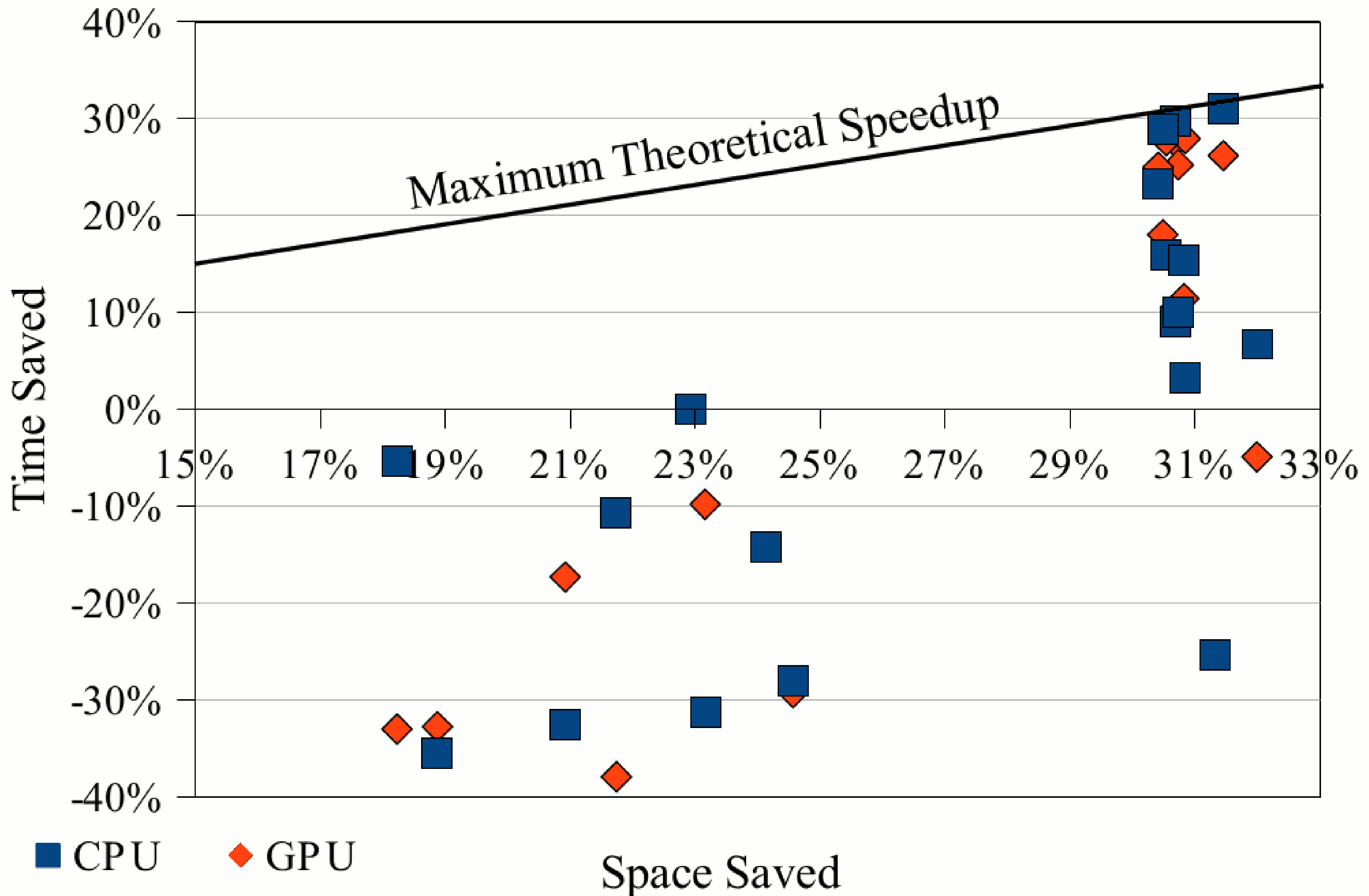
Performance Numbers

- **Matrix: AF_shell10, 1.5m rows**
  - **CSR: 206.7 MB**
  - **CCI: 18.2 MB (saved 91%)**
  - **Total read bandwidth saved: 30%**

- **Multicore: Core i7-3770K**
  - **CCI runs at 9.6 GF: 11ms per SpMV**
  - **23% faster than CSR via Intel MKL 11**

- **GPU: NVIDIA GeForce 570**
  - **CCI runs at 18.8 GF: 5.5ms per SpMV**
  - **25% faster than CSR via CUSPARSE 4.2**

CCI Beats CSR For Good Matrices



The Future: Memory Bandwidth

- **Today: >1TF/s, but only 0.1TB/s**
- **Don't communicate, <u>recompute</u>**
  - **Compiler+runtime decides when**
- **64-bit -> 32 -> 16 -> 8 -> 4 ...**
  - **Spend arithmetic on decompression**
  - **Split solution + residual storage**
    - **Most flops use fewer bits, in residual**
  - **Fight roundoff with stochastic rounding**
    - **Add noise to improve precision**

Conclusions

- **Fight bandwidth, not arithmetic**
  - **Save bandwidth, save time**
  - **Saves energy, too!**
- **CPU and GPU performance are similar**
  - **But GPU needs strided access**
  - **SIMD for CPU is explicit**
- **In-memory data compression is feasible today, at over 46GB/s**

http://tinyurl.com/lawlorCCI