

Research Paper Presentation #1

~

Michael M. Folkerts

CSE 262

Spring 2011

Research Paper Presentation

JW Choi, A Sing, RW Vuduc

Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs

Proc 15th ACM SIGPLAN symposium on Principles and practice of
parallel programming (PPoPP 2010), pp 115-126

Outline

- Introduction
- CSR and BCSR
- Blocked ELLPACK
 - Implementation
 - Results
- Autotuning Model
 - Computation Estimate
 - Occupancy
 - Results

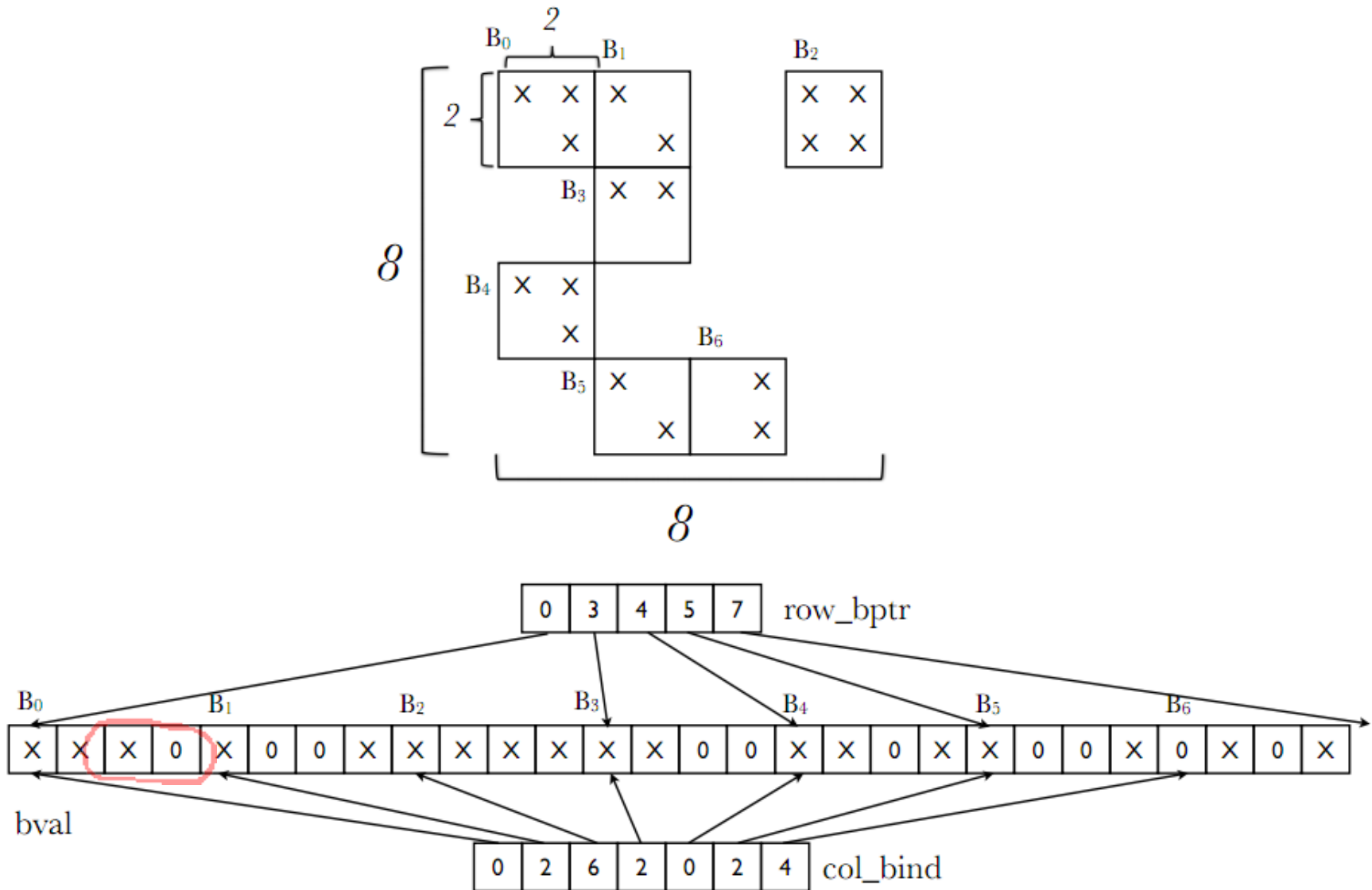
Introduction

- Motivation for GPU study
 - SpMV multiplication is memory bandwidth intensive
 - GPUs have significantly higher memory bandwidth than CPUs
- Obvious Issues
 - SpMV is irregular in memory access patterns
 - SpMV requires extra (memory) arrays to track indices

CSR and BCSR

- Compressed Sparse Row (CSR)
 - $m \times n$ sparse matrix w/ k non-zero entries
 - Stored in 3 arrays:
 - `val[k]` and `col_ind[k]` store and index non-zeros
 - `row_ptr[m+1]` store starting pointer to each row
- Blocked CSR (BCSR)
 - Sub-divide matrix into $r \times c$ blocks
 - Pad with zeros as needed
 - Reduces column index storage by at least $1/(r \times c)$
 - Only one index per block

BCSR



BCSR SpMV Implementation

Algorithm 1: 2×2 BCSR kernel to compute $y \leftarrow y + A \cdot x$

Input: $m \times n$ matrix A , stored in BCSR($r \times c$) format as
(bval, col_bind, row_bptr);
vectors x as $x[1 \dots n]$, y as $y[1 \dots m]$

Output: Modifies y

Let TB = thread block size (1-D)

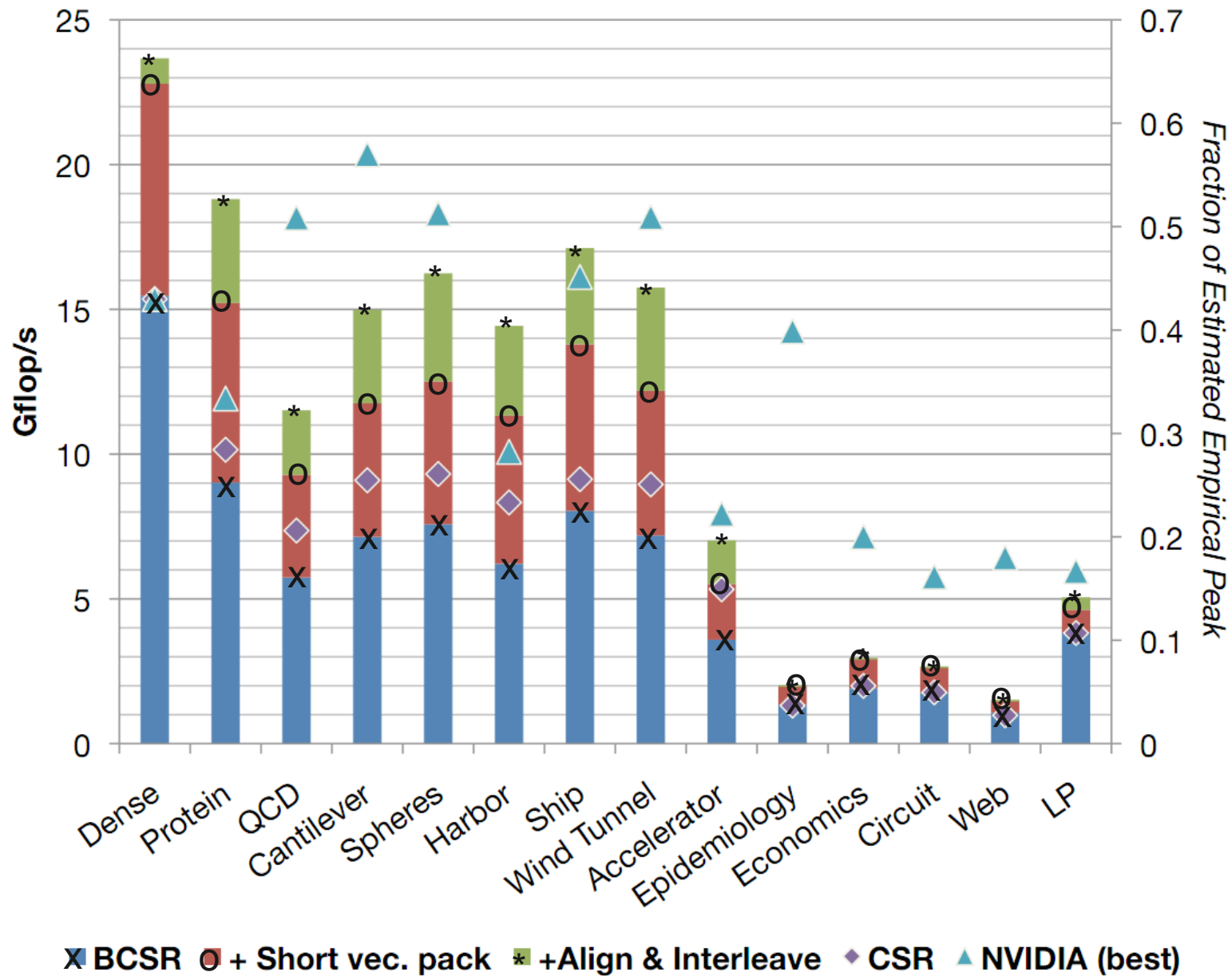
Let tid = local thread ID

Initialize sdata[TB][2]

```
1 for each block row,  $I$  do
2   row_start = row_bptr[ $I$ ]
3   row_end = row_bptr[ $I + 1$ ]
4   for  $k = \text{row\_bptr}[I]; k < \text{row\_bptr}[I + 1]; k = k + \text{TB}$  do
5     for each thread do
6        $j_0 = \text{col\_bind}[k + \text{tid}]$ 
7       float4 tmp = bval[ $k + \text{tid}$ ]
8       sdata[tid][0] += tmp.x  $\times$  x[ $j_0$ ] + tmp.y  $\times$ 
          x[ $j_0 + 1$ ]
9       sdata[tid][1] += tmp.z  $\times$  x[ $j_0$ ] + tmp.w  $\times$ 
          x[ $j_0 + 1$ ]
10    parallel reduction in shared memory to sdata[0][0]
11    parallel reduction in shared memory to sdata[0][1]
12     $Y[I \times 2] += \text{sdata}[0][0]$ 
13     $Y[I \times 2 + 1] += \text{sdata}[0][1]$ 
```

- Extends 'state of the art' CSR SpMV [3]
 - Thread handles row of matrix sub-blocks
- Short Vector Packing
 - Load sub-block as 'float4'
- Row Alignment
 - Pad short sub-block rows with zeros

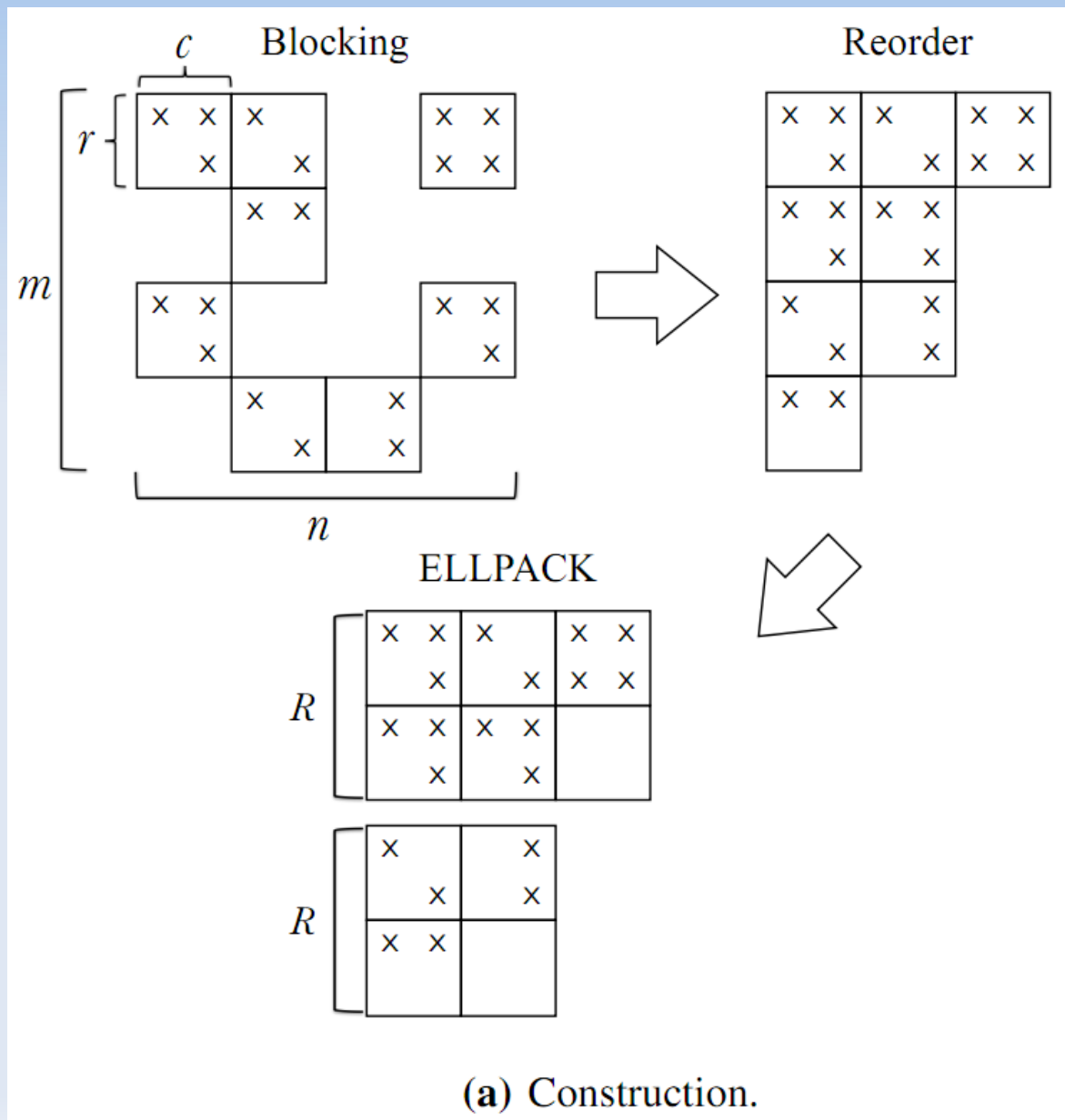
BCSR SpMV Results



ELLPACK and BELLPACK

- ELLPACK
 - $m \times n$ sparse matrix w/ k non-zero entries
 - Stored in 2 arrays:
 - $V[m][L]$ and $J[m][L]$ store and index non-zeros
 - L is maximum number of non-zeros in a row
- Blocked ELLPACK (BELLPACK)
 - Sub-divide matrix into $r \times c$ blocks
 - Pad with zeros as needed
 - Resort rows - descending order of number of blocks
 - Group block rows in sets of size R

BELLPACK Summary



BELLPACK Implementation

Algorithm 2: 2×2 BELLPACK kernel to compute $y \leftarrow y + A \cdot x$

Input: array of arrays *Ad_values* for values and *Ad_index* for indices, array *Ad_numBlocks* for block sizes, vector *x* as $x[1 \dots n]$, and blocking sizes *r* and *c*

Output: Modifies *y*

Let *bid* = block ID

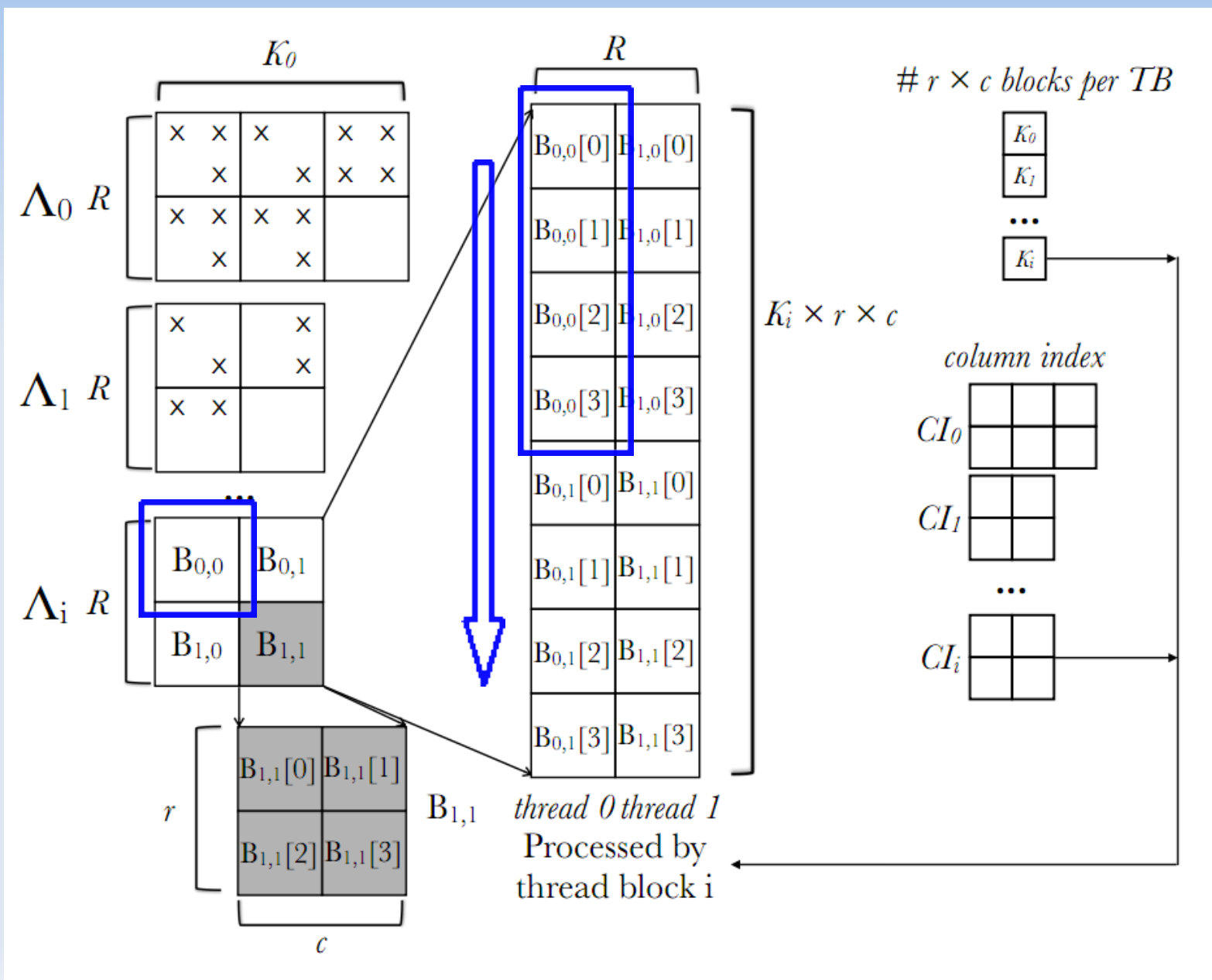
Let *tid* = local thread ID

Let *TB* = thread block size (1-D)

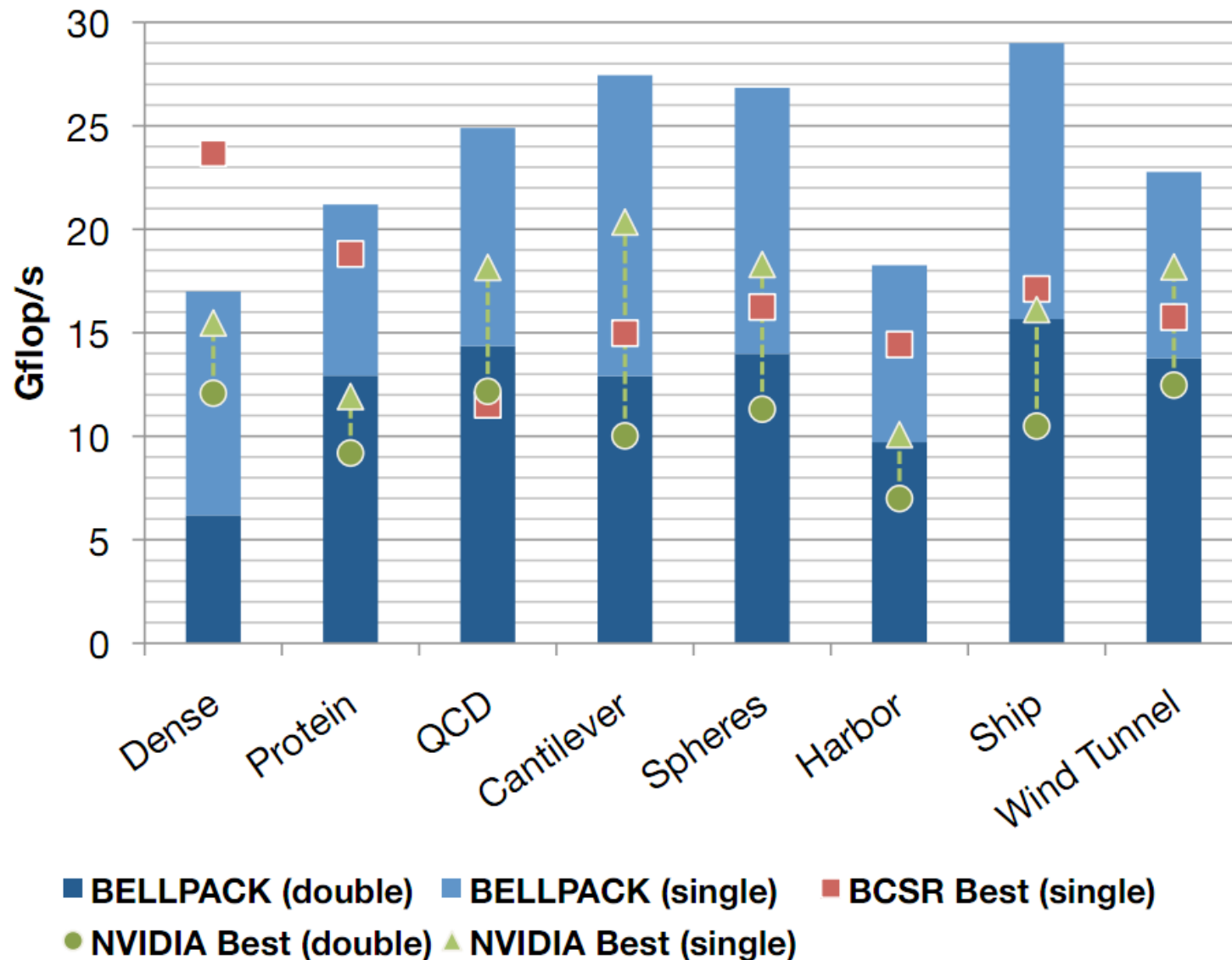
```
1 if tid == 0 then
2   numBlocks = Ad_numBlocks[bid]
3   A_values = Ad_values[bid]
4   A_index = Ad_index[bid]
5 for i = 0 to numBlocks do
6   col_Index = A_index[TB × i + tid] × c
7   xval = x[col_Index]
8   rs_1 += A_values[TB × 4 × i + tid] × xval
9   rs_2 += A_values[TB × 4 × i + TB × 2 + tid] × xval
10  xval = X[col_Index++]
11  rs_1 += A_values[TB × 4 × i + TB × 1 + tid] × xval
12  rs_2 += A_values[TB × 4 × i + TB × 3 + tid] × xval
13 y[TB × bid × 2 + tid × 2] = rs_1
14 y[TB × bid × 2 + tid × 2 + 1] = rs_2
```

- Each thread handles row in $R \times L$ sub-matrix (stored as column)
- *X* is stored as a texture and is therefore cached on chip

BELLPACK Detail



BELLPACK Results



(a) Performance on “naturally blocked” matrices

Auto-tuning Model

- Generic GPU Model

- For a workload L (thrad blocks): M (# of SMs)
process T (thread blocks) for I (iterations)

- $I = \text{ceil}(L / (T * M))$

- Time for each iteration (assume homogeneity):

$$t_i(k) = \sigma_i + \alpha_i \times (k - 1), \quad (3)$$

$$t = \tau_1 + (I - 2) \times \tau + \tau_I \quad (4)$$

$$\tau_1 = \sigma_1 + \alpha \times (T - 1) \quad (5)$$

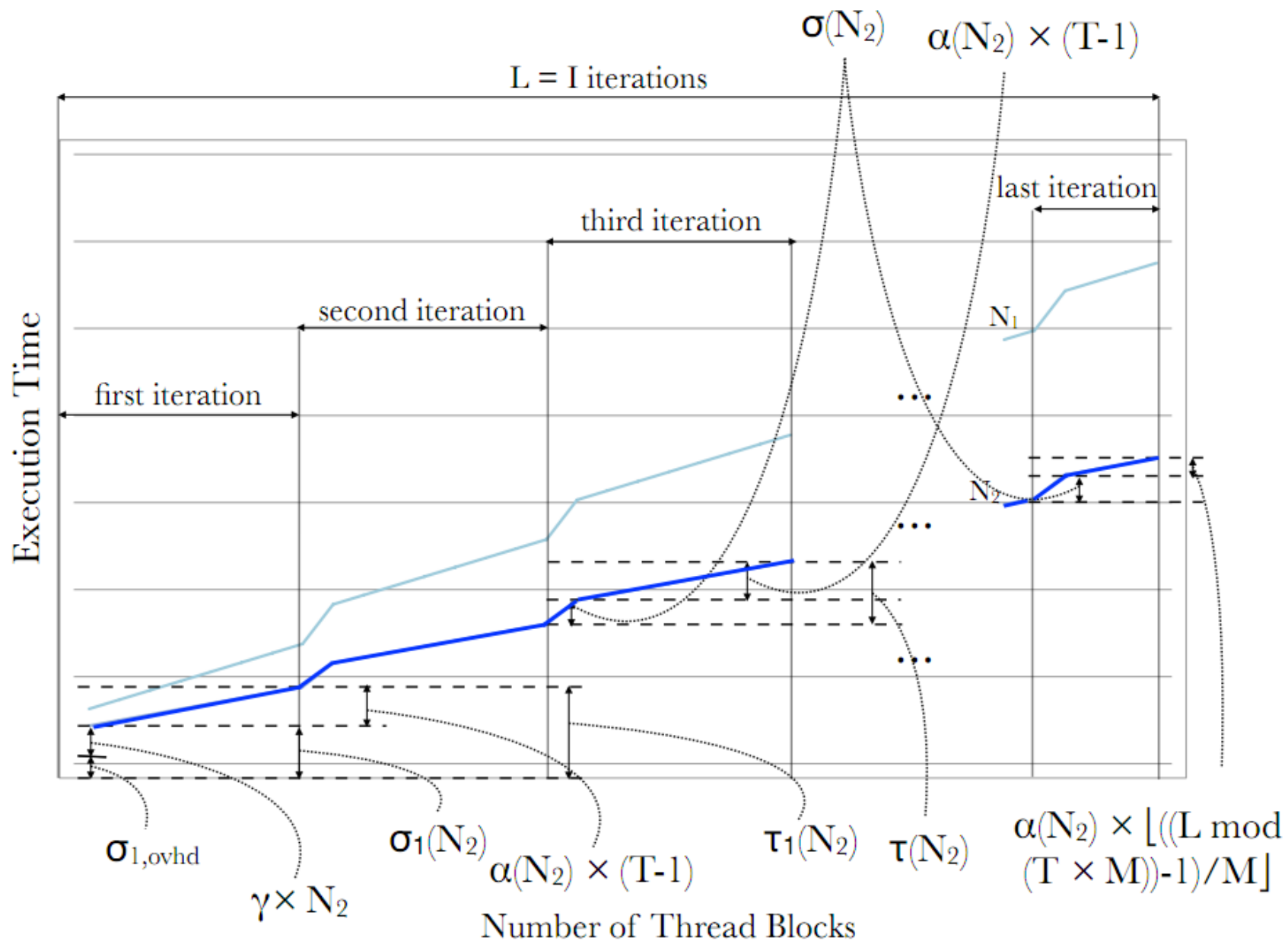
$$\tau = \sigma + \alpha \times (T - 1) \quad (6)$$

$$\tau_I = \sigma + \alpha \times \left\lfloor \frac{(L \bmod (T \times M)) - 1}{M} \right\rfloor. \quad (7)$$

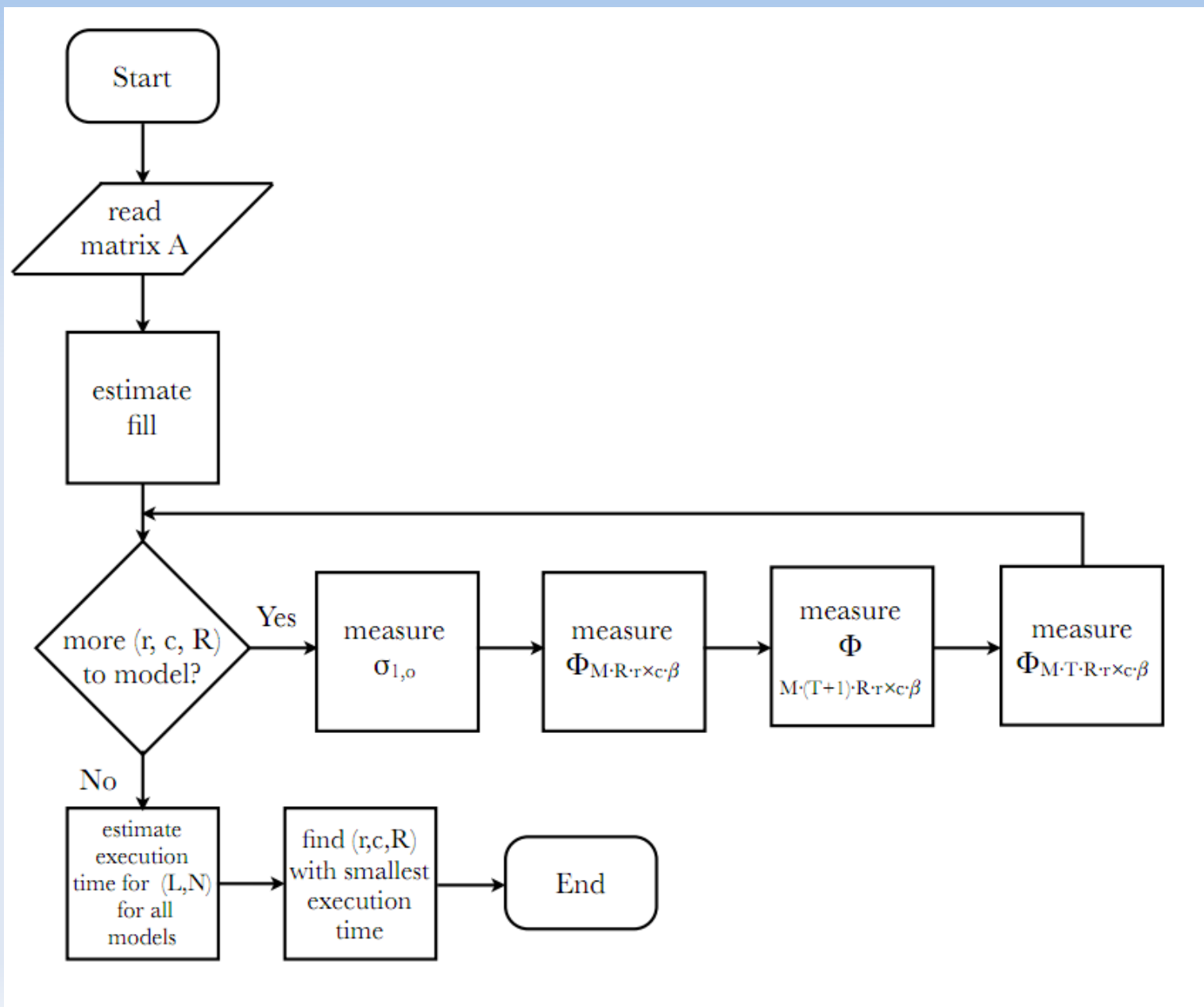
Thread Occupancy

- Computing T max (threads blocks per SM)
 - Computed using hardware specific properties
 - Register Memory
 - Shared Memory
 - Max # thread blocks/SM
 - ...
 - Exactly like 'occupancyCalculator.xls' provided by NVIDIA

GPU Model



Auto-Tuning



Results

