Efficient and Portable ALS Matrix Factorization for Recommender Systems

Jing Chen[1], Jianbin Fang[1], Weifeng Liu[2], Tao Tang[1], Xuhao Chen[1], Canqun Yang[1]

College of Computer, National University of Defense Technology [1]

Niels Bohr Institute, University of Copenhagen [2]

Contents

1.BACKGROUND

1.1 Recommender systems

System Goal :

build a **model**

train with observed imcomplete rating data;

and predict preference over items not rated.

Recommendation Approaches :

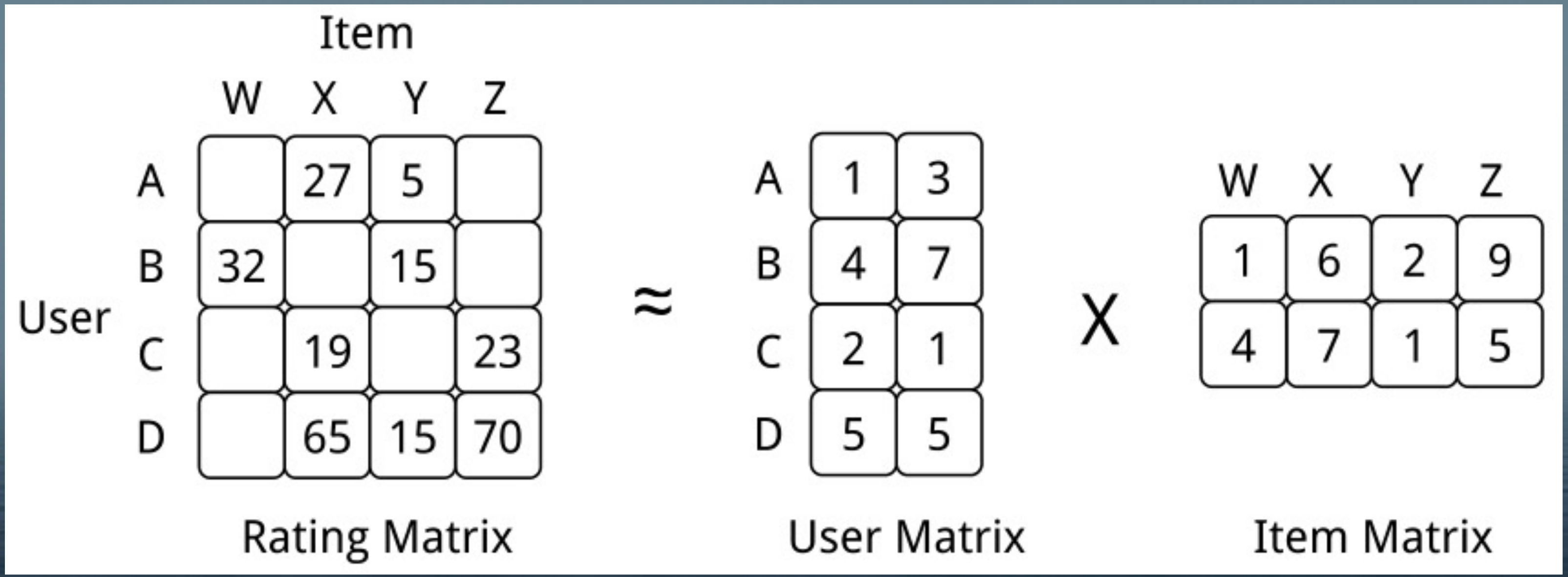
Matrix factorization (MF), nearest-neighbor ...

Popular Algorithms of Matrix Factorization :

ALS (Alternating least squares), SGD (Stochastic gradient descent), CCD (Cyclic coordinate descent) ...

1.2 Matrix Factorization

- Input : Rating matrix between users and items, $R(m \times n)$
- Output : $X(m \times k)$ matrix and $Y(n \times k)$ matrix so that $r_{ui} \approx x_u y_i^T$



1.2 Matrix Factorization

- Input : Relation matrix between users and items, R(m×n)
- Output : X(m×k) matrix and Y(n×k) matrix so that $r_{ui} \approx x_u y_i^T$

- minimize the **regularized squared error** to obtain X, Y

$$L(X,Y) = \sum_{u,i \in \Omega} (r_{ui} - x_u^T y_i)^2 + lambda(|x_u|^2 + |y_i|^2)$$

x_u^T : the u^th row vector of matrix X

y_i : the i^th column vector of matrix Y

Ω : all the nonzero ratings of matrix R

lambda : regularized coefficient (to avoid over-fitting)

1.3 ALS

Principle : to ***keep one fixed while calculating the other***

1. We minimize the equation over X while fixing Y , the function becomes,

$$\mathrm{L}(X) = \sum_{i \in \Omega_u} (r_{ui} - x_u^T y_i)^2 + \lambda |x_u|^2$$

2. Calculating derivative of x_u and let the partial derivative equal zero,

$$x_u = \left(Y^T Y + \lambda I\right)^{-1} Y^T r_u$$

3. In a same way, $y_i = \left(X^T X + \lambda I\right)^{-1} X^T r_i$

4. ALS *iterates* until it reaches the maximum specified cycles or error rate.

2.MOTIVATIONS

2.1 Motivations

Observation 1 :

ALS on CPUs runs faster than on GPUs.

- ✓ **11×** faster on the CPU than on the GPU.
- ✓ Restructure the algorithm
- ✓ Customize optimizations according to the architectural specifics.

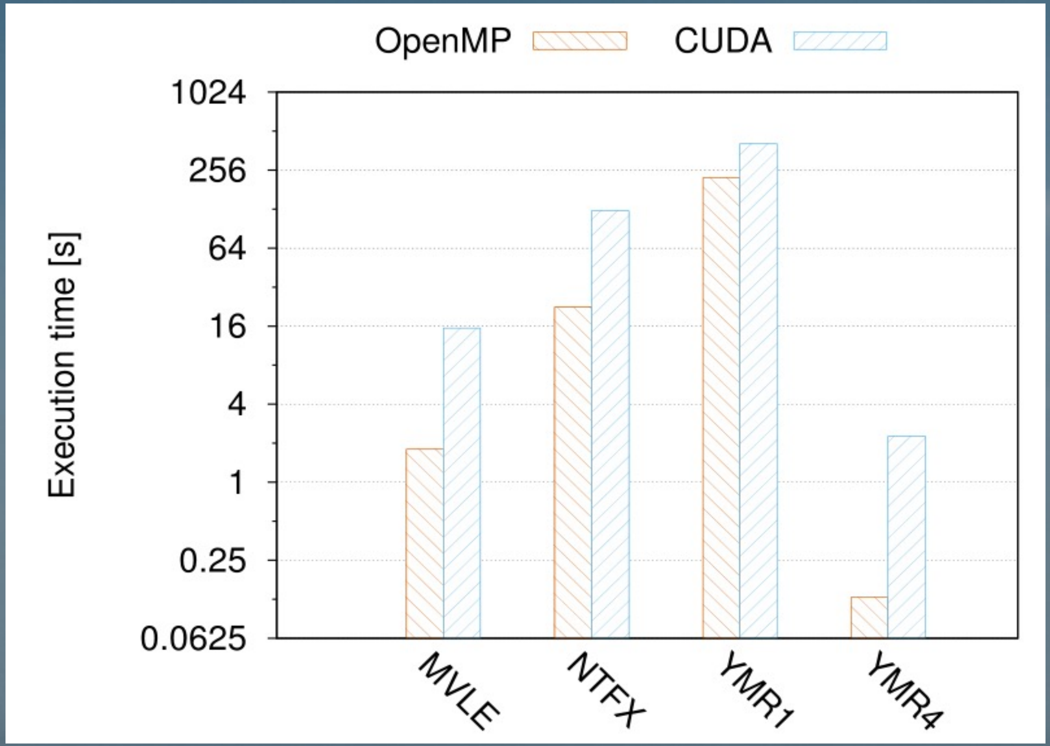


Fig 1. Performance comparison of an OpenMP implementation on a 16-core CPU versus a CUDA implementation on K20C.

2.2 Motivations

Observation 2 :

The current implementation cannot run on the coprocessors, such as Intel Xeon Phi.

Various Platforms : GPUs, MICs, FPGAs, DSPs ...

The current implementations cannot be offloaded to run on FPGAs.

Porting is time-consuming and error-prone.

- ✓ Speed
- ✓ Portability

3. DESIGN and IMPLEMENTATION

Thread Batching Parallelization

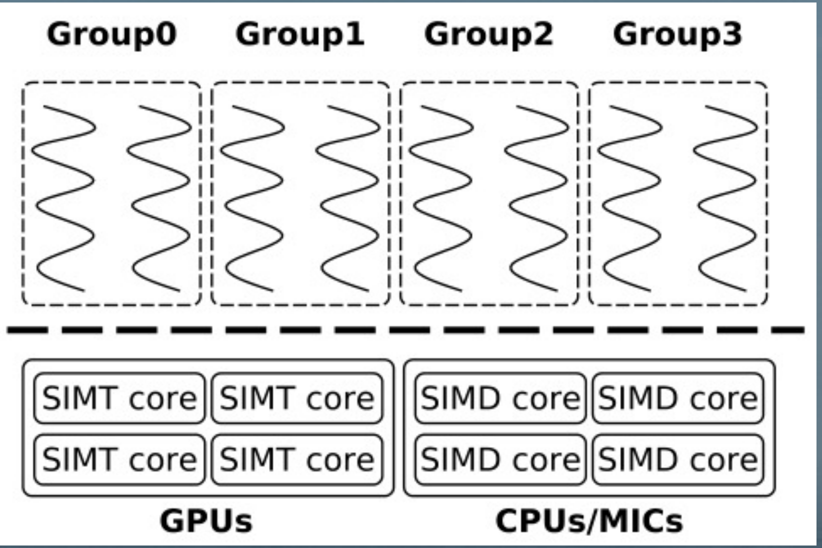
Baseline Design :

using ***<u>one thread</u>*** to update a row of X or a column of Y.

unaware of the hierarchical thread organization in CPU / GPU / MIC

Problem :

- *<u>unbalanced thread use</u>*
- *<u>random memory access</u>*



Solution :

- using thread batching technique
- wrap a branch of threads to deal with a row / column

Architecture-specific Optimizations

■ Using Registers

Modern GPUs feature abundant registers (small accessing latency).

Tesla K20C : 256KB registers in each SM.

```
float sum[k*k]={0};
for (int i = lx; i < k; i+=ws){
  for (int j = i; j < k; j++) {
    for (int z = 0; z < omegaSize; z++){
      int d = col_idx[row_ptr + z] * k;
      sum[i*k+j] += Y[d + i] * Y[d + j];
    }
    smat[(j*k)+i] = sum[i*k+j];
    smat[(i*k)+j] = sum[i*k+j];
  }
}
```

(a) Original code.

```
float sum0=0,sum1=0,sum2=0,sum3=0,sum4=0;
for (int z = 0; z < omegaSize; z++){
  int d = col_idx[row_ptr + z] * k;
  if(0<=lx<k) sum0 += Y[d + lx] * Y[d + 0];
  if(1<=lx<k) sum1 += Y[d + lx] * Y[d + 1];
  if(2<=lx<k) sum2 += Y[d + lx] * Y[d + 2];
  if(3<=lx<k) sum3 += Y[d + lx] * Y[d + 3];
  if(4<=lx<k) sum4 += Y[d + lx] * Y[d + 4];
  ...
}
// updating the smat matrix
```

(b) Unrolling the code.

Figure 2. An example of unrolling the code to calculate Y^TY, where k=5.

Original version: private array sum[**k*k**] for each thread

Unrolling version: **k registers** for each thread block are enough

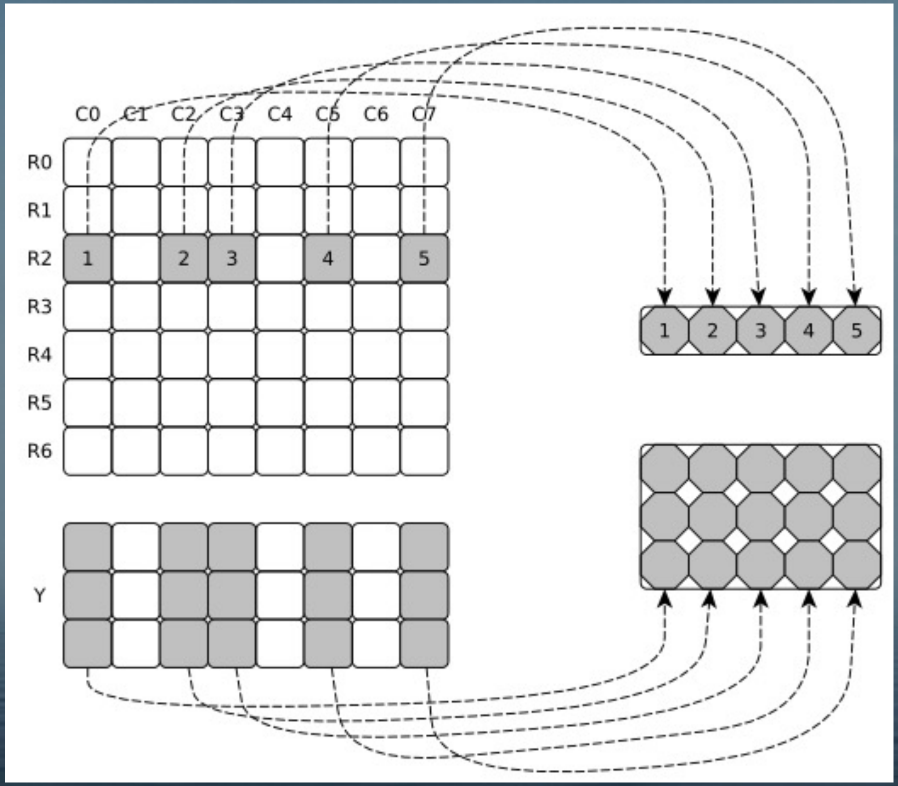
Architecture-specific Optimizations

■ Using Scratch-pad memory (Local memory in OpenCL)

- is a high-speed memory unit located on-chip
- *data sharing* in a same thread block
- increase *data moving bandwidth* between off-chip and on-chip memory

Sparsity of R matrix

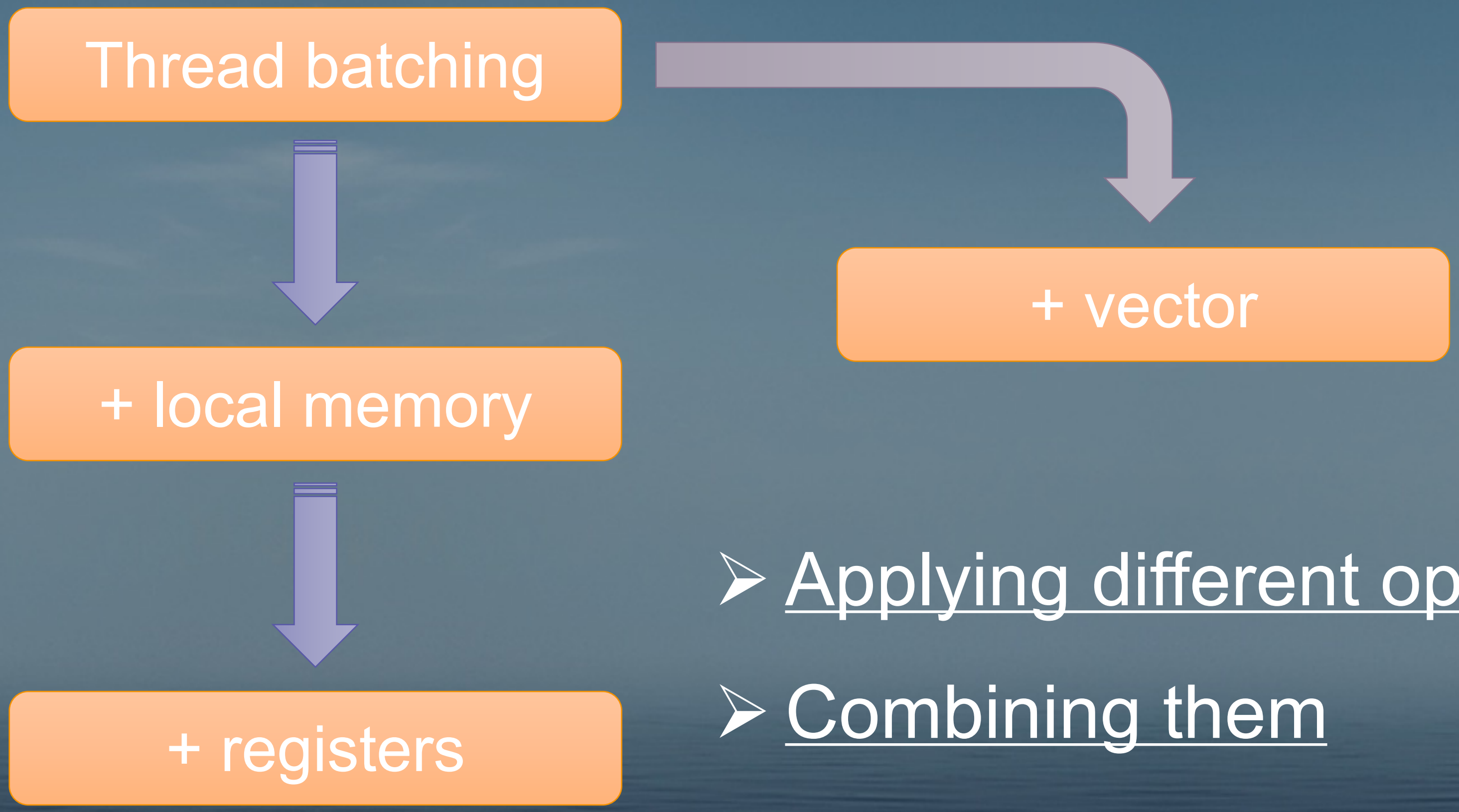
Incontiguous Data



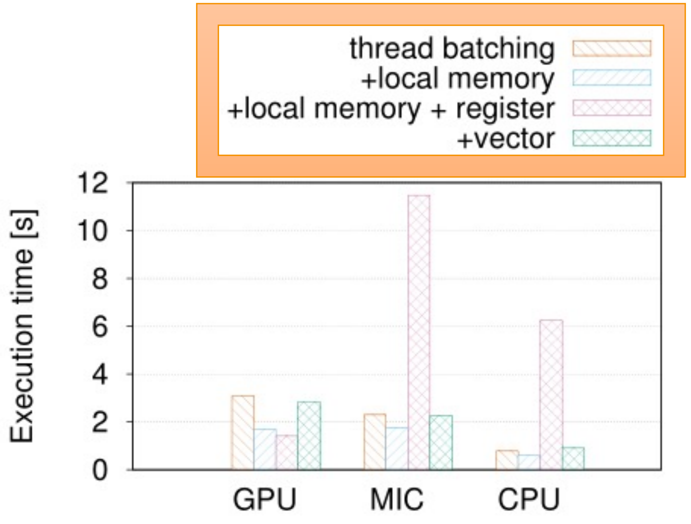
■ Using Vector Units

- ➢ CPU
- ➢ MIC

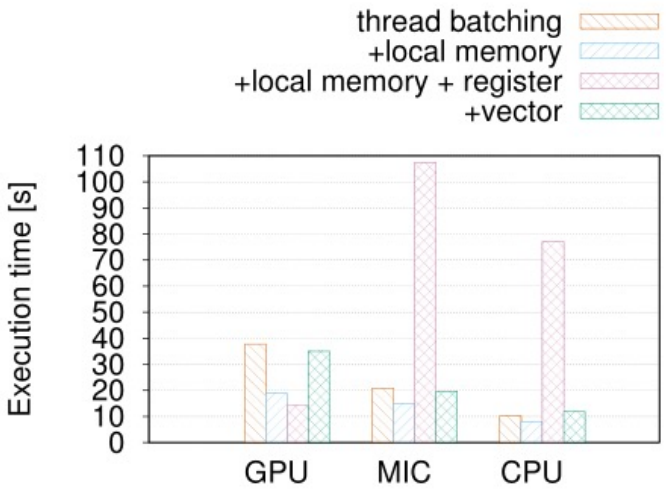
Code Variant Selection



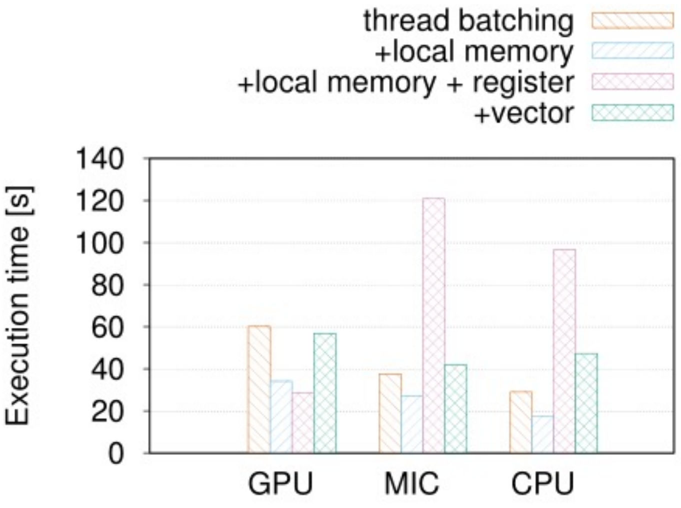
- <u>Applying different optimizations</u>
- <u>Combining them</u>



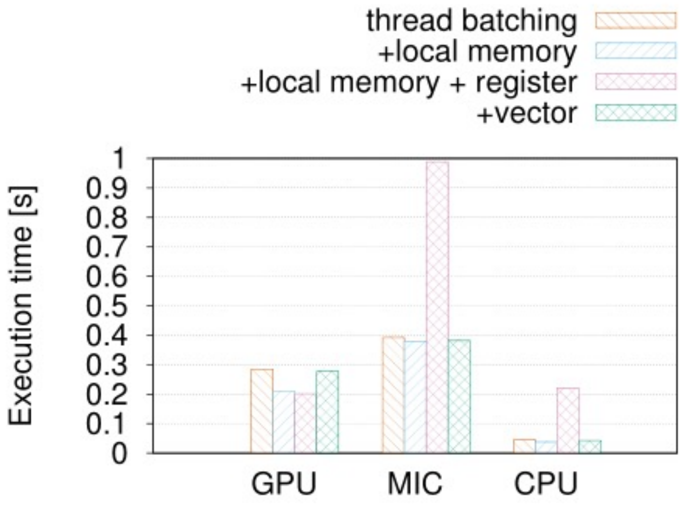
(a) Movielens



(b) Netflix



(c) YahooMusic R1



(d) YahooMusic R4

Goal:

Select the most appropriate implementations for a specific execution context (i.e. target architectures and input datasets).

4.SETUP and DATASET

Platform Configurations

- ✓ Intel Xeon E5-2670 (CPU): 16 cores
- ✓ NVIDIA Tesla K20c (GPU): 13 SM, 192 CUDA cores in each SM
- ✓ Intel Xeon Phi 31SP (MIC): 57 cores, 6GB global memory
- ✓ OpenCL (version 1.2)
- ✓ Host CPU: Redhat (v7.0)   GCC(v4.9.2)

Input Datasets

- **Format of datasets**

  <user ID, item ID, rating>

	Abbr.	**m**	**n**	**Training N_z**
Movielens10M	MVLE	71567	65133	8000044
Netflix	NTFX	480189	17770	99072112
YahooMusic R1	YMR1	1948882	98212	115248575
YahooMusic R4	YMR4	7642	11916	211231

5. PERFORMANCE RESULTS

5.1 Compare with the State-of-the-art

- vs **SAC15**
  - 5.5× faster than OpenMP
  - 21.2× faster on K20c GPU
- vs **HPDC16** (CuMF) 2.2× ~ 6.8×

Using **cusparse** library

usage of thread batching

archi-specific optimizations

highly tune

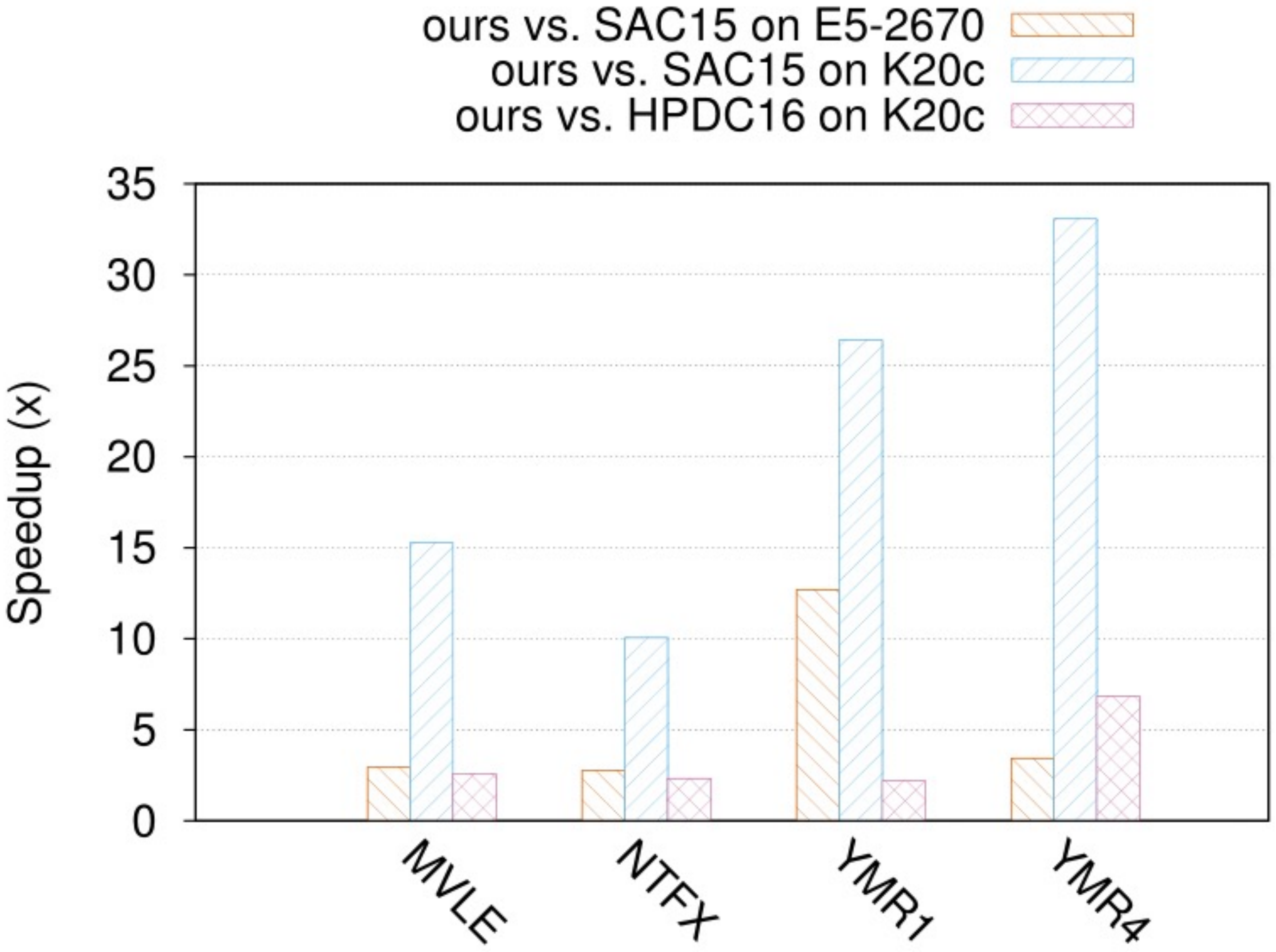
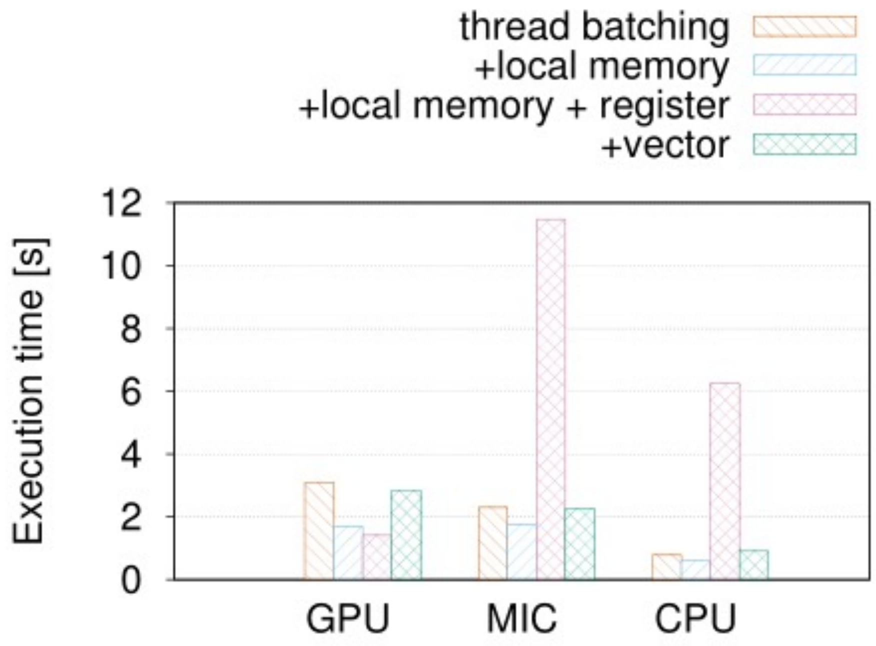
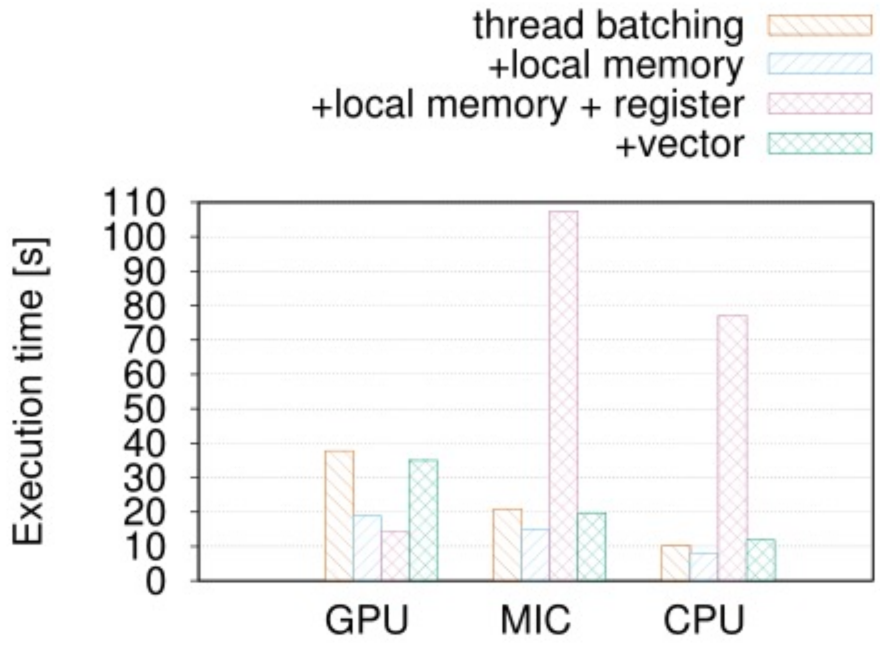


Figure 3. A performance comparison of our implementation versus the state-of-the-art implementations.

5.2 Evaluate Optimizations



(a) Movielens



(b) Netflix

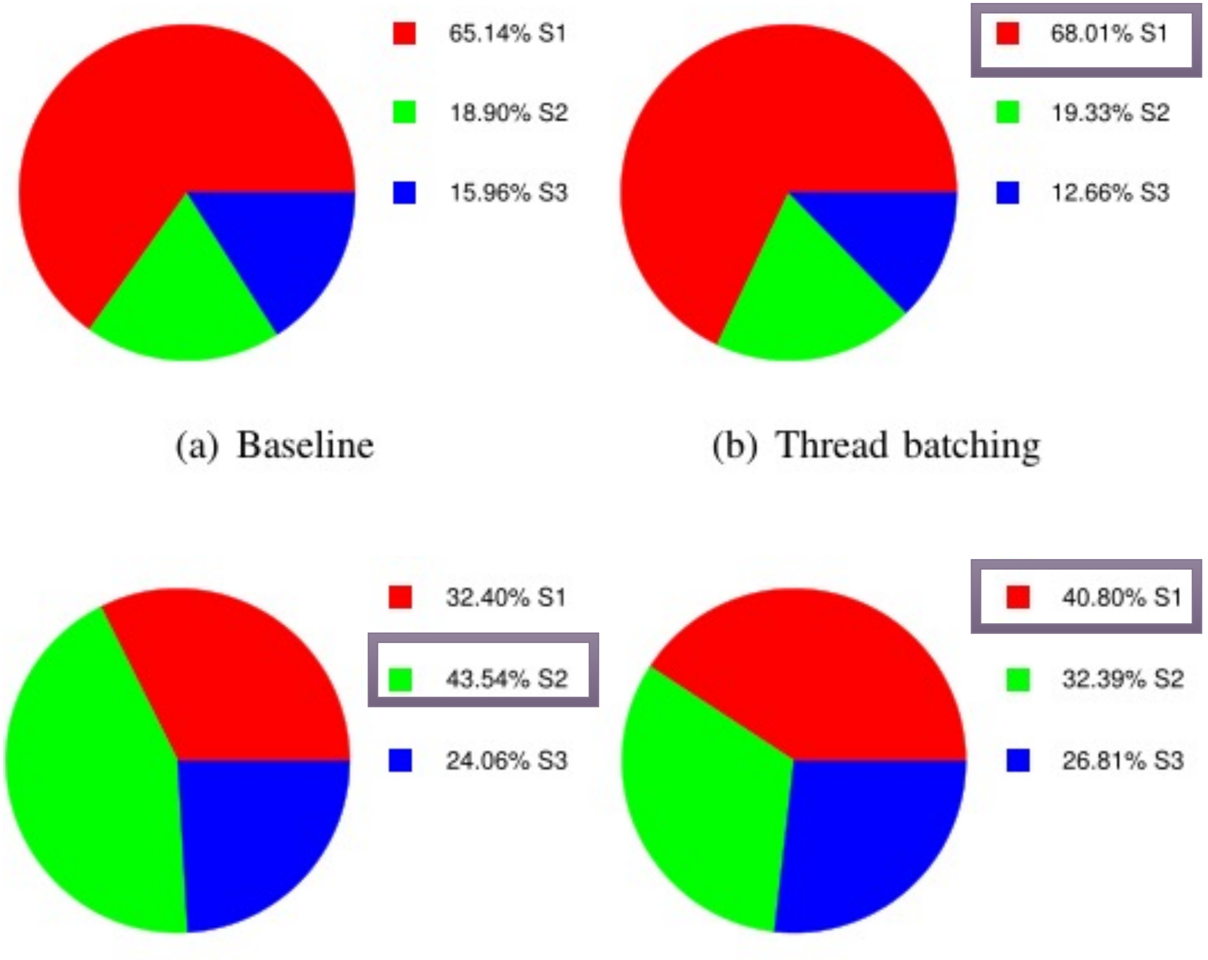
GPU: thread batching + local memory + register (upto 2.6×)

CPU/MIC: thread batching + local memory (1.4× for MIC, 1.6× for CPU)

5.3 Apply Optimizations

- ALS
  - step1: Y^TY+lambda*I
  - step2: Y^Tr_u
  - step3: solve the linear system
- give a priority to the most time-consuming step

Applying Optimizations



(a) Baseline (b) Thread batching

(c) Optimizing S1 (d) Optimizing S2

Hot-spot guided manner

step a

+ thread baching

step b

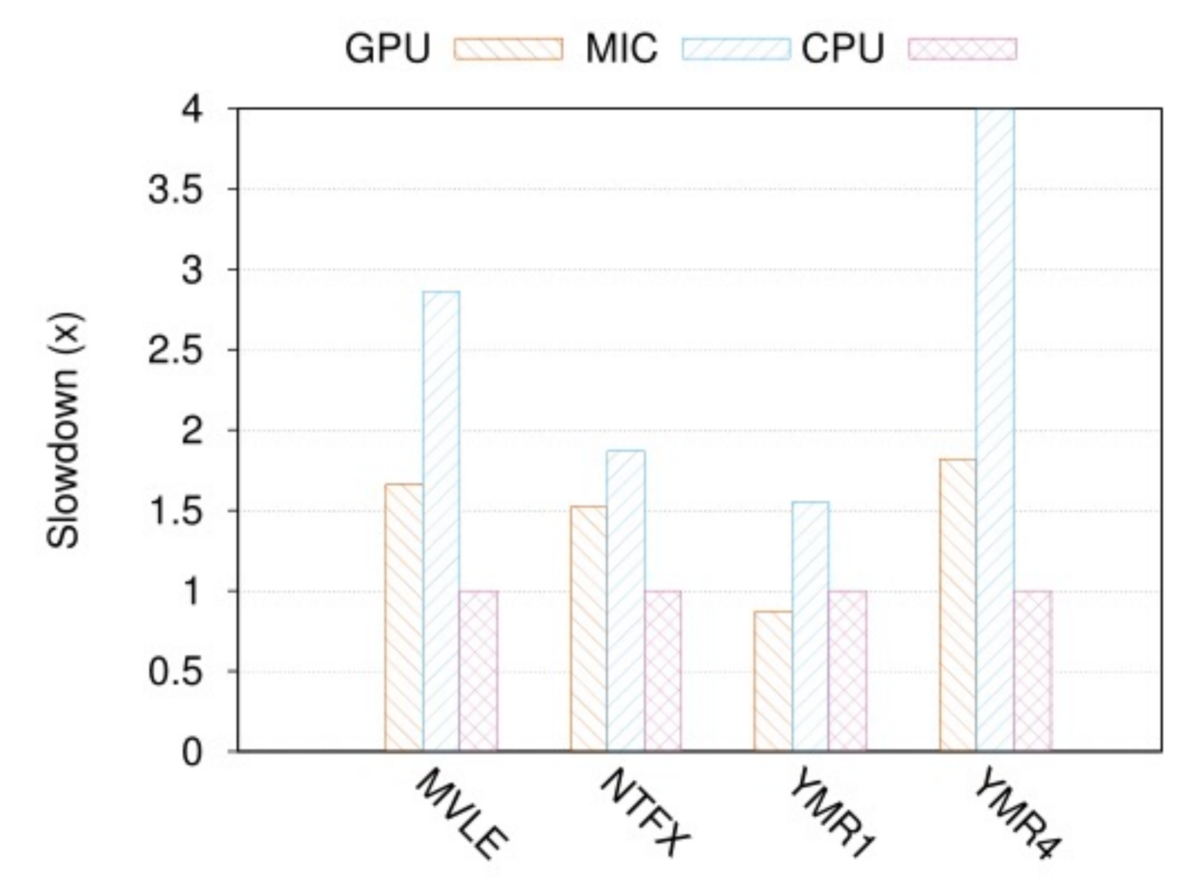
optimize Y^TY+lambda*I

step c

optimize Y^Tr_u

step d

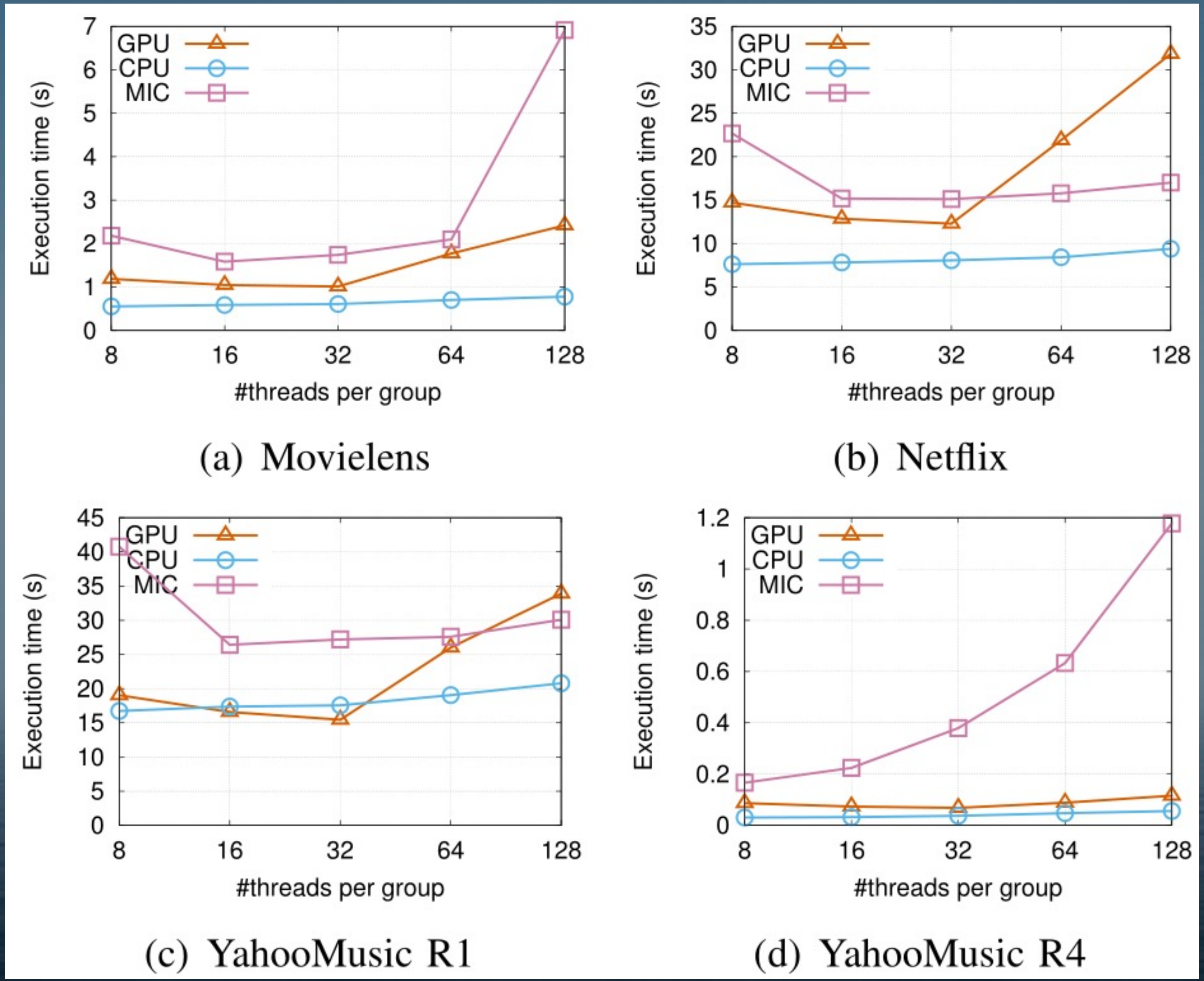
5.4 Compare between Different Architectures



- **CPU performs best**
- **1.5x slower on GPU**
- **4.1x slower on MIC**

But, for Yahoo Music R1 the performance on GPU outperform that on 16-core CPU

5.5 Sensitivity to Thread Blocks



Configuration:

- 8192 * 32, k=10
- thread batching + local memory + registers

GPU: threads per block=16 / 32, best performance!

K = 10

warp size = 32

6. CONCLUSION

Efficient and Portable ALS solver

- ✓ hierarchical thread organization on modern hardware
- ✓ thread batching
- ✓ architecture-specific optimizations
- ✓ OpenCL implementation (CPUs, GPUs, MICs)
- ✓ select suitable variant for each platform

THANK YOU