DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

COMPUTER ARCHITECTURES

Locality-aware Programming

Prepared by: **Gábor Horváth**, ghorvath@hit.bme.hu

Budapest,
2024. 04. 09.

MŰEGYETEM 1782

THE IMPORTANCE OF LOCALITY OF REFERENCES

- How to develop **slow** programs?
  - Let us refer the memory content in a random order
    - Many cache misses
    - Many TLB misses
    - Many page faults
    - DRAM row activation many times
- How to develop **fast** programs?
  - Let us take the memory hierarchy into consideration
  - Memory references should exhibit
    - Spatial locality
    - Temporal locality

Quantifying the effect of the locality of references

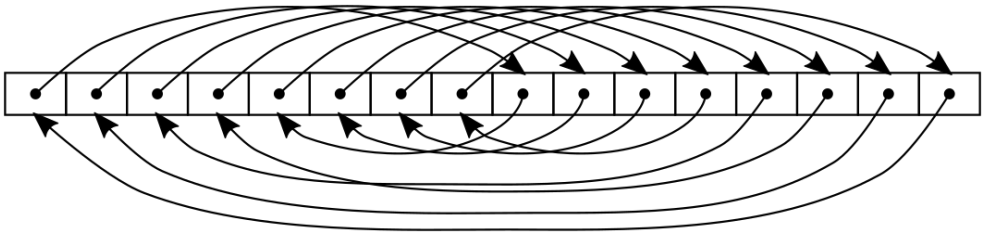
Is it worth using memory hierarchy aware programing tricks?

WHAT TO MEASURE?

- The effect of **temporal** locality
  - How important is it to decrease the memory used by the programs, and use the same data many times?
- The effect of **spatial** locality
  - How important is it to traverse the data in a memory continuous way?

MEASURING THE EFFECT OF TEMPORAL LOCALITY

- Measurement method:
  - Let us take a large (N) array
  - Array entries: pointers to further entries of the array
  - The pointer chain includes all elements in a random-like order



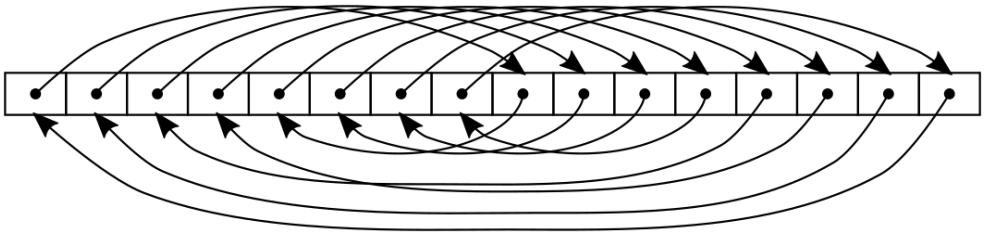
- The naive algorithm for traversing the pointer chain is:

```
for (int i=0; i<iterations; i++)
    p = *p;
```

- We use a trick called “loop unrolling” to amortize the cost of the handling of the **`for`** cycle (see next slide)

MEASURING THE EFFECT OF TEMPORAL LOCALITY

- Measurement method:
  - Let us take a large (N) array
  - Array entries: pointers to further entries of the array
  - The pointer chain includes all elements in a random-like order

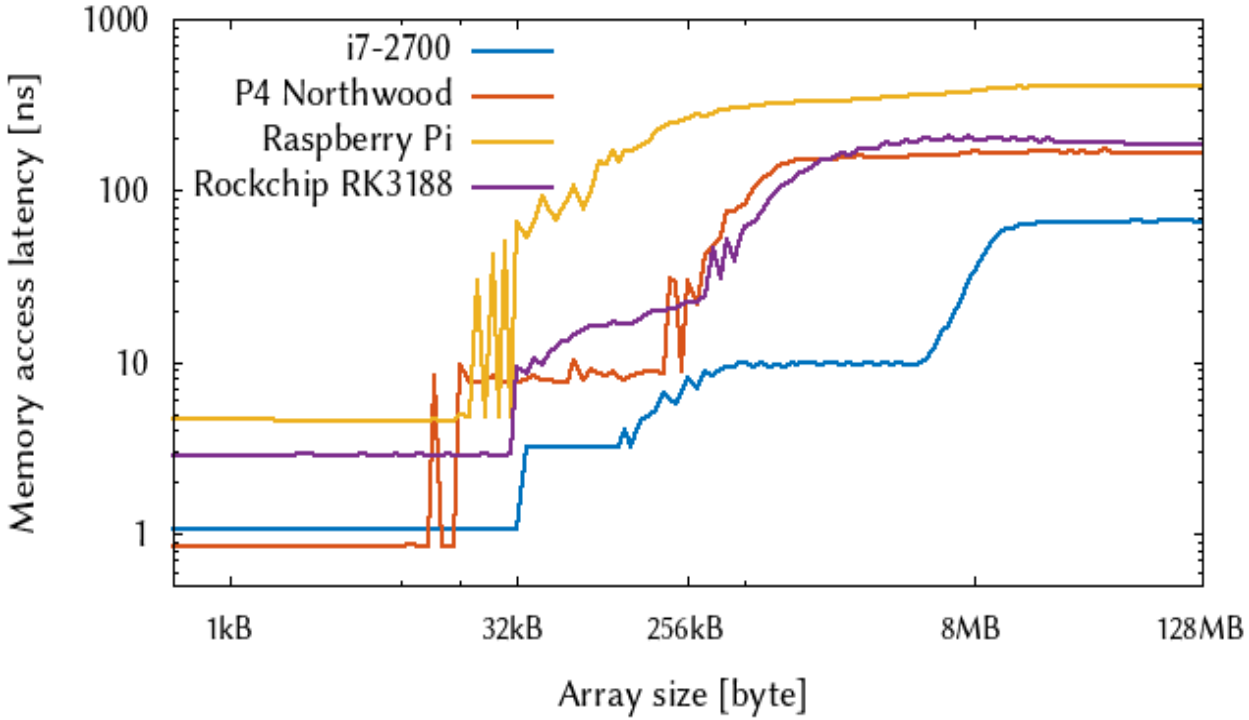


- We measure the execution time of traversing the pointer chain:

```
for (int i=0; i<iterations/100; i++) {
    p = *p;
    p = *p;
    ...
    p = *p;
}
```

MEASURING THE EFFECT OF TEMPORAL LOCALITY

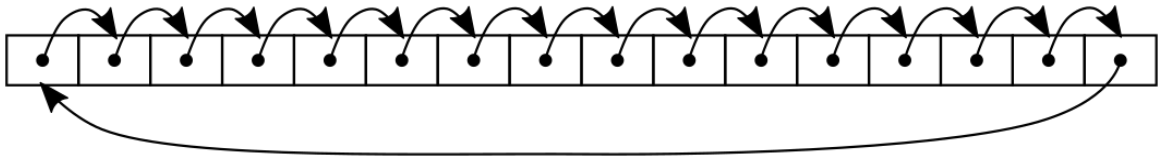
Memory access latency as a function of array size and CPU type



- Conclusion:
  - The size of caches can be identified
  - Message:
    - Temporal locality does matter a lot
    - **Difference of memory access times can be up to 200x !!!**

MEASURING THE EFFECT OF SPATIAL LOCALITY

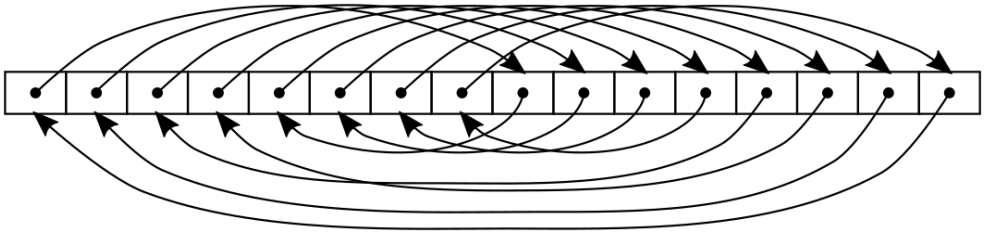
- Measurement method:
  - Like before, but the chain is sequential now (rather than random)



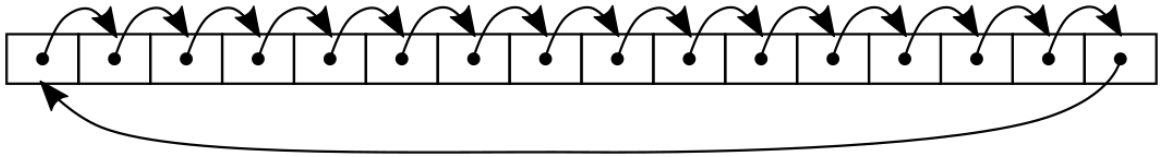
- Expected results: improved memory access times
  - Cache miss ratio decreases:
    - If a cache block is referred to, we proceed and use all further elements of the same cache block as well
    - If there is a cache prefetch algorithm in the CPU, it can take the advantage of sequential read operations
  - The TLB miss ratio decreases
    - If a page is referred to, we proceed and use all further elements of the same page, thus the same page table entry can be used for address translations

MEASURING THE EFFECT OF SPATIAL LOCALITY

- The results of three kinds of measurements are compared:
  - The previous measurement with random array (used as reference)
    → Only temporal locality may result in a cache hit



  - The currently introduced “elementwise” sequential traversal of the array
    → Spatial locality will result in cache hits

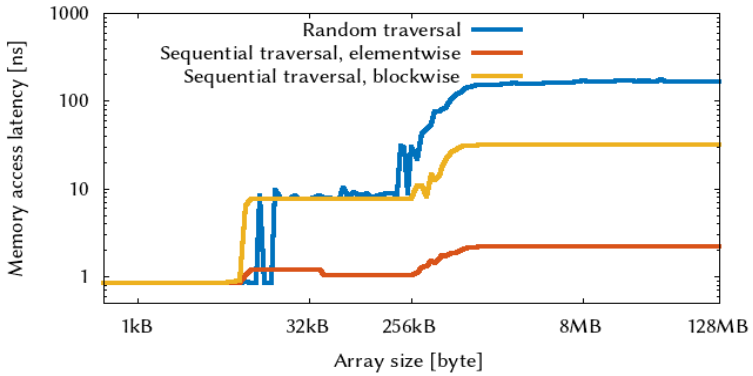


  - Sequential traversal of the array, using stride equal to the cache block size
    → Only successful prefetch can result in a cache hit!

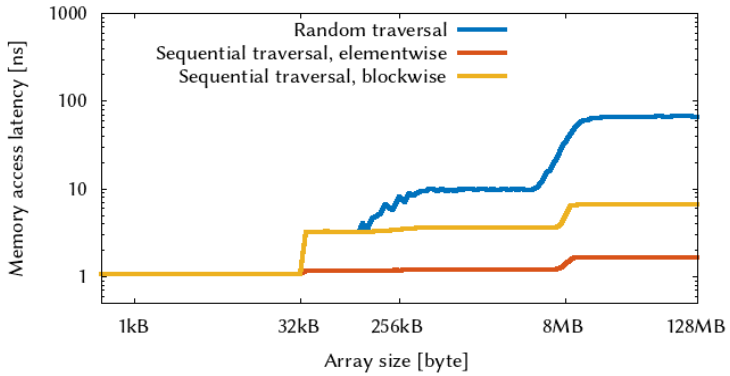
MEASURING THE EFFECT OF SPATIAL LOCALITY

- Results:

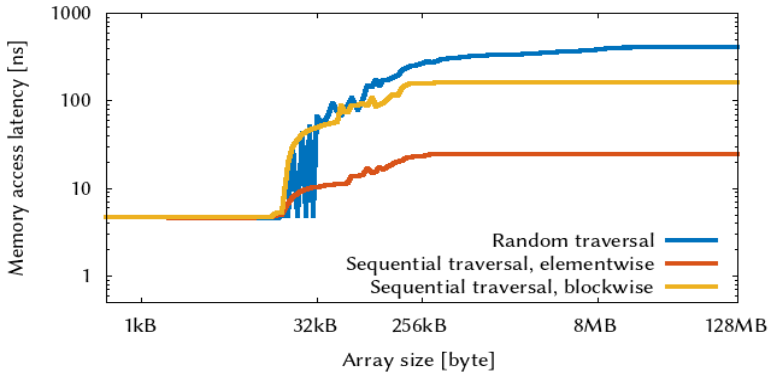
Pentium 4



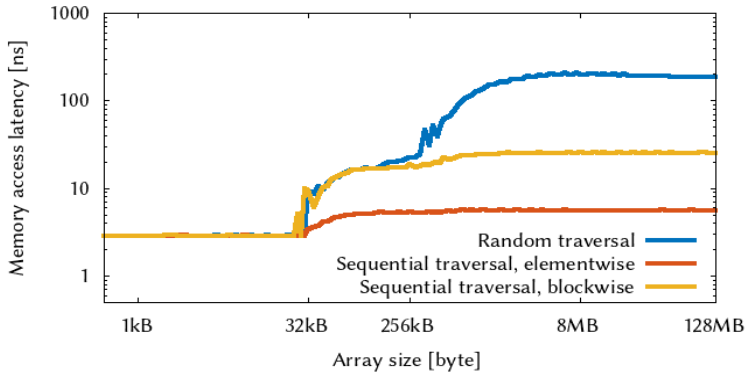
Core i7



ARM (raspberry pi)



ARM (rk3188)



MEASURING THE EFFECT OF SPATIAL LOCALITY

- Conclusion:
  - It is worth traversing data structures in a memory continuous way
  - In case of large arrays the difference is 40x-80x

Locality aware loops

What can a C programmer do?

LOOP FUSION

Original C code:

```
for (i=0; i<N; i++)
    b[i] = c * a[i] + x;
sum = 0;
for (i=0; i<N; i++)
    sum += b[i];
for (i=0; i<N; i++)
    d[i] = a[i] + b[i];
```

After loop fusion:

```
sum = 0;
for (i=0; i<N; i++) {
    b[i] = c * a[i] + x;
    sum += b[i];
    d[i] = a[i] + b[i];
}
```

- Cache miss analysis: (assume 8 double / cache block, N is large)
- Original code:
  - First loop: 2N references, 2N/8 cache misses
  - Second loop: N references, N/8 misses
  - Third loop: 3N references, 3N/8 misses
- Total: 6N references, 6N/8 cache misses

  → **Cache miss ratio: 1/8 = 12.5%**

LOOP FUSION

Original C code:

```
for (i=0; i<N; i++)
    b[i] = c * a[i] + x;
sum = 0;
for (i=0; i<N; i++)
    sum += b[i];
for (i=0; i<N; i++)
    d[i] = a[i] + b[i];
```

After loop fusion:

```
sum = 0;
for (i=0; i<N; i++) {
    b[i] = c * a[i] + x;
    sum += b[i];
    d[i] = a[i] + b[i];
}
```

- After loop merging:
  - First line of the loop: 2N references, 2N/8 cache miss
  - Second line: N references, 0 cache miss!
  - Third line: 3N references, N/8 miss (due to d[i])
- Total: 6N references, 3N/8 cache miss

  → **Cache miss ratio: 1/16 = 6.25%**

LOOP FUSION

- Conclusion:
  - Traversing arrays several times should be avoided
  - A common loop is better than multiple small loops
- Measurement results:
  - $N=2^{22}$

	i7-2600	P4	Rasp. Pi	RK3188
Original algorithm	16.533 ms	109.974 ms	698.450 ms	115.354 ms
After loop merging	8.469 ms	84.917 ms	203.755 ms	97.126 ms

OPTIMIZING THE ORDER OF LOOPS

Row-continuous traversal:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        sum += a[i][j];
```

Column-continuous traversal:

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        sum += a[i][j];
```

- C language: arrays are stored in a row-continuous way
  - Also called: row major order https://en.wikipedia.org/wiki/Row-_and_column-major_order
- Assumption: 8 double/cache block, N large
- Cache miss analysis:
- With row-continuous traversal:
  - Array is traversed in a memory-continuous way
  - We saw how fast it is
  - 1 cache miss for 8 memory references
  - **Cache miss ratio: 1/8 = 12.5%**

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]

OPTIMIZING THE ORDER OF LOOPS

Row-continuous traversal:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        sum += a[i][j];
```

Column-continuous traversal:

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        sum += a[i][j];
```

Column-continuous traversal:

- N-1 elements are skipped after each memory reference
- If the CPU supports cache prefetch, it can adapt to this behavior and fetch the data before the first references
- If there is no prefetch and N > cache size:
  - Blocks are replaced before incrementing j
  - Each memory references imply a cache miss!
  - **Cache miss ratio: 100%**

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]

OPTIMIZING THE ORDER OF LOOPS

- Conclusion:
  - Data structures should be traversed in a memory-continuous way

- Measurement results:
  - N=2048

	i7-2600	P4	Rasp. Pi	RK3188
Row-continuous	6.312 ms	8.973 ms	605.757 ms	14.879 ms
Column-continuous	6.926 ms	160.78 ms	4363.13 ms	60.96 ms

(Core i7 has a cache prefetch algorithm)

LOOP TILING

Original C code:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        b[j][i] = a[i][j];
```

After loop tiling:

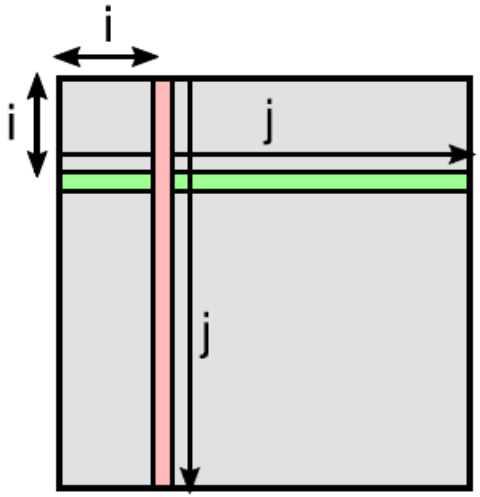
```
for (bi=0; bi<=N-BLK; bi+=BLK)
    for (bj=0; bj<=N-BLK; bj+=BLK)
        for (i=bi; i<bi+BLK; i++)
            for (j=bj; j<bj+BLK; j++)
                b[j][i] = a[i][j];
```

- Matrix transpose (image rotation, etc.)
- Assumptions: 8 double/cache block, N is large
- Cache miss analysis:
- Original C code:
  - a[i][j]: row-continuous traversal, N^2 references, $N^2/8$ cache misses
  - b[j][i]: column-continuous traversal, N^2 references, N^2 cache misses
  - Total: $2N^2$ references, $N^2/8 + N^2$ cache miss
  - **Cache miss ratio: 9/16 = 56.25%**

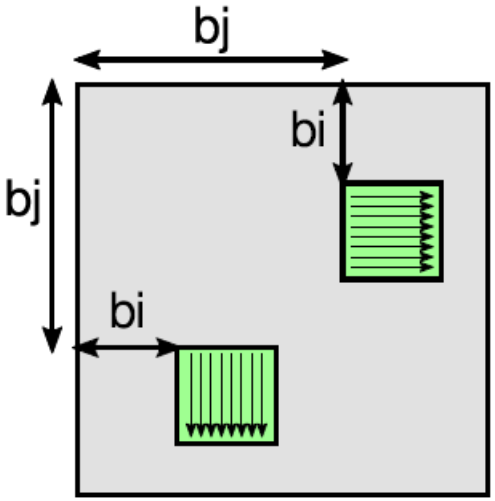
LOOP TILING

- Operation of the matrix transposition:

  Original C code:



  After loop tiling:



- The N x N arrays do not fit into the cache
  - Their column continuous access has 100% cache miss ratio
- The two (green colored) BLK x BLK block do fit into the cache
  - Their column-continuous access will have only 1/8 cache miss ratio

LOOP TILING

Original C code:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        b[j][i] = a[i][j];
```

After loop tiling:

```
for (bi=0; bi<=N-BLK; bi+=BLK)
    for (bj=0; bj<=N-BLK; bj+=BLK)
        for (i=bi; i<bi+BLK; i++)
            for (j=bj; j<bj+BLK; j++)
                b[j][i] = a[i][j];
```

- After loop tiling:
  - We proceed block-by-block
  - If BLK is properly set, a BLK x BLK sized block fits into the cache
  - a[i][j] and b[j][i] will both be in the cache!
  - a[i][j]: row-continuous traversal, N^2 references, $N^2/8$ cache miss
  - b[j][i]: column-continuous traversal, N^2 references, $N^2/8$ cache miss
  - Total: $2N^2$ references, $N^2/8$ +$N^2/8$ cache miss
  - **Cache miss ratio: 1/8 = 12.5%**

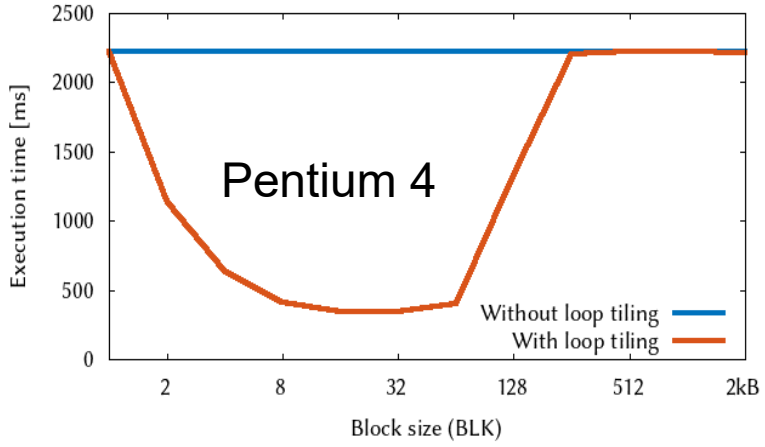
LOOP TILING

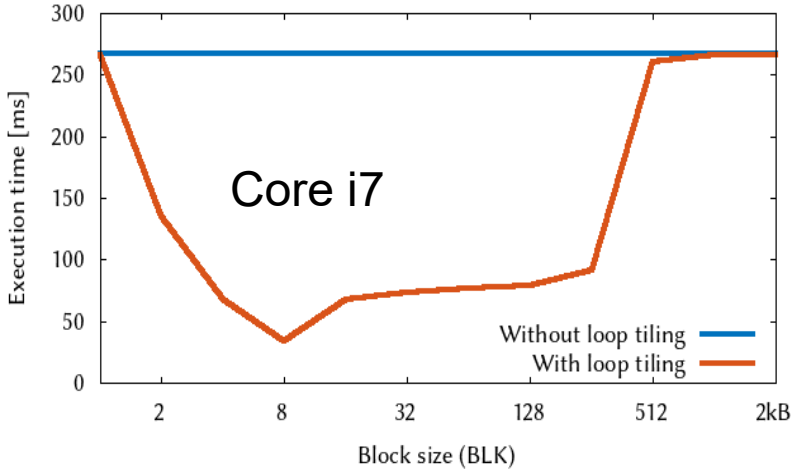
- How to determine the optimal block size?
  - Too small → like without loop tiling
  - Too large → like without loop tiling
  - Architecture dependent! It depends on both
    - Cache size
    - Cache block size

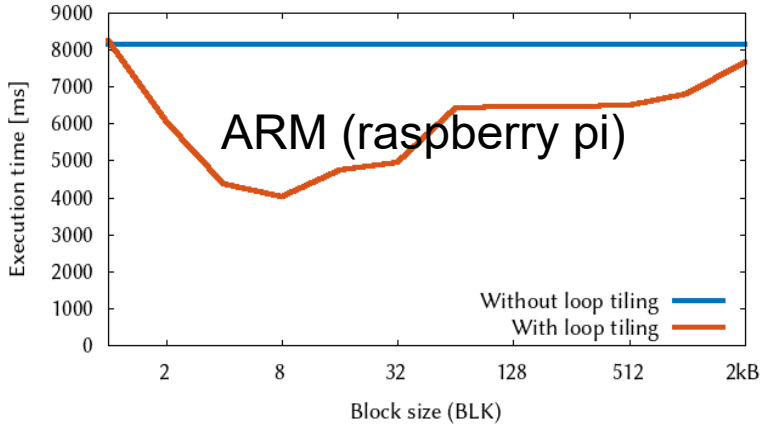
- Measurement results:
  - N=2048
  - BLK from 1 to 2048

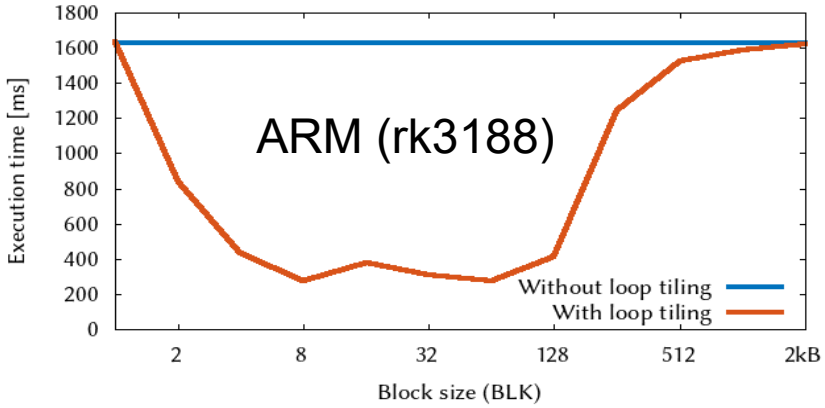
LOOP TILING

- Results:









DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

MŰEGYETEM 1782