Cache Complexity in Computer Algebra

Marc Moreno Maza

Ontario Research Center for Computer Algebra
Departments of Computer Science and Mathematics
University of Western Ontario, Canada

École Polytechnique, 5 Décembre 2022

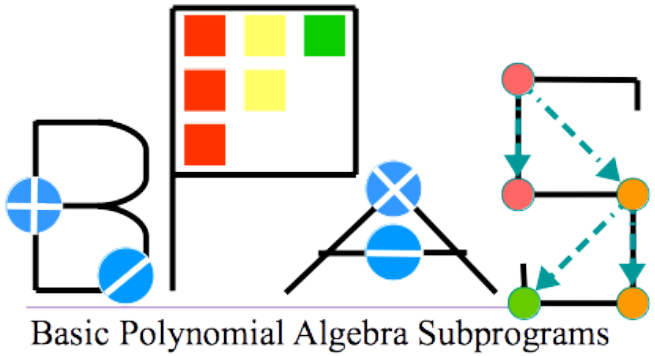
Acknowledgements

- Many thanks to the organizers of this seminar for their invitation.
- This talk a collection of observations, rather than a research talk with new results. I will show many slides that you may have already seen.
- These observations are based on research projects in which many of my former and current graduate students have played an essential role. By alphabetic order: Ali Asadi, Alexander Brandt Changbo Chen, Xiaohui Chen, Svyatoslav Covanov, Sardar Haque, Xin Li, Farnam Mansouri, Davood Mohajerani, Robert Moir, Wei Pan, Delaram Talaashrafi, Linxiao Wang, Ning Xie, Yuzhen Xie, Haoze Yuan.
- This talk is also based on collaborations with MIT/CSAIL, Intel and IBM Canada, with funding support from IBM and NSERC of Canada.
- Special thanks go to Alexander Brandt who is leading the development of the Basic Polynomial Algebra Subprograms (BPAS) [1].

Tentative Plan

- Part 1: (Well-known) Motivations
- Part 2: (Well-known) Memory Models
- Part 3: A case study
- Part 4: Multi-measure models

The BPAS library



http://www.bpaslib.org/

A high-performance polynomial algebra library

- Core of library written in C, wrapped in C++ 11 interface for usability and object-oriented programming

Optimized algorithms and data structures, data locality, and parallelism

- Sparse multivariate polynomials [3], dense univariate and bivariate [24]
- Triangular decomposition of polynomial systems [2, 4]

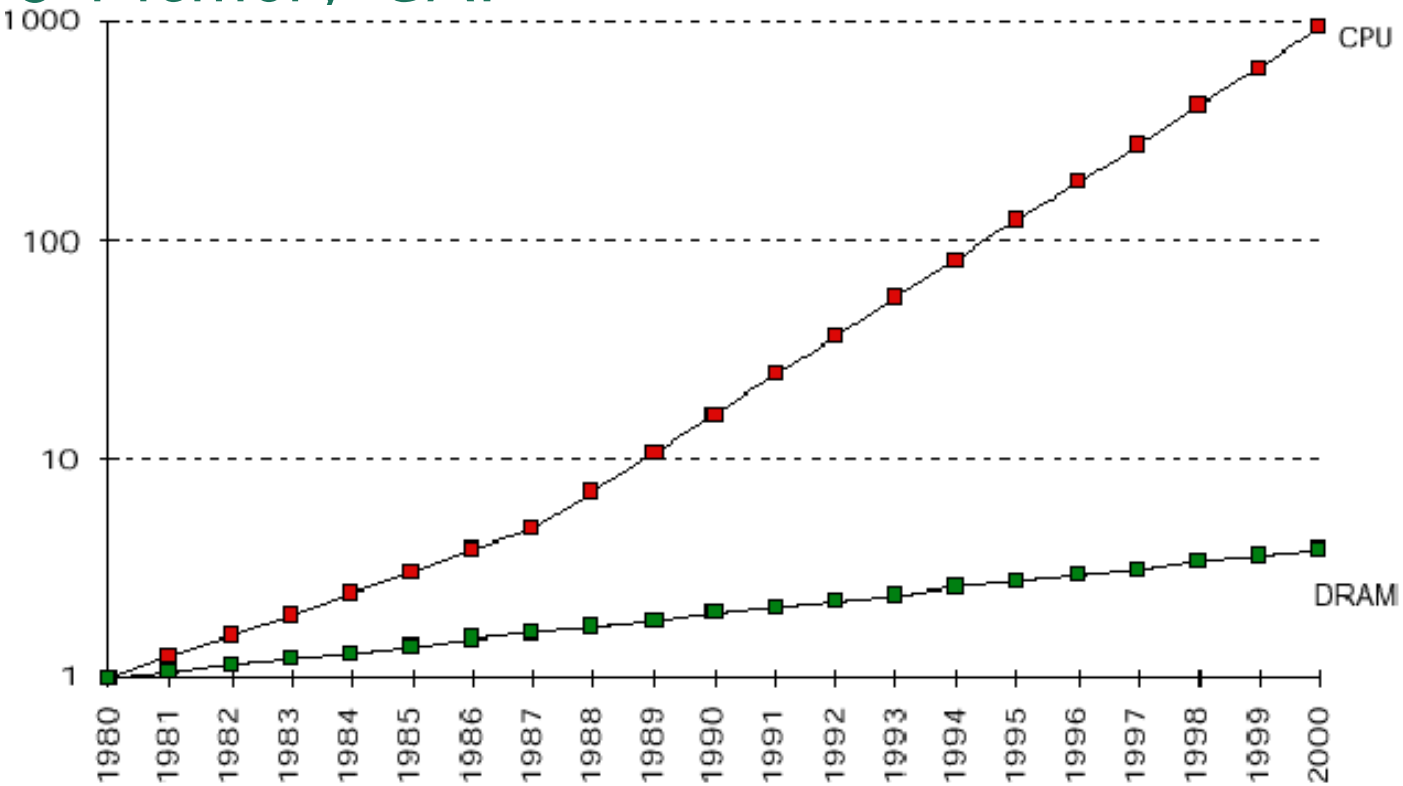
User-friendly, object-oriented interface based on template meta-programming [5]

- A natural encoding of the algebraic hierarchy
- "Dynamic" creation of algebraic types through composition
- Compile-time type safety between algebraic types

Generic support for parallel programming and parallel patterns (this talk)

Outline

The CPU-Memory GAP

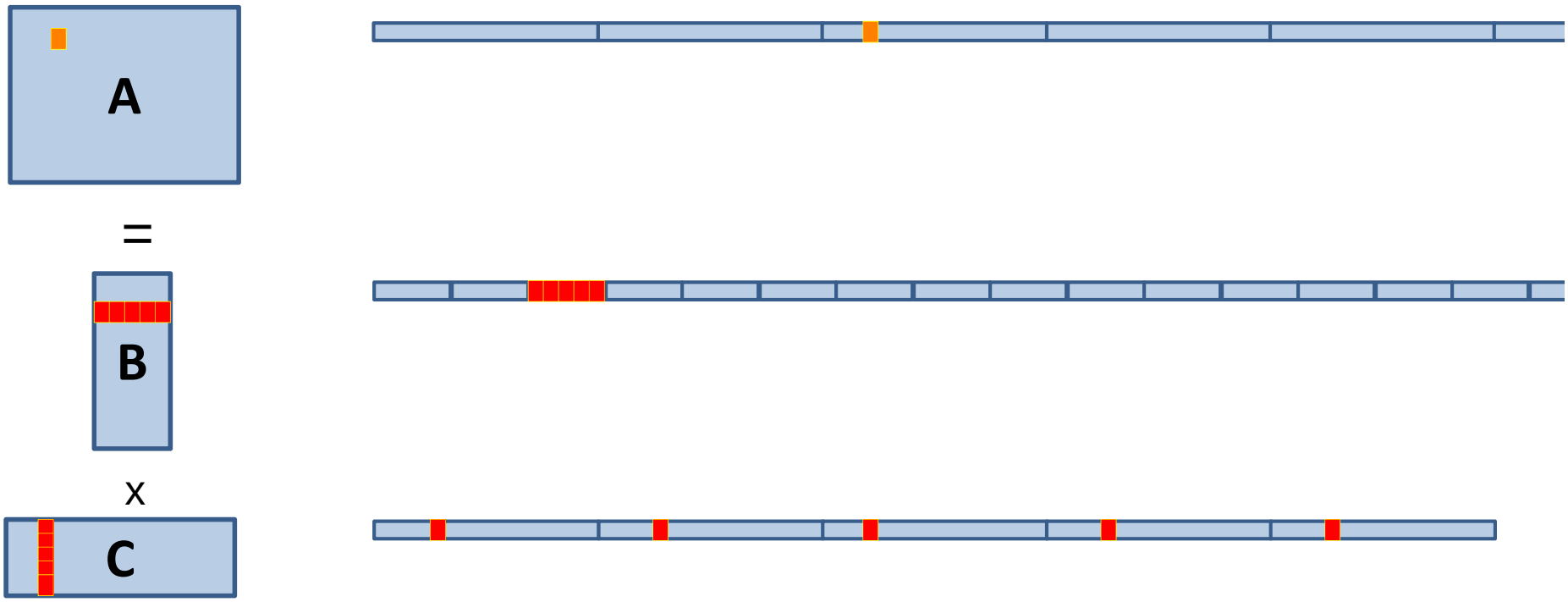


- In the 1980's, a memory access and a CPU operation were both as slow as the other
- CPU frequency increased between 1985 and 2005 has reduced CPU op times much more than DRAM technology improvement could reduce memory access times
- Even after the introduction of multicore processors, the gap is still huge.

A typical matrix multiplication C code

```
#define IND(A, x, y, d) A[(x)*(d)+(y)]
uint64_t testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C;
        long started, ended;
        float timeTaken;
        int i, j, k;
        srand(getSeed());
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
                    //  A[i][j] += B[i][k] * C[k][j];
                    IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with matrix representation

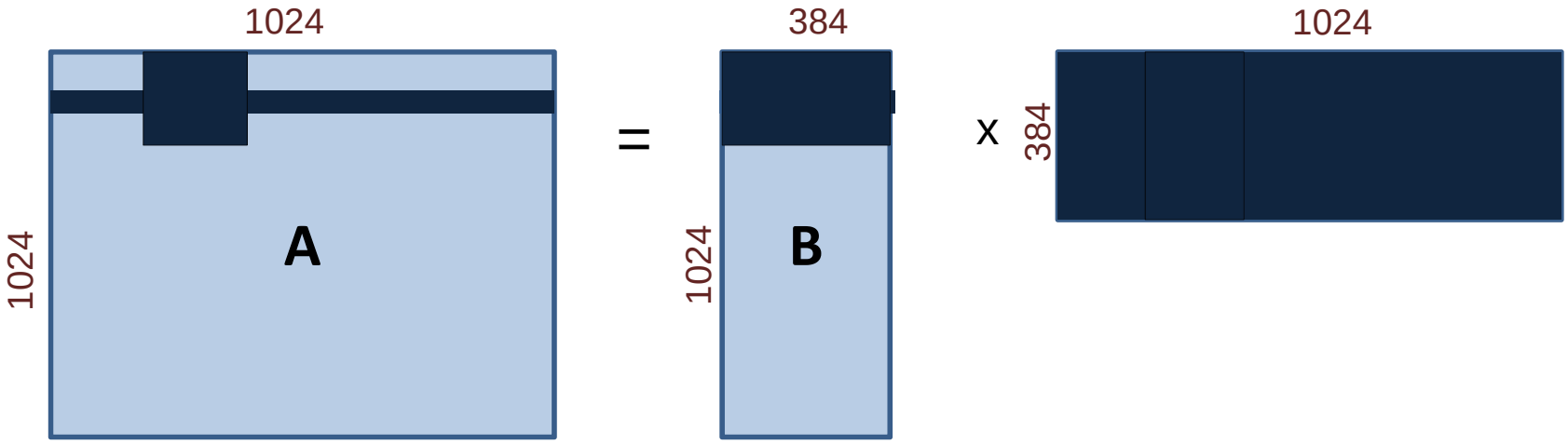


- Contiguous accesses are better:
  - ↳ Data fetch as cache line (Core 2 Duo: 64 byte per cache line)
  - ↳ With contiguous data, a single cache fetch supports 8 reads of doubles.
  - ↳ Transposing the matrix `C` should reduce L1 cache misses!

Transposing for optimizing spatial locality

```
float testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        Cx = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for(j =0; j < y; j++)
          for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C,k,j,y);
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
               IND(A, i, j, y) += IND(B, i, k, z) *IND(Cx, j, k, z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with data reuse



- Naive calculation of a row of `A`, so computing 1024 coefficients: 1024 accesses in `A`, 384 in `B` and $1024 \times 384 = 393,216$ in `C`. Total $= 394,524$.
- Computing a 32×32-block of `A`, so computing again 1024 coefficients: 1024 accesses in `A`, 384×32 in `B` and 32×384 in `C`. Total $= 25,600$.
- The iteration space is traversed so as to reduce memory accesses.

Blocking for optimizing temporal locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                       IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,k0,j0,y);
         ended = example_get_time();
         timeTaken = (ended - started)/1.f;
   return timeTaken;
}
```

Transposing and blocking for optimizing data locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                       IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,j0,k0,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
        return timeTaken;
}
```

Experimental results

Computing the product of two $n \times n$ matrices on my laptop (Core2 Duo CPU P8600 @ 2.40GHz, L1 cache of 3072 KB, 4 GBytes of RAM)

n	naive	transposed	speedup	64×64-tiled	speedup	t. & t.	speedup
128	7	3		7		2	
256	26	43		155		23	
512	1805	265	6.81	1928	0.936	187	9.65
1024	24723	3730	6.62	14020	1.76	1490	16.59
2048	271446	29767	9.11	112298	2.41	11960	22.69
4096	2344594	238453	**9.83**	1009445	**2.32**	101264	**23.15**

Timings are in milliseconds.

The cache-oblivious multiplication (more on this later) runs within 12978 and 106758 for $n = 2048$ and $n = 4096$ respectively.

Other performance counters

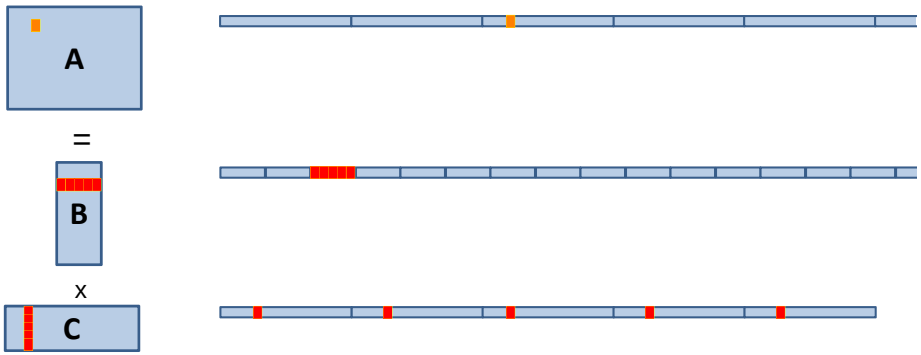
Hardware count events

- CPI – Clock cycles Per Instruction: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism
- L1 and L2 Cache Miss Rate.
- Instructions Retired: In the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled.

	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

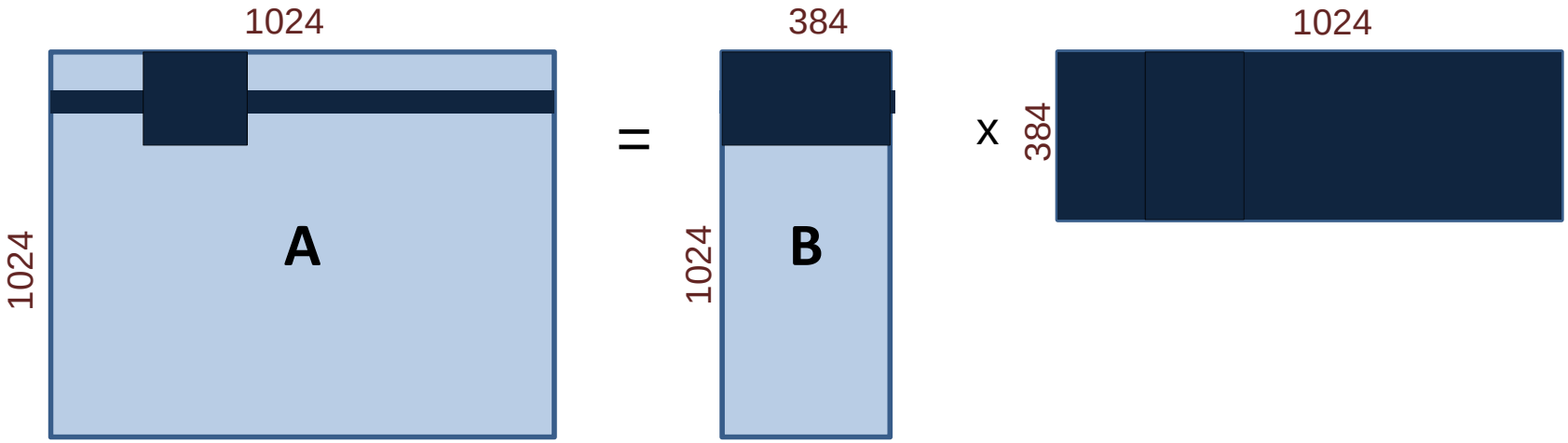
skip slide

Analyzing cache misses in the naive and transposed multiplication



- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned one, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.
- C is accessed "nearly randomly" (for m large enough) leading to mnp cache misses.
- Since $2m\,n\,p$ arithmetic operations are performed, this means roughly **one cache miss for two flops!**
- If C is transposed, then the ratio improves to 1 for L.

Analyzing cache misses in the tiled multiplication



- Let A, B and C are all square of order of n.
- Assume all tiles are square of order b and three fit in cache.
- If C is transposed, then loading three blocks in cache cost $3b^2/L$.
- This process happens n^3/b^3 times, leading to $3n^3/(bL)$ cache misses.
- Three blocks fit in cache for $3b^2 < Z$, if Z is the cache size.
- So $O(n^3/(\sqrt{Z}L))$ cache misses, if b is well chosen, which is optimal.

Outline

Overview

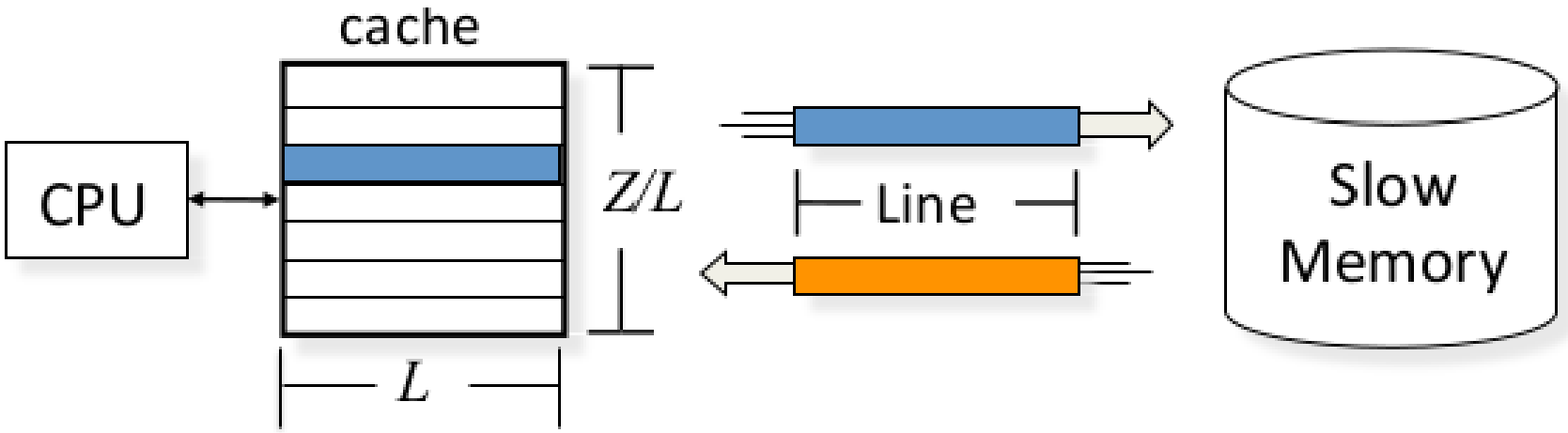
We will discuss

- the details of the *ideal cache model* proposed by Frigo, Leiserson, Prokop and Ramachandran in [9],
- the principles of the *I/O Complexity Model* proposed by Jia-Wei Hong and Hsiang-Tsung Kung in [17].

Outline

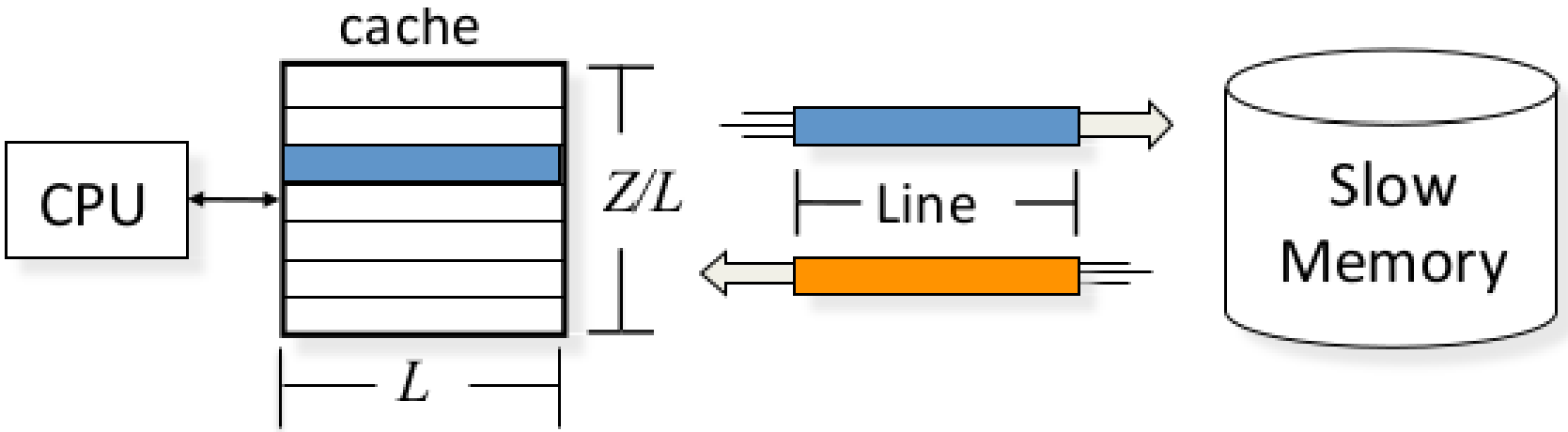
Outline

The ideal cache model (1/5)



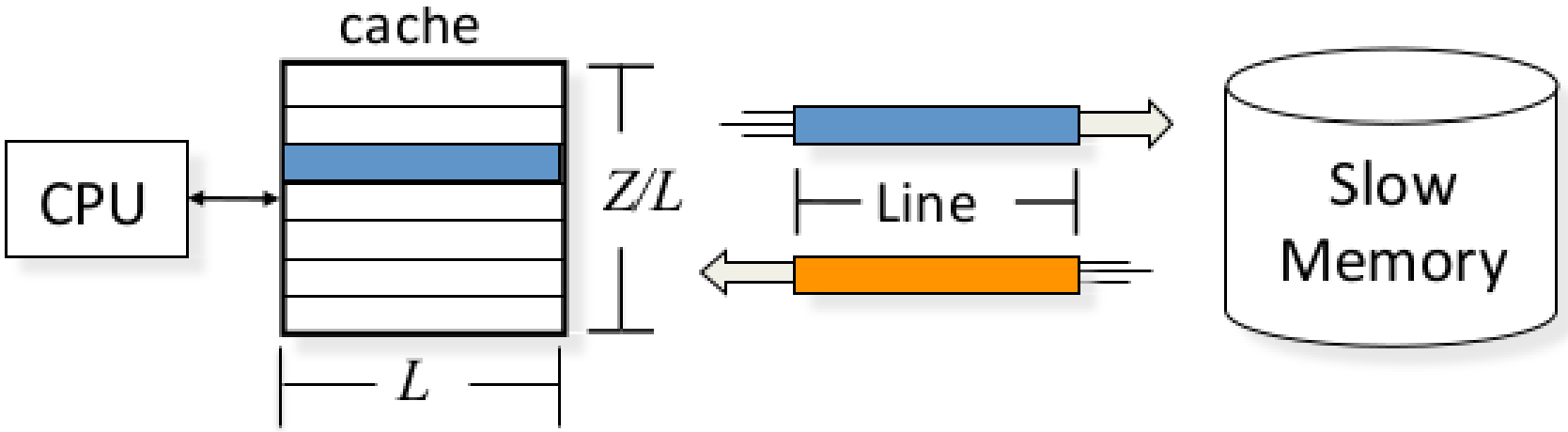
- Computer with a **two-level memory hierarchy**:
  - ↳ an ideal (data) cache of Z words partitioned into Z/L *cache lines*, where L is the number of words per cache line.
  - ↳ an arbitrarily large main memory.
- Data moved between cache and main memory are always cache lines.
- The cache is **tall**, that is, Z is much larger than L, say $Z \in \Omega(L^2)$.

The ideal cache model (2/5)



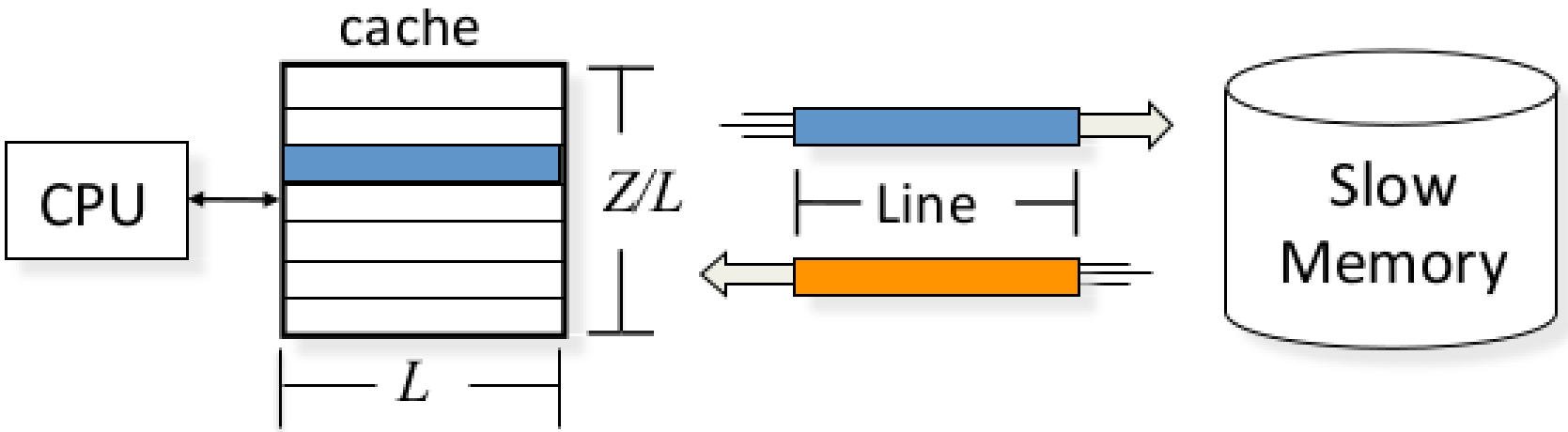
- The processor can only reference words that reside in the cache.
- If the referenced word belongs to a line already in cache, a **cache hit** occurs, and the word is delivered to the processor.
- Otherwise, a **cache miss** occurs, and the line is fetched and installed into the cache.

The ideal cache model (3/5)



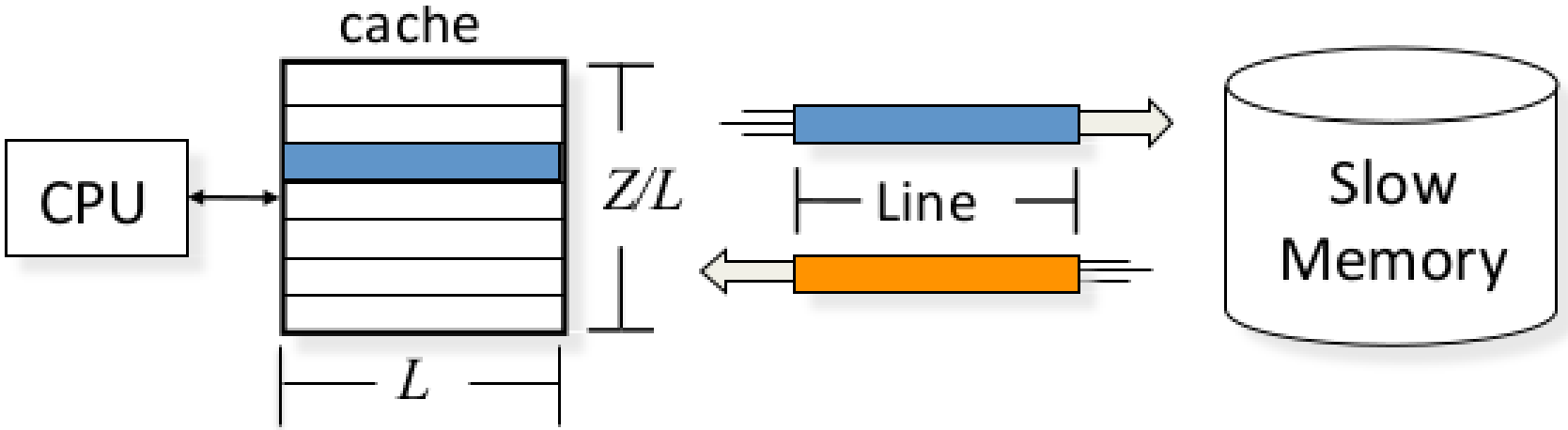
- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.
- The ideal cache uses the **optimal off-line strategy of replacing** the cache line whose next access is furthest in the future, and thus it exploits temporal locality perfectly.

The ideal cache model (4/5)



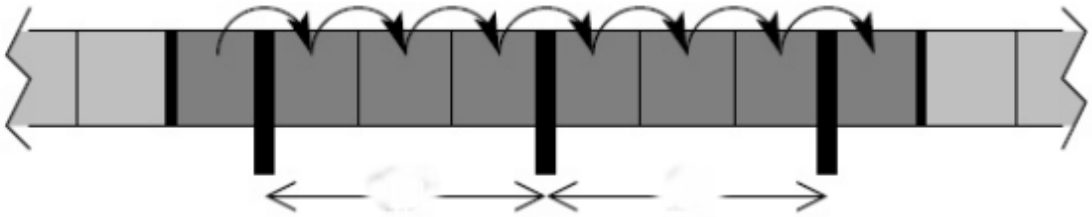
- For an algorithm with an input of size n, the ideal-cache model uses two complexity measures:
  - the **work complexity** $W(n)$, which is its conventional running time in a RAM model.
  - the **cache complexity** $Q(n; Z, L)$, the number of cache misses it incurs (as a function of the size Z and line length L of the ideal cache).
  - When Z and L are clear from context, we simply write $Q(n)$ instead of $Q(n; Z, L)$.

The ideal cache model (5/5)



- An algorithm is said to be **cache aware** if its behavior (and thus performances) can be tuned (and thus depend on) on the particular cache size and line length of the targeted machine.
- Otherwise the algorithm is **cache oblivious**.
- Cache oblivious naturally performs well on hierarchical memories.

Scanning



- Scanning n words stored in a contiguous segment of memory with cache-line size L costs at most $\lceil n/L \rceil + 1$ cache misses.
- If this vector of n words is aligned in memory, then this estimate is simply $\lceil n/L \rceil$.

Proof.

- Let (q, r) be the quotient and remainder in the integer division of n by L.
- Let u (resp. w) be the total number of words stored in cache-lines fully (not fully) used by those n consecutive words. Thus, we have $n = u + w$. Three cases arise.
  1. if $w = 0$ then $(q, r) = (\lfloor n/L \rfloor, 0)$ and the scanning costs exactly q; thus the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor$ in this case.
  2. if $0 < w < L$ then $(q, r) = (\lfloor n/L \rfloor, w)$ and the scanning cost is at most $q + 2$; the conclusion is clear since $\lceil n/L \rceil = \lfloor n/L \rfloor + 1$ in this case.
  3. if $L \leq w < 2L$ then $(q, r) = (\lfloor n/L \rfloor, w - L)$ and the scanning cost is at most $q + 1$; the conclusion is clear again. ⏩ skip slide

Adding vectors

- Consider $m \geq 2$ vectors $V_1, \ldots, V_m$ of size $n \geq 1$ aligned in memory.
- Consider $m-1$ scalars $\alpha_1, \ldots, \alpha_{m-1}$, stored in a contiguous segment of memory in $m-1$ words.
- Assume that the ideal cache has at least $\lceil m/L \rceil + 4$ cache-lines.
- Then, computing the linear combination $\alpha_1 V_1 + \cdots + \alpha_{m-1} V_{m-1}$ and writing it to V_m can be done in no more cache misses than those required for scanning $V_1, \ldots, V_m, \alpha_1, \ldots, \alpha_{m-1}$,
- thus, within $m\lceil n/L \rceil + \lceil m/L \rceil + 1$ cache misses.

Proof.

- We first load $\alpha_1, \ldots, \alpha_{m-1}$ into the cache, thus using at most $\lceil m/L \rceil + 1$ cache-lines.
- In the pseudo-code below, vector indexing starts at 0.
  1. For b with $0 \leq b \leq \lfloor n/L \rfloor$, for each j with $1 \leq j < m$, for each i with $0 \leq i < L$ do:
     1. $k := b * L + i$,
     2. if $k < n$ then $V_m[k] := V_m[k] + \alpha_j V_j[k]$
- Use the optimal replacement policy and the fact that vectors are aligned in memory

Counting sort: the algorithm

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- *Counting sort* takes as input a collection of n items, each of which known by a key in the range $0 \cdots k$.
- The algorithm computes a *histogram* of the number of times each key occurs.
- Then performs a *prefix sum* to compute positions in the output.

Counting sort: poor spatial locality

```
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```

- For n large enough: $Q(n; Z, L) = 3n + 3n/L + 2k/L$ cache misses (worst case).
- The possibly random distribution of the input values creates possibly many non-cold misses, see counting_sort.pdf for an animation.

Counting sort: improved by a *blocking strategy*

```
alloacate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
alloacate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput
```

- Split the input value range into m buckets (given by well-chosen pivot values) so that counting sort can be applied in succession to several smaller input arrays, with smaller value ranges, incurring cold misses only, see counting_sort_bucket.pdf for an animation.
- This yields $Q(n; Z, L) = 9n/L + 3m/L + m + 2k/L$ (assuming $m < Z/(1 + L)$) improving on $3n + 3n/L + 2k/L$.

Counting sort: experimentation

- Experimentation on an *Intel(R) Core(TM) i7 CPU @ 2.93GHz*. It has L2 cache of 8MB.
- CPU times in seconds for both classical and cache-friendly counting sort algorithm.
- The keys are random machine integers in the range $[0, n]$.

n	classical counting sort	cache-friendly counting sort (bucketing + sorting)
100000000	13.74	4.66 (= 3.04 + 1.62)
200000000	30.20	9.93 (= 6.16 + 3.77)
300000000	50.19	16.02 (= 9.32 + 6.70)
400000000	71.55	22.13 (= 12.50 +9.63)
500000000	94.32	28.37 (= 15.71 + 12.66)
600000000	116.74	34.61 (= 18.95 + 15.66)

Cache-friendly counting sort: extension to sample sort

1. Split the input array into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$ and sort those subarrays recursively.
2. Choose $m := \sqrt{n} - 1$ "good" pivot values $p_1 \leq p_2 \leq \cdots \leq p_m$.
3. Distribute subarrays into buckets $B_1, \ldots, B_{m+1}$ according to pivots. Bucket B_i has size $n_i \simeq \sqrt{n}$, expectedly.
4. Recursively sort the buckets
5. Copy-concatenate the buckets back to the input array.

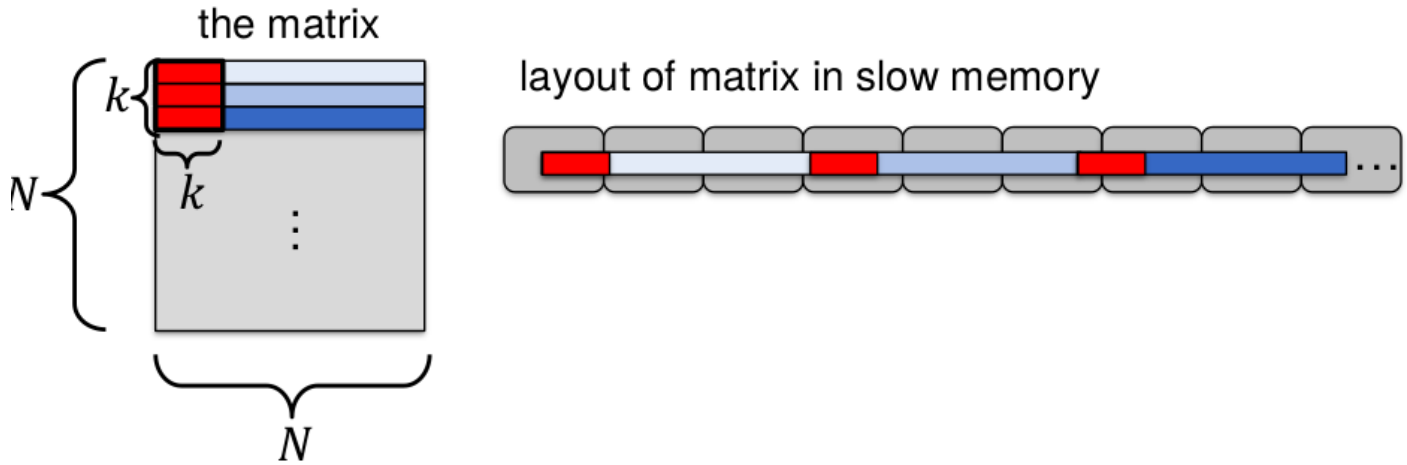
Cache complexity analysis of Sample sort

- Step 1 costs $\sqrt{n}Q(\sqrt{n})$, Step 4 (expectedly) costs $\sqrt{n}Q(\sqrt{n})$ also and Steps 2, 3, 5 cost $\Theta(n/L)$. Thus, we have:

$$Q(n) = \begin{cases} n/L & \text{if} \quad n < Z \quad \text{(base case)} \\ 2\sqrt{n}Q(\sqrt{n}) + \Theta(n/L) & \text{if} \quad n \geq Z \quad \text{(recurrence)} \end{cases}$$

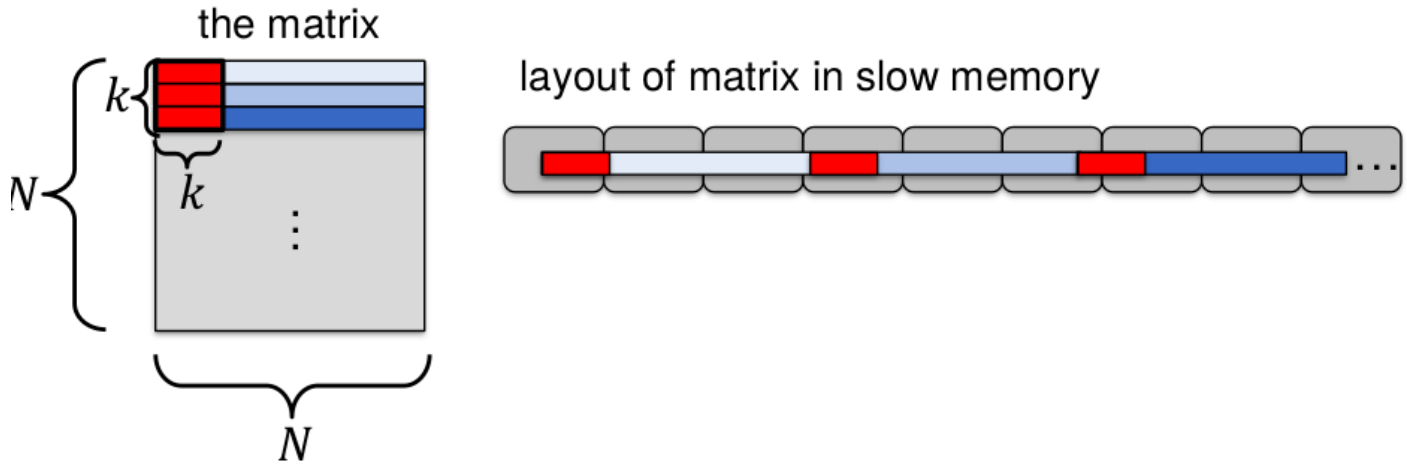
- This yields $Q(n) \in \Theta(\frac{n}{L}\log_Z(n))$.

Transposition of a matrix



- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.
- Assume that each array coefficient is stored on a single word.
- Therefore, reading a $k \times k$ block may incur $k(\lceil k/L \rceil + 1)$ caches misses.
- In this exercise sheet, determine the cache complexity of the proposed algorithms for transposing a square matrix of order n. Assume n large (say $n > Z$) and n is a power of 2.

Transposition of a matrix



- Assume that multi-dimensional arrays (and in particular dense rectangular matrices) are stored in memory using a row-major layout.
- Assume that each array coefficient is stored on a single word.
- Therefore, reading a $k \times k$ block may incur $k(\lceil k/L \rceil + 1)$ caches misses.
- In this exercise sheet, determine the cache complexity of the proposed algorithms for transposing a square matrix of order n. Assume n large (say $n > Z$) and n is a power of 2.
- Algo 1: $\Theta(n^2)$. Algo 2: $\Theta(\log_2(\frac{n}{Z})\,\frac{n^2}{L})$. Algo 3: $\Theta(n^2/L)$. Proofs and precise estimates below. ⏩ skip slide

Matrix transposition: various algorithms

- **Matrix transposition problem:** Given an $m \times n$ matrix A stored in a row-major layout, compute and store A^T into an $n \times m$ matrix B also stored in a row-major layout.
- We shall describe a recursive cache-oblivious algorithm which uses $\Theta(mn)$ work and incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.
- The straightforward algorithm employing doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $m \gg Z/L$ and $n \gg Z/L$.
- We shall start with an apparently good algorithm and use complexity analysis to show that it is even worse than the straightforward algorithm.

Matrix transposition: a first divide-and-conquer (1/4)

- For simplicity, assume that our input matrix A is square of order n and that n is a power of 2, say $n = 2^k$.
- We divide A into four square quadrants of order $n/2$ and we have

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \Rightarrow {}^tA = \begin{pmatrix} {}^tA_{1,1} & {}^tA_{2,1} \\ {}^tA_{1,2} & {}^tA_{2,2} \end{pmatrix}.$$

- This observation yields an "in-place" algorithm:
  1. If $n = 1$ then return A.
  2. If $n > 1$ then
     1. recursively compute ${}^tA_{1,1}, {}^tA_{2,1}, {}^tA_{1,2}, {}^tA_{2,2}$ in place as
     $$\begin{pmatrix} {}^tA_{1,1} & {}^tA_{1,2} \\ {}^tA_{2,1} & {}^tA_{2,2} \end{pmatrix}$$
     2. exchange ${}^tA_{1,2}$ and ${}^tA_{2,1}$.
- What is the number $M(n)$ of memory accesses to A, performed by this algorithm on an input matrix A of order n?

Matrix transposition: a first divide-and-conquer (2/4)

- $M(n)$ satisfies the following recurrence relation

$$M(n) = \begin{cases} 0 & \text{if} \quad n = 1 \\ 4M(n/2) + 2(n/2)^2 & \text{if} \quad n > 1. \end{cases}$$

- Unfolding the tree of recursive calls or using the *Master's Theorem*, one obtains:

$$M(n) = 2(n/2)^2 \ \log_2(n).$$

- This is worse than the straightforward algorithm (which employs doubly nested loops). Indeed, for this latter, we have $M(n) = n^2 - n$. Explain why!
- Despite of this negative result, we shall analyze the cache complexity of this first divide-and-conquer algorithm. Indeed, it provides us with an easy training exercise
- We shall study later a second and efficiency-optimal divide-and-conquer algorithm, whose cache complexity analysis is more involved.

Matrix transposition: a first divide-and-conquer (3/4)

- We shall determine $Q(n)$ the number of cache misses incurred by our first divide-and-conquer algorithm on a (Z, L)-ideal cache machine.
- For n small enough, the entire input matrix or the entire block (input of some recursive call) fits in cache and incurs only the cost of a scanning. Because of possible misalignment, that is, $n(\lceil n/L \rceil + 1)$.
- **Important:** For simplicity, some authors write n/L instead of $\lceil n/L \rceil$. This can be dangerous.
- **However:** these simplifications are fine for asymptotic estimates, keeping in mind that n/L is a rational number satisfying

$$n/L - 1 \leq \lfloor n/L \rfloor \leq n/L \leq \lceil n/L \rceil \leq n/L + 1.$$

  Thus, for a fixed L, the functions $\lfloor n/L \rfloor$, n/L and $\lceil n/L \rceil$ are asymptotically of the same order of magnitude.
- We need to translate "for n small enough" into a formula. We claim that there exists a real constant $\alpha > 0$ s.t. for all n and Z we have

$$n^2 < \alpha Z \quad \Rightarrow \quad Q(n) \leq n^2/L + n.$$

Matrix transposition: a first divide-and-conquer (4/4)

- $Q(n)$ satisfies the following recurrence relation

$$Q(n) = \begin{cases} n^2/L + n & \text{if} \quad n^2 < \alpha Z \quad \text{(base case)} \\ 4Q(n/2) + \frac{n^2}{2L} + n & \text{if} \quad n^2 \geq \alpha Z \quad \text{(recurrence)} \end{cases}$$

- Indeed, **exchanging 2 blocks** amount to $2((n/2)^2/L + n/2)$ accesses.
- Unfolding the recurrence relation k times (more details in class) yields

$$Q(n) = 4^k\, Q(\frac{n}{2^k}) + k\,\frac{n^2}{2L} + (2^k - 1)n.$$

- The minimum k for reaching the base case satisfies $\frac{n^2}{4^k} = \alpha Z$, that is, $4^k = \frac{n^2}{\alpha Z}$, that is, $k = \log_4(\frac{n^2}{\alpha Z})$. This implies $2^k = \frac{n}{\sqrt{\alpha Z}}$ and thus

$$\begin{aligned} Q(n) &\leq \frac{n^2}{\alpha Z}\,(\alpha Z/L + \sqrt{\alpha Z}) + \log_4(\frac{n^2}{\alpha Z})\,\frac{n^2}{2L} + \frac{n}{\sqrt{\alpha Z}}\,n \\ &\leq n^2/L + 2\frac{n^2}{\sqrt{\alpha Z}} + \log_4(\frac{n^2}{\alpha Z})\,\frac{n^2}{2L}. \end{aligned}$$

A matrix transposition cache-oblivious algorithm (1/2)

- If $n \geq m$, the Rec-Transpose algorithm partitions

$$A = (A_1 \ A_2) \ , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

- If $m > n$, the Rec-Transpose algorithm partitions

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \ , \quad B = (B_1 \ B_2)$$

and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

A matrix transposition cache-oblivious algorithm (2/2)

- Recall that the matrices are stored in row-major layout.
- Let α be a constant sufficiently small such that the following two conditions hold:
  - (i) two sub-matrices of size $m \times n$ and $n \times m$, where $\max\{m,n\} \leq \alpha L$, fit in cache
  - (ii) even if each row starts at a different cache line.
- We distinguish three cases for the input matrix A:
  - ↳ Case I: $\max\{m,n\} \leq \alpha L$.
  - ↳ Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.
  - ↳ Case III: $m, n > \alpha L$.

Case I: $\max\{m,n\} \leq \alpha L$.

- Both matrices fit in $O(1) + 2mn/L$ lines.
- From the choice of α, the number of lines required for the entire computation is at most Z/L.
- Thus, no cache lines need to be evicted during the computation. Hence, it feels like we are simply scanning A and B.
- Therefore $Q(m,n) \in O(1 + mn/L)$.

Case II: $m \leq \alpha L < n$ or $n \leq \alpha L < m$.

- Consider $n \leq \alpha L < m$. The REC-TRANSPOSE algorithm divides the greater dimension m by 2 and recurses.
- At some point in the recursion, we have $\alpha L/2 \leq m \leq \alpha L$ and the whole computation fits in cache. At this point:
  - ↳ the input array resides in contiguous locations, requiring at most $\Theta(1 + nm/L)$ cache misses
  - ↳ the output array consists of nm elements in n rows, where in the **worst case** every row starts at a different cache line, leading to at most $\Theta(n + nm/L)$ cache misses.
- Since $m/L \in [\alpha/2, \alpha]$, the **total** cache complexity for this base case is $\Theta(1 + n)$, yielding the recurrence (where the resulting $Q(m, n)$ is a **worst case estimate**)

$$Q(m,n) = \begin{cases} \Theta(1+n) & \text{if } m \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2,n) + O(1) & \text{otherwise ;} \end{cases}$$

whose solution satisfies $Q(m, n) = \Theta(1 + mn/L)$.

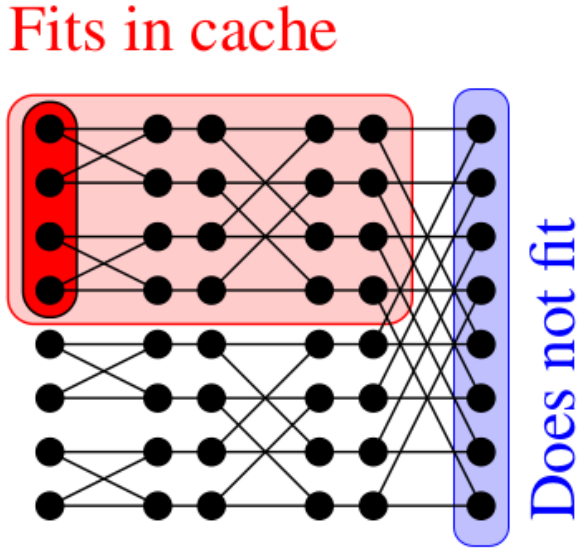
Case III: $m, n > \alpha L$.

- As in Case II, at some point in the recursion both n and m fall into the range $[\alpha L/2, \alpha L]$.
- The whole problem fits into cache and can be solved with at most $\Theta(m + n + mn/L)$ cache misses.
- The **worst case cache miss estimate** satisfies the recurrence

$$Q(m,n) = \begin{cases} \Theta(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L] \ , \\ 2Q(m/2,n) + O(1) & \text{if } m \geq n \ , \\ 2Q(m,n/2) + O(1) & \text{otherwise;} \end{cases}$$

  whose solution is $Q(m,n) = \Theta(1 + mn/L)$.
- **Therefore, the Rec-Transpose algorithm has optimal cache complexity.**
- Indeed, for an $m \times n$ matrix, the algorithm must write to mn distinct elements, which occupy at least $\lceil mn/L \rceil$ cache lines.

1-D FFTs: classical cache friendly algorithm



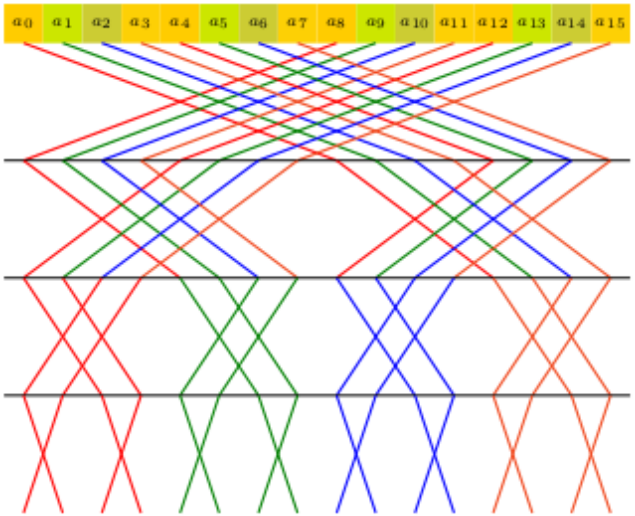
FFT$([a_0, a_1, \cdots, a_{n-1}], \omega)$

```
if n ≤ HTHRESHOLD then
    ArrayBitReversal(a_0, a_1, ..., a_{n-1})
    return FFT_iterative_in_cache([a_0, a_1, ..., a_{n-1}], ω)
end if
Shuffle(a_0, a_1, ..., a_{n-1})
[a_0, a_1, ..., a_{n/2-1}] = FFT([a_0, a_1, ..., a_{n/2-1}], ω^2)
[a_{n/2}, a_{n/2+1}, ..., a_{n-1}] = FFT([a_{n/2}, a_{n/2+1}, ..., a_{n-1}], ω^2)
return [a_0 + a_{n/2}, a_1 + ω · a_{n/2+1}, ..., a_{n/2-1} − ω^{n/2-1} · a_{n-1}]
```

Cache friendly 1-D FFT

- If the input vector does not fit in cache, a recursive algorithm is applied
- Once the vector fits in cache, an iterative algorithm (not requiring shuffling) takes over.
- On an ideal cache of Z words with L words per cache line this yields a cache complexity of $\Omega(n/L(\log_2(n) - \log_2(Z)))$ which is not optimal.

1-D FFTs: cache complexity optimal algorithm



Fast Fourier Transform in R

procedure FFT$((\alpha_0\alpha_1...\alpha_{N-1}), \omega, N = J \cdot K, \Omega = \omega^{N/K})$
  for $0 \leq k < K-1$ **do** ▷ Inner transforms
    for $0 \leq k' < J-1$ **do**
      $\gamma[k][k'] = \alpha_{k'K+k}$
    end for
    $c[k] = FFT(\gamma[k], \omega^K, J, \Omega)$
  end for
  for $0 \leq j < J-1$ **do** ▷ Outer transforms
    for $0 \leq k < K-1$ **do**
      $\delta[j][k] = c[k][j] * \omega^{jk}$ ▷ Computation of coefficients
    end for
    $d[j] = FFT(\delta[j], \omega^J, K, \Omega)$
    for $0 \leq j' < K-1$ **do**
      $\beta_{j'J+j} = d[j][j']$
    end for
  end for
  return $b = (\beta_0, ..., \beta_{N-1})$
end procedure

Cache optimal 1-D FFT

- Instead of processing row-by-row, one computes as deep as possible while staying in cache (resp. registers): this yields a blocking strategy.
- On the left picture, assuming $Z = 4$, on the first (resp. last) two rows, we successively compute the red, green, blue, orange 4-point blocks.
- On an ideal cache of Z words with L words per cache line the cache complexity drops to $\boxed{O(n/L(\log_2(n)/\log_2(Z)))}$ which is optimal.

1-D FFTs in BPAS

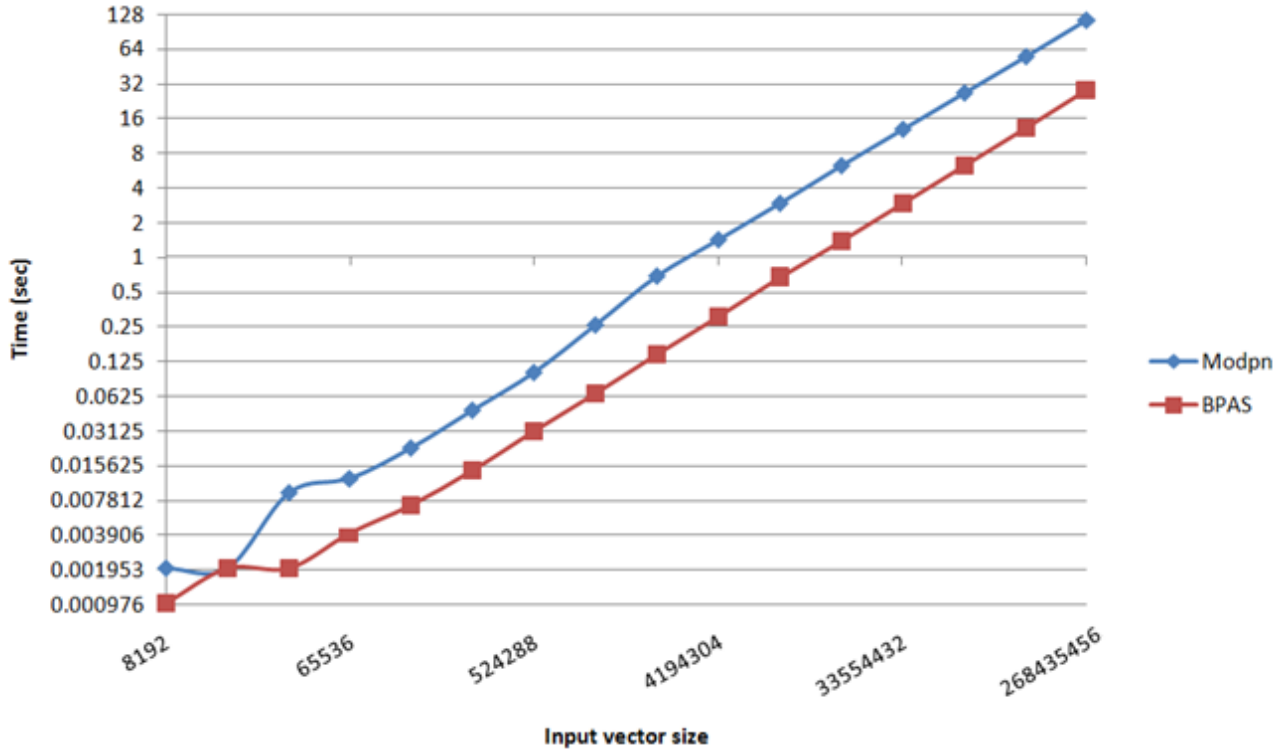


Figure: 1-D modular FFTs: Modpn (**serial**) vs BPAS (**serial**).

- In addition to the above optimal blocking strategy, instruction level parallelism (ILP) is carefully considered: vectorized instructions are explicitly used and instruction pipeline usage is highly optimized.
- BPAS 1-D FFT code automatically generated by configurable Python scripts.

Outline

Notations

- For two positive integers a, b, we write a/b instead of $\lfloor a/b \rfloor$.
- Let $\mathbb{K}$ be a finite field so that each element of $\mathbb{K}$ can be stored in a machine word.
- We assume that each polynomial P of $\mathbb{K}[x]$ is stored in a vector V_P of $d+1$ words, aligned in memory, where d is the degree of P, and so that the coefficient of x^i in P is stored in the $(d-i)$-th slot of V_P, for $0 \leq i \leq d$.
- Let $A = \sum_{i=0}^{i=m-1} a_i x^i$ and $B = \sum_{i=0}^{i=n-1} b_i x^i$ be in $\mathbb{K}[x]$ with $m \geq n$.

Plain polynomial multiplication

- Recall $A = \sum_{i=0}^{i=m-1} a_i x^i$ and $B = \sum_{i=0}^{i=n-1} b_i x^i$ in $\mathbb{K}[x]$ with $m \geq n$.
- Counting cache misses, the plain multiplication incurs
$$O((m/L+1)n)$$
- This estimate can be substantially improved by performing the plain multiplication in a divide-and-conquer manner, following the scheme of the matrix multiplication algorithm of [9].
- This recursive algorithm is presented in [7]; it runs within
$$O(mn/(ZL))$$
- It leads to clear gains on Graphics Processing Units (GPUs) due to the fine grained control of hardware resources.
- However, with a CPU implementation, for relatively small n and m, any plain multiplication algorithm is outperformed by an FFT-based polynomial multiplication.

Plain polynomial division

- Let $Q = \mathrm{quo}(A, B)$ and $R = \mathrm{rem}(A, B)$
- The schoolbook plain Euclidean division, using a two-loop nest, computes Q and R, within

$$O((m - n + 1)(n/L + 3))$$

- By means of a *blocking strategy*, this estimate can be improved to

$$O(((2Z + 9L)(m - n + 1)(n/(Z^2 L) + 1))$$

  See [15, 16].
- This strategy is inspired by the Half-Gcd algorithm, see Lemma 11.1 in Chapter 11 of [11]. See [**DBLP:conf/issac/Maza21**].

Outline

DAG encodings of computations

By computation, we mean the execution of a program, not a program itself, similarly to the **instruction stream DAG** of a `Cilk` program.

Notations

From now on we consider a connected directed acyclic graph $G = (V, E)$:

- Each vertex represents an **operation** and its **result**.
- An edge from a vertex v_1 to a vertex v_2 indicates that the result of v_1 is needed for performing the operation of v_2.
- A vertex v of G is an **input** (resp. **output**) if it has no predecessors (resp, no successors).
- The sets of inputs and outputs are respectively denoted by $I(G)$ and $O(G)$. Note that these sets are disjoint.

The Red-Blue Pebble Game (1/3)

The **red**-**blue** pebble game is played on a directed and connected acyclic graph $G = (V, E)$.

- At any point of the game, some vertices have **red** pebbles, others have **blue**, others have pebbles of both types, others have no pebbles.
- A **configuration** is a pair of subsets (R, B) of the vertex set V such that any vertex $v \in R$ (resp. $v \in B$) has a **blue** pebble (resp. **red** pebble).
- The **initial configuration** is the one given by $(\varnothing, I(G))$.
- The **final configuration** is the one given by $(\varnothing, O(G))$.

The Red-Blue Pebble Game (2/3)

The rules of the **red**-**blue** pebble game are as follows.

(R_1) **Input rule:** A **red** pebble may be placed on any vertex that has a **blue** pebble.

(R_2) **Output rule:** A **blue** pebble may be placed on any vertex that has a **red** pebble.

(R_3) **Compute rule:** If all immediate predecessors of a vertex v have **red** pebbles then a **red** pebble may be placed on v.

(R_4) **Delete rule:** A pebble **red** or **blue** may be removed at any time from any vertex.

The Red-Blue Pebble Game (3/3)

Key concepts:

- A **transition** is an ordered pair of configurations, the second of which follows from the first according to one of the rules (R_1) to (R_4).
- A **caculation** is a sequence of configurations, each successive pair of which form a transition.
- A **complete caculation** is one that begins with the initial configuration and ends with the final configuration.

Application to cache complexity (1/4)

- A DAg on which the **red**-**blue** pebble game is played can model a computation performed on a two-level memory structure, say, a **fast** memory (or cache) and a **slow** memory.
- Recall: Each vertex represents an **operation** and its **result**.
- Recall: An edge from a vertex v_1 to a vertex v_2 indicates that the result of v_1 is needed for performing the operation of v_2.
- An operation can be performed only if **all operands reside in cache** (or **fast** memory).
- The maximum allowable number of **red** (or **blue**) pebbles on the DAG at any point in the game corresponds to the number of words available for use in the fast (or slow) memory, respectively.

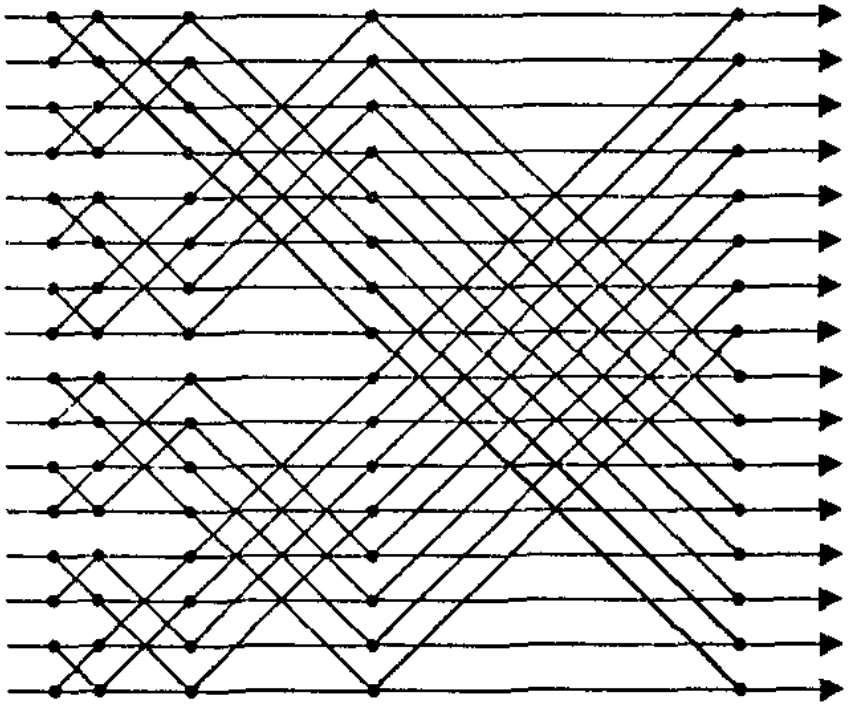
Application to cache complexity (2/4)

- Placing a **red** pebble using Rule (R_3) corresponds to performing an operation and storing the result in cache
- Placing a **blue** pebble using Rule (R_2) corresponds to storing a copy of a result (currently in the fast memory) into the slow memory.
- Placing a **red** pebble using Rule (R_1) corresponds to retrieving a copy of a result (currently in the slow memory) into the fast memory.
- Removing a red or **red** or **blue** pebble using Rule (R_4) means freeing a memory location in the fast or slow memory respectively.

Application to cache complexity (3/4)

- In what follows, the fast memory can only hold S words, where S is a constant, while the slow memory is arbitrarily large.
- For any given connected DAG, we are interested in the **I/O time**, denoted by Q, which is the minimum number of transitions according to Rules (R_1) or (R_2) required by any complete calculation.
- In the original work of (J.W. Hong, H.T. Kung, 1981) a "static problem" is associated with the **red**-**blue** pebble game, the *S-Partitioning Problem*. Then lower bounds for the S-Partitioning Problem lead to lower bounds for the **red**-**blue** pebble game.
- To establish bounds like those (but weaker) of (J.W. Hong, H.T. Kung, 1981) a simpler approach due to J.E. Savage (see his book *Models of Computations*) [27] reducing to *simpler* the **red** pebble game.

Application to cache complexity (4/4)



Theorem. Assume $S \geq 3$. For the n-point FFT graph we have $Q \log(S) \in \Omega(n \log(n))$. Moreover, there is a pebbling strategy for which $Q \log(S) \in \Theta(n \log(n))$ holds.

Outline

Overview

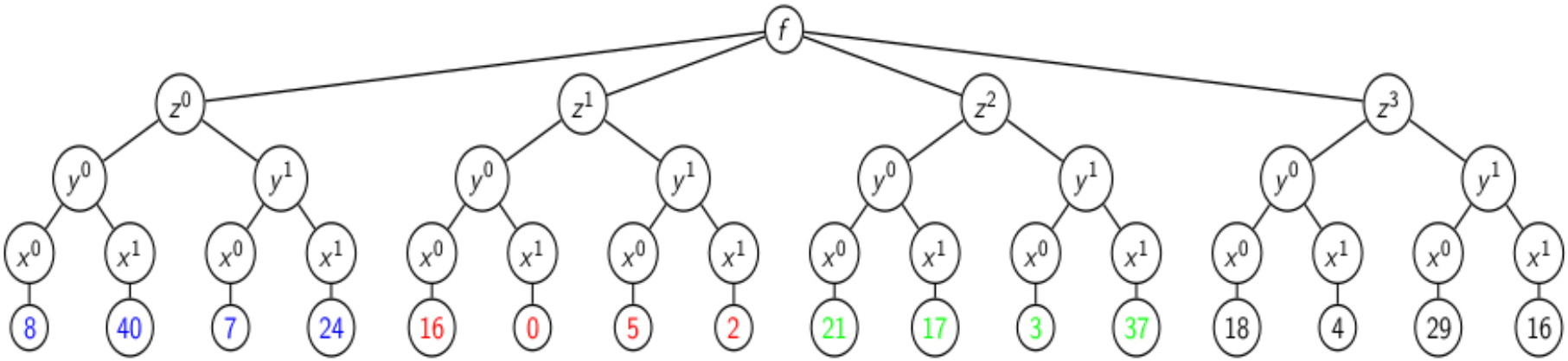
- In [24], the authors show that multiplying dense polynomials $f, g \in Z/pZ[x_1, \ldots, x_n]$ makes an optimal use of multicore processors when $n = 2$, $\deg(f, x_1) = \deg(g, x_1)$ and $\deg(f, x_2) = \deg(g, x_2)$.
- Under some assumption, the authors of [24] give a practical heuristic reducing multivariate multiplication to multiplying a balanced pair of bivariate polynomials. The authors of [25] relax their assumption.

Strategy

1. Since performance is, in practice, hardware-dependent, we focus on a specific architecture, namely multi-core processors.
2. We aim at optimizing the implementation of an algebraic algorithm in terms of parallelism and, thus in terms of memory accesses.
3. To do so, we reshape the input data at a cost which is amortized by the performance benefits.

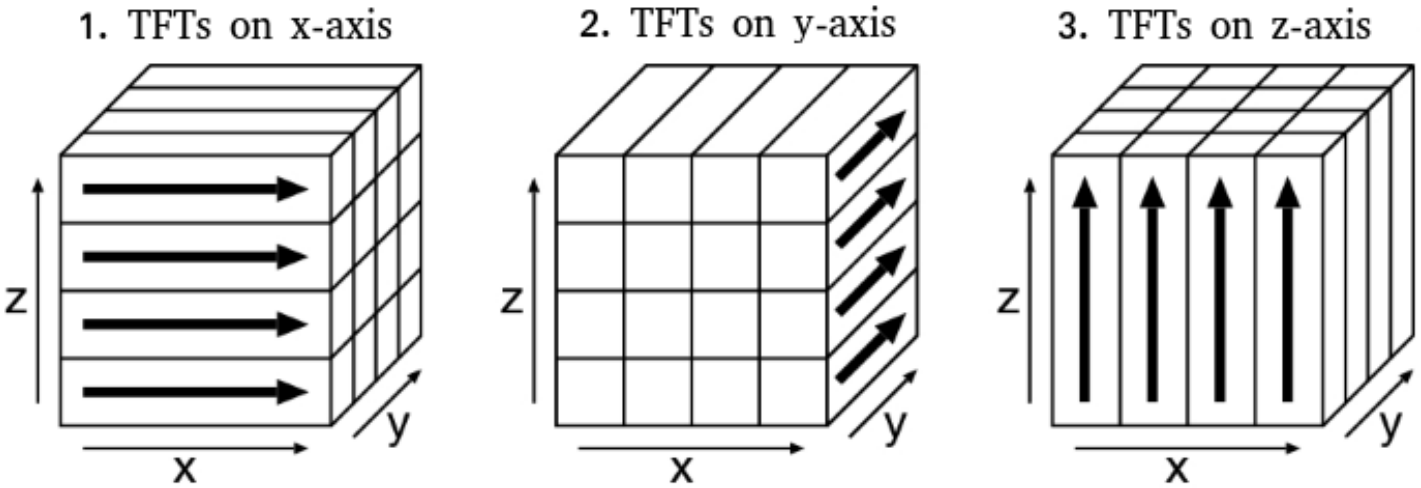
Recursive representation of multivariate polynomials

Example. Let $f \in \mathbb{K}[z > y > x]$ where $\mathbb{K} = \mathbb{Z}/41\mathbb{Z}$, with $d_x = d_y = 1, d_z = 3$. A *recursive dense representation* (RDR) of f is:



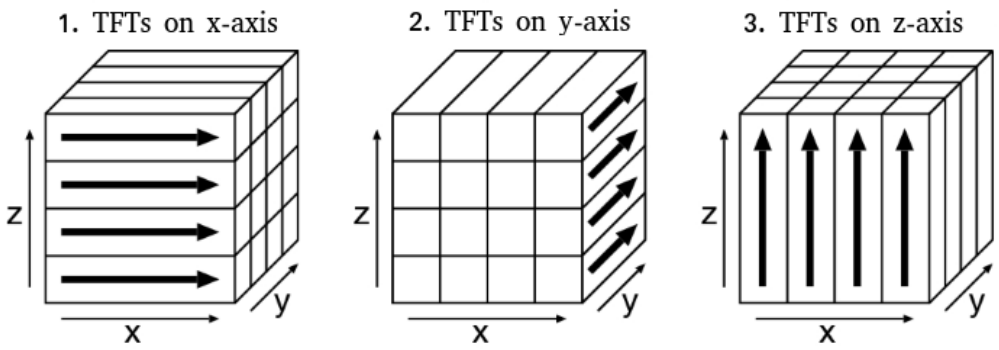
- the coefficients are stored in a contiguous array.
- the coefficient of the monomial $x^{e_1} y^{e_2} z^{e_3}$ has index $e_1 + s_x e_2 + s_x s_y e_3$, where s_x and s_y are integers satisfying $s_x > \deg(f, x)$ and $s_y > \deg(f, y)$.

Multi-dimensional TFTs and ITFTs (1/2)



- To minimize algebraic complexity, our dense polynomial multiplication relies on TFTs and ITFTs.
- Targeting multi-core processors, we need to extract coarse-grained parallelism, thus we apply the row-column algorithm (illustrated above) on multi-dimensional TFTs, thus $n \geq 2$.
- To minimize cache complexity, we transpose the data between TFTs along x (resp. y) and TFTs along y (resp. z) in order to maintain spatial locality.

Multi-dimensional TFTs and ITFTs (2/2)



- Let $T(s)$ be the number of cache misses for transposing a matrix of s elements using the (optimal) $\mathrm{REC\text{-}TRANPOSE}$ algorithm of [9] with an ideal cache of cache-line size L.
- The cache complexity $Q(n, s)$ of n-D TFT satisfies

$$nT(s) + n\frac{s}{L+1} \leq Q(n, s) \leq nT(s) + 2n\frac{s}{L}.$$

- For a fixed s, this estimate suggests to minimize n, so letting $n = 2$.
- The parallelism $P(n, s)$ of n-D TFT satisfies $P(n, s) \geq \frac{s}{\max_{i=1\cdots n}(s_i)}$, where $s_1, \ldots, s_n$ are the dimension sizes w.r.t. $x_1, \ldots, x_n$.
- When $n = 2$, this estimate suggests $s_1 = s_2 = \sqrt{s}$. (Balanced case)

Balanced multiplication

1. Mapping monomials of $\mathbb{K}[x_1, \ldots, x_m, \ldots, x_n]$ to $\mathbb{K}[x_1, \ldots, x_u, x_v, \ldots, x_n]$:

$$x_1^{e_1} x_2^{e_2} \ldots x_n^{e_n} \longmapsto x_1^{e_1} \ldots x_{m-1}^{e_{m-1}} x_u^{e_u} x_v^{e_v} x_{m+1}^{e_{m+1}} \ldots x_n^{e_n}, \quad \text{where :}$$

   - e_u and e_v are the quotient and the remainder in the Euclidean division of e_m by b,
   - m and b are parameters to be determined later.

2. Mapping monomials of $\mathbb{K}[x_1, \ldots, x_u, x_v, \ldots, x_n]$ to $\mathbb{K}[x, y]$:

$$x_1^{e_1} \ldots x_{m-1}^{e_{m-1}} x_u^{e_u} x_v^{e_v} x_{m+1}^{e_{m+1}} \ldots x_n^{e_n} \longmapsto x^{c_1} y^{c_2}, \quad \text{where :}$$

$$\begin{aligned} c_1 &= \alpha_1 e_1 + \alpha_2 e_2 + \ldots + \alpha_{m-1} e_{m-1} + \alpha_u e_u \\ c_2 &= \alpha_v e_v + \alpha_{m+1} e_{m+1} + \alpha_{m+2} e_{m+2} + \ldots + \alpha_n e_n \end{aligned}$$

with:

$$\alpha_1 = \alpha_{m+1} = 1 \text{ and } \alpha_{i+1} = \alpha_i (d_i + d'_i + 1) \text{ otherwise.}$$

so that the bivariate images of f, g form a (nearly) balanced pair.

Determination of the parameters m and b

- Consider multiplying $f, g \in \mathbb{K}[x_1, \ldots, x_n]$ and let $h = fg$.
- Let $h := fg$. We define $s_i^{(f)} := d_i + 1$ and $s_i^{(g)} := d'_i + 1$, so that we have $s_i^{(h)} := s_i \geq s_i^{(f)} + s_i^{(g)} - 1$, for $1 \leq i \leq n$. That is, h can be stored in an RDR with dimension sizes $s_1^{(h)}, s_2^{(h)}, \ldots, s_n^{(h)}$.
- The bivariate images of f and g will be represented with an $\underline{s}$-RDR, where $\underline{s} = (s_x, s_y)$, $s_x := s_u \Pi_{i=1}^{m-1} s_i$, $s_y := s_v \Pi_{i=m+1}^{n} s_i$, thus we have $s_x s_y = \frac{s_u s_v}{s_m} \Pi_{i=1}^{n} s_i$.
- Define $\sigma_1 = \Pi_{i=1}^{m-1} s_i^{(h)}$ and $\sigma_2 = \Pi_{i=m+1}^{n} s_i^{(h)}$,
- The size difference w.r.t. x and y of the bivariate image of h is:
$$D = s_x^{(h)} - s_y^{(h)} = s_u^{(h)} \sigma_1 - s_v^{(h)} \sigma_2.$$
- After simplification, we have:
$$D = \sigma_1 (s_m^{(f)}/b + s_m^{(g)}/b - 1) - \sigma_2 (2b - 1).$$
- there is only one m satisfying
$\Pi_{i=1}^{m-1} s_i^{(h)} < \sqrt{\Pi_{i=1}^{n} s_i^{(h)}}$ and $\Pi_{i=1}^{m} s_i^{(h)} \geq \sqrt{\Pi_{i=1}^{n} s_i^{(h)}}$.
- Then, there is one integer b making D is as close as possible to 0.

Experimentation: Test Case # 2

In this test, we multiply two 8-variate polynomials f and g where all partial degrees of are equal and the partial degrees range in $1 \ldots 5$.

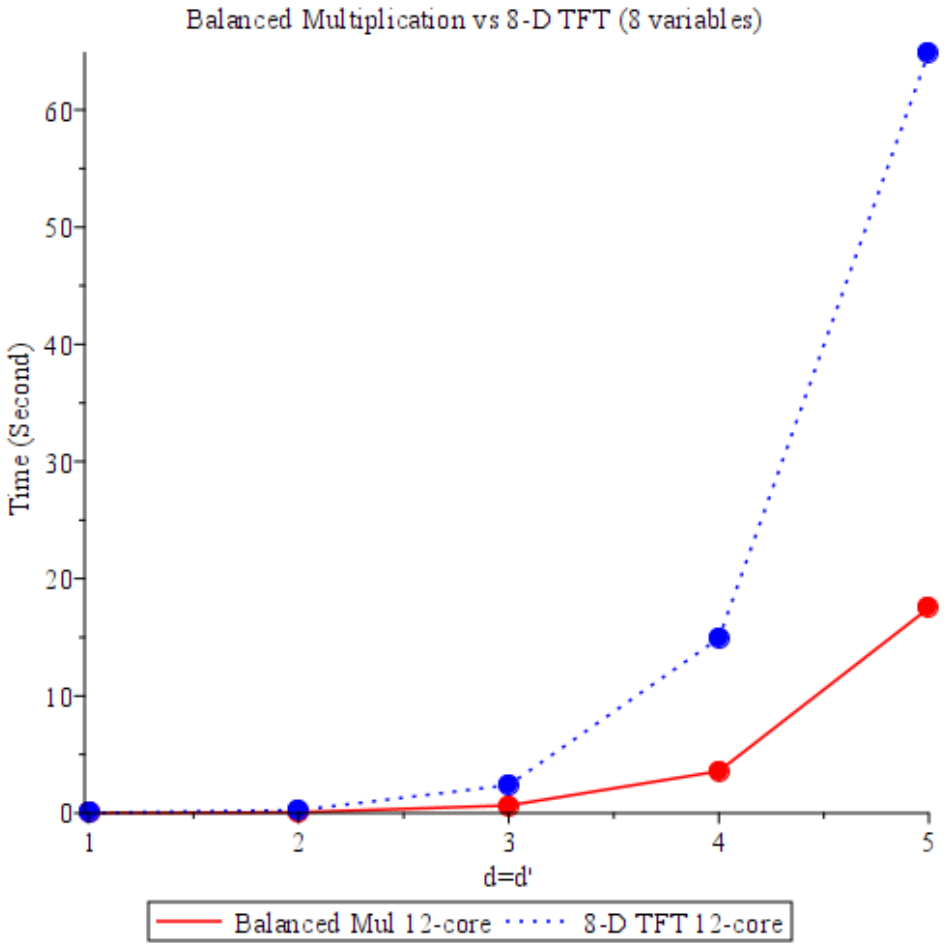
Table: Performance analysis: Balanced bivariate multiplication Vs. 8-D TFT-based multiplication

Method	Balanced bivariate	8-D TFT-based
CPU-cycles	69,321,767,227	255,290,461,510
Instructions	118,622,179,258	316,889,335,524
CPI	0.58	0.80
Branch miss rate	0.11%	0.69%
MPKI	1.07	1.28

MPKI stands for *misses per one thousand instructions*.

Test Case

Running time comparison on Intel Xeon multi-core (12 cores).



Experimentation: summary

- In most of cases the proposed strategy outperforms the direct approach based on multi-dimensional TFTs.
- The unfavourable cases happen when the number of variables is small, while
- the most favourable cases occur when the number of variables increases and the input pair of polynomials is far from balanced.

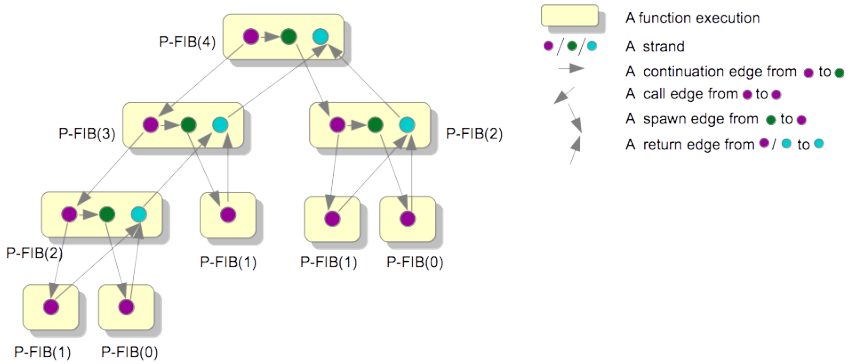
Outline

Overview

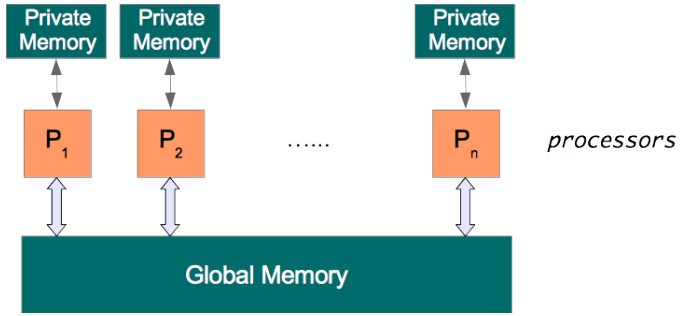
- We present a model of multithreaded computation with an emphasis on estimating parallelism overheads of programs written for modern many-core architectures.
- We evaluate the benefits of our model with fundamental algorithms from scientific computing.
  - ↳ For two case studies, our model is used to minimize parallelism overheads by determining an appropriate value range for a given program parameter.
  - ↳ For the others, our model is used to compare different algorithms solving the same problem.
- In each case, the studied algorithms were implemented[1] and the results of their experimental comparison are coherent with the theoretical analysis based on our model.
- This work is published in ParCo'15 as [15].

[1] Publicly available written in CUDA from `http://www.cumodp.org/`

Models of computation



The fork-join model: 4-th Fibonacci number



The PRAM model

- The classical models of parallel computation, the fork-join concurrency model and the parallel random access machine (PRAM) model do not distinguish between the task-based and data-based parallelism.
- Recent many-core machine models:
  - ↳ Lin Ma, Kunal Agrawal, and Roger D. Chamberlain. A memory access model for highly-threaded many-core architectures. *Future Generation Computer Systems*, 30:202-215, 2014.
  - ↳ Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152-163. ACM, 2009.

The MCM model

We propose a many-core machine (MCM) model which aims at

- tuning program parameters to minimize parallelism overheads of algorithms targeting GPU-like architectures as well as
- comparing different algorithms independently of the value of machine parameters of the targeted hardware device.

In the design of this model, we insist on the following features:

- Two-level DAG programs
- Parallelism overhead
- A Graham-Brent theorem

Characteristics of the abstract many-core machines (1/2)

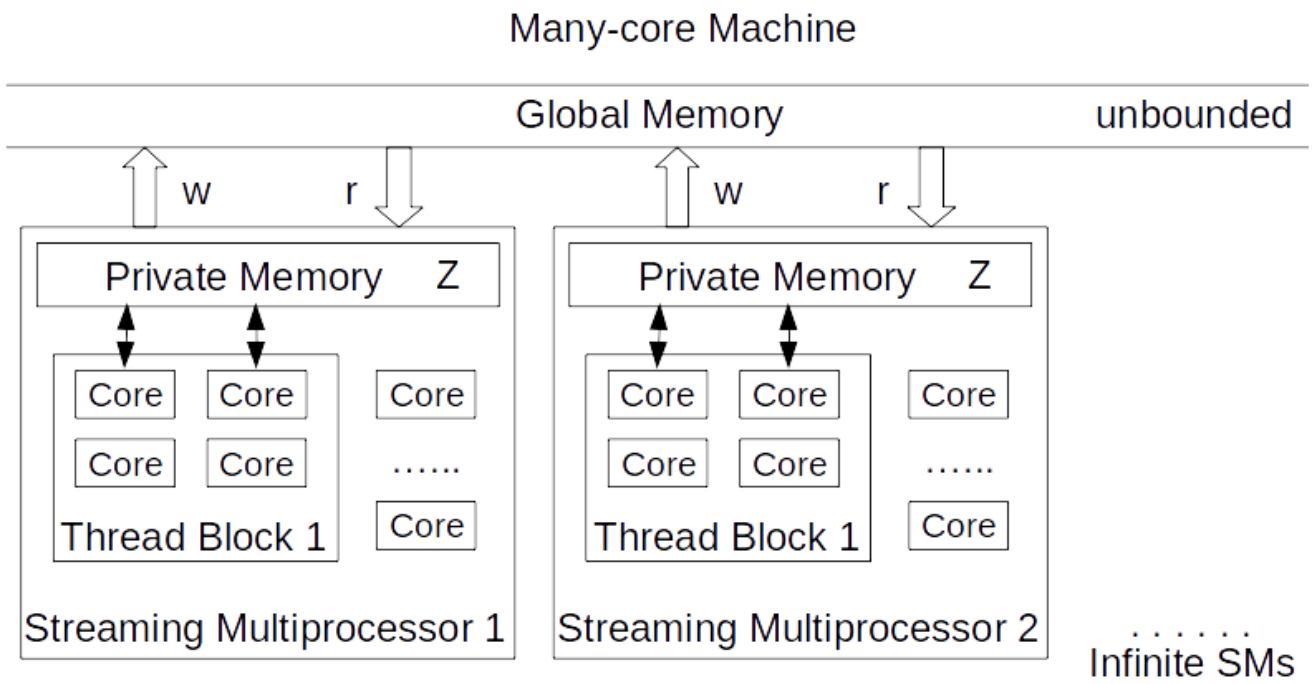


Figure: A many-core machine

- It has a global memory with high latency, while private memories have low latency.

Characteristics of the abstract many-core machines (2/2)

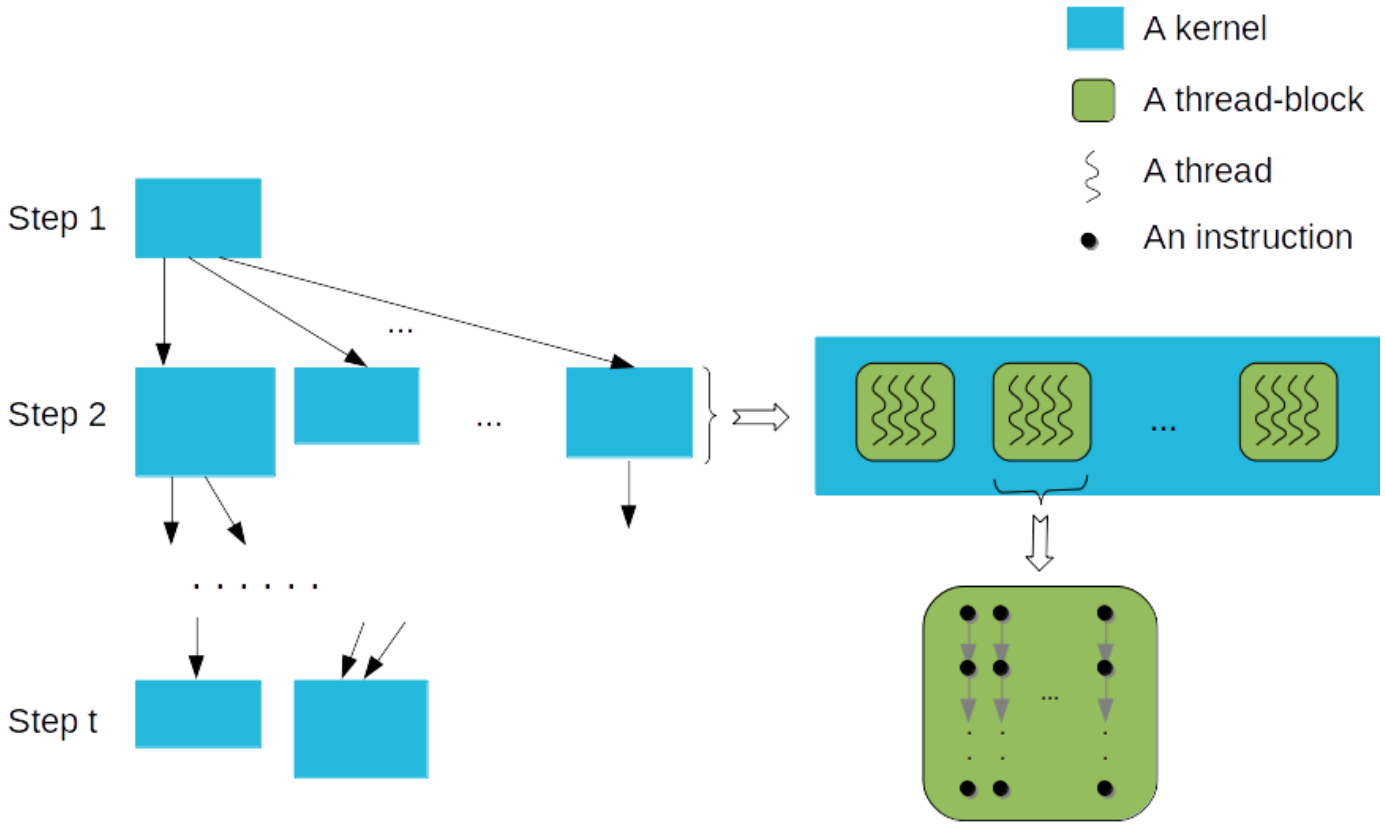


Figure: Overview of a many-core machine program, also called *kernel DAG*

Machine parameters of the abstract many-core machines

Z: Private memory size of any SM

It sets up an upper bound on several program parameters (number of threads of a thread-block, number of words in a data transfer between the global memory and the private memory of a Sm).

U: Data transfer time

- Time (expressed in clock cycles) to transfer one machine word between the global memory and the private memory of any SM.
- As an abstract machine, the MCM aims at capturing either the best or the worst scenario for data transfer time of a thread-block, that is,

$$\begin{array}{rcl} T_D & \leq & (\alpha+\beta)\,U, \text{ if coalesced accesses occur;} \\ & \text{or} & \ell\,(\alpha+\beta)\,U, \text{ otherwise,} \end{array}$$

where α and β are the numbers of words respectively read and written to the global memory by one thread of a thread-block B and ℓ be the number of threads per thread-block.

Complexity measures for the many-core machine model

For any kernel $\mathcal{K}$ of an MCM program,

- **work** $W(\mathcal{K})$ is the total number of local operations of all its threads;
- **span** $S(\mathcal{K})$ is the maximum number of local operations of one thread;
- **parallelism overhead** $O(\mathcal{K})$ is the total data transfer time among all its thread-blocks.

For the entire program $\mathcal{P}$,

- **work** $W(\mathcal{P})$ is the total work of all its kernels;
- **span** $S(\mathcal{P})$ is the longest path, counting the weight (span) of each vertex (kernel), in the kernel DAG;
- **parallelism overhead** $O(\mathcal{P})$ is the total parallelism overhead of all its kernels.

Characteristic quantities of the thread-block DAG

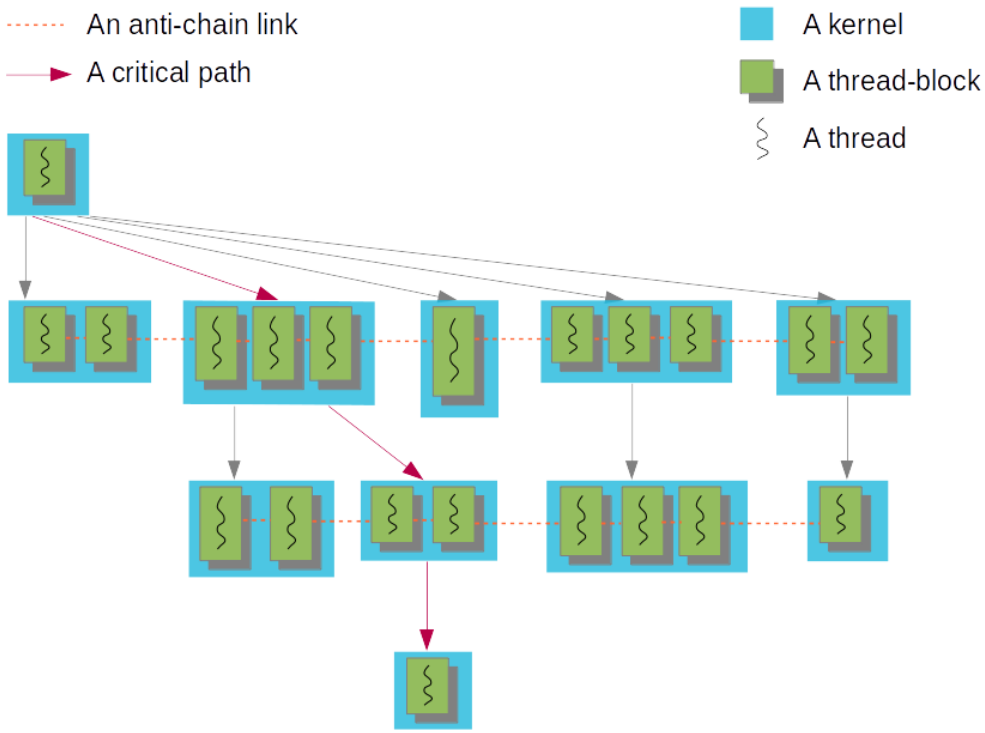


Figure: Thread-block DAG of a many-core machine program

$\mathrm{N}(\mathcal{P})$: number of vertices in the thread-block DAG of $\mathcal{P}$,

$\mathrm{L}(\mathcal{P})$: critical path length (where length of a path is the number of edges in that path) in the thread-block DAG of $\mathcal{P}$.

Complexity measures for the many-core machine model

Theorem (A Graham-Brent theorem with parallelism overhead)

We have the following estimate for the running time T_{P} *of the program* $\mathcal{P}$ *when executed on* P *SMs:*

$$T_{\mathsf{P}} \leq (\mathrm{N}(\mathcal{P})/\mathsf{P} + \mathrm{L}(\mathcal{P}))\mathrm{C}(\mathcal{P}) \tag{1}$$

where $\mathrm{C}(\mathcal{P})$ *is the maximum running time of local operations (including read/write requests) and data transfer by one thread-block.*

Corollary

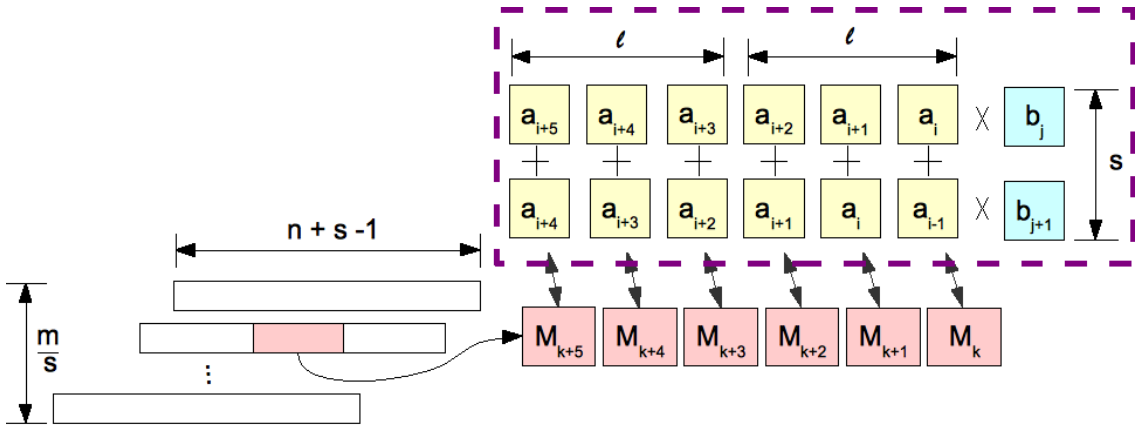
Let K *be the maximum number of thread-blocks along an anti-chain of the thread-block DAG of* $\mathcal{P}$*. Then the running time* $T_{\mathcal{P}}$ *of the program* $\mathcal{P}$ *satisfies:*

$$T_{\mathcal{P}} \leq (\mathrm{N}(\mathcal{P})/\mathsf{K} + \mathrm{L}(\mathcal{P}))\mathrm{C}(\mathcal{P}) \tag{2}$$

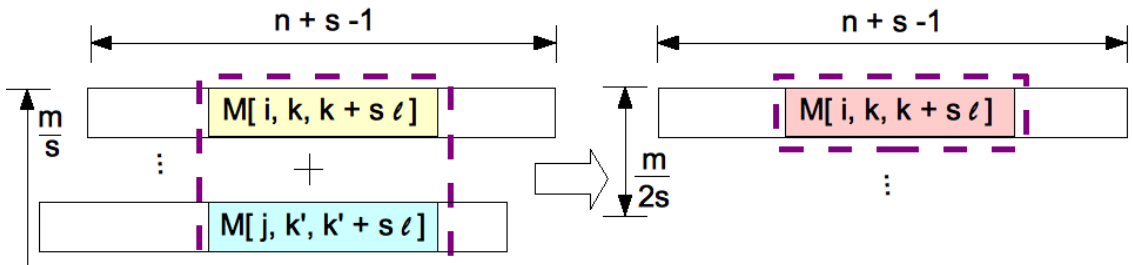
Plain univariate polynomial multiplication (1/3)

Tuning the parameter s

Multiplication phase: every coefficient of a is multiplied with every coefficients of b; each thread accumulates s partial sums into an auxiliary array M.



Addition phase: these partial sums are added together repeatedly to form the polynomial f.



Plain univariate polynomial multiplication (2/3)

The work, span and parallelism overhead ratios between $s_0 = 1$ (initial program) and an arbitrary s are, respectively[2],

$$\frac{\mathrm{W}_1}{\mathrm{W}_s} = \frac{n}{n+s-1},$$

$$\frac{\mathrm{S}_1}{\mathrm{S}_s} = \frac{\log_2(m)+1}{s\left(\log_2\left(m/s\right)+2\,s-1\right)},$$

$$\frac{\mathrm{O}_1}{\mathrm{O}_s} = \frac{n\,s^2\left(7\,m-3\right)}{\left(n+s-1\right)\left(5\,m\,s+2\,m-3\,s^2\right)}.$$

- Let m esacpe to infinity with $m \leq n$.
- Increasing s leaves work essentially constant, while span increases and parallelism overhead decreases in the same order when $m \to \infty$.
- Hence, should s be large or close to $s_0 = 1$?

[2] See the detailed analysis in the form of executable MAPLE worksheets of three applications: `http://www.csd.uwo.ca/~nxie6/projects/mcm/`

Plain univariate polynomial multiplication (3/3)

Applying our version of the Graham-Brent theorem, the ratio R of the estimated running times on $\Theta(\frac{(n+s-1)\,m}{\ell\,s^2})$ SMs is

$$R = \frac{(m\,\log_2(m) + 3\,m - 1)\,(1 + 4\,U)}{(m\,\log_2(\frac{m}{s}) + 3\,m - s)\,(2\,U\,s + 2\,U + 2\,s^2 - s)},$$

which is asymptotically equivalent to $\frac{2\,U\,\log_2(m)}{s\,(s+U)\,\log_2\,(m/s)}$. This latter ratio is less than 1 for $s > 1$, since $U > 0$.

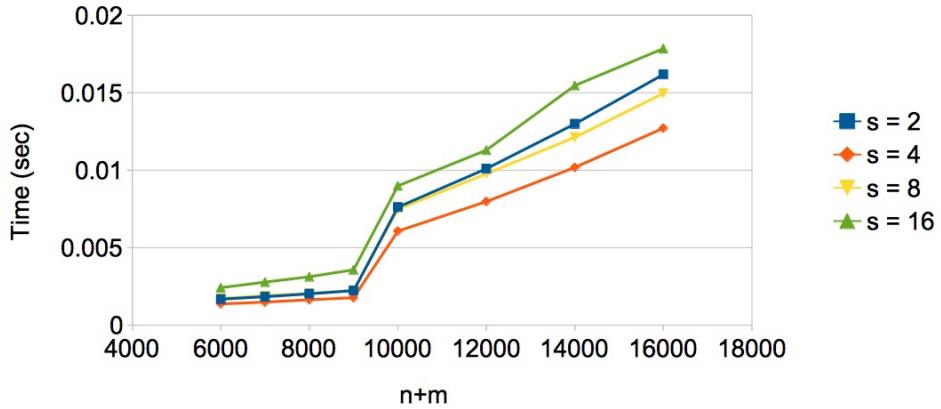
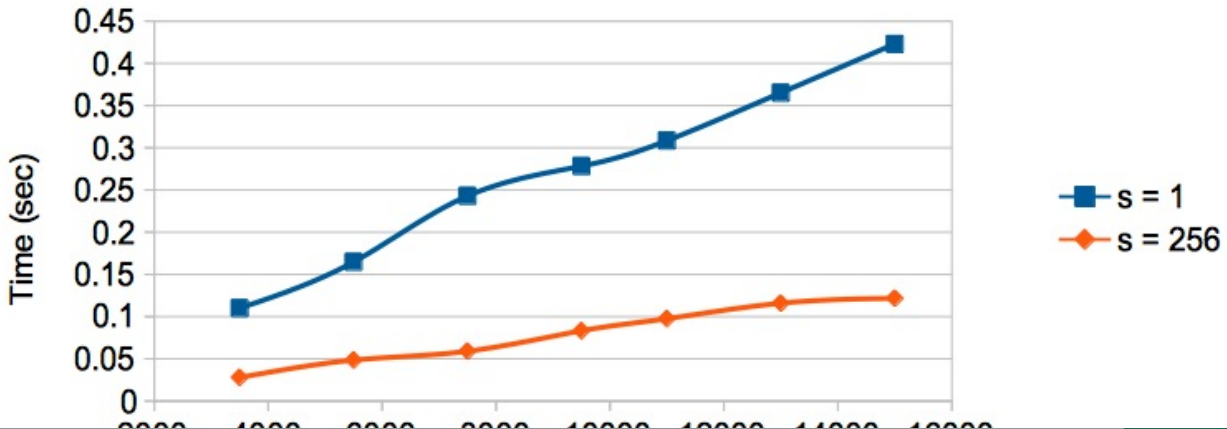


Figure: Running time of the plain polynomial multiplication algorithm with polynomials a $(\deg(a) = n - 1)$ and b $(\deg(b) = m - 1)$ and the parameter s on GeForce GTX 670.

The Euclidean algorithm

Let $s > 0$. We proceed by repeatedly calling a subroutine which

- takes as input a pair (a, b) of polynomials and
- returns another pair (a', b') of polynomials such that $\gcd(a, b) = \gcd(a', b')$ and, either $b' = 0$ or we have $\deg(a') + \deg(b') \leq \deg(a) + \deg(b) - s$.
- When $s = \Theta(\ell)$ (the number of threads per thread-block), the work is increased by a constant factor and the parallelism overhead will reduce by a factor in $\Theta(s)$.
- Further, the estimated running time ratio T_1/T_s on $\Theta(\frac{m}{\ell})$ SMs is greater than 1 if and only if $s > 1$.



Fast Fourier Transform (FFT) (1/3)

Let f be a vector with coefficients in a field (either a prime field like Z/pZ or $\mathbb{C}$) and size n, which is a power of 2. Let ω be a n-th primitive root of unity.

The n-point *Discrete Fourier Transform* (DFT) at ω is the linear map defined by $x \longmapsto \mathrm{DFT}_n\, x$ with

$$\mathrm{DFT}_n = [\omega^{ij}]_{0 \leq i,\, j < n}.$$

We are interested in comparing popular algorithms for computing DFTs on many-core architectures:

- Cooley & Tukey FFT algorithm
- Stockham FFT algorithm

FFT: Cooley & Tukey vs Stockham (2/3)

The work, span and parallelism overhead ratios between Cooley & Tukey's and Stockham's FFT algorithms are, respectively,

$$\frac{\mathrm{W}_{ct}}{\mathrm{W}_{sh}} \sim \frac{4\,n\,(34\,\log_2(n)\,\ell\,\log_2(\ell)+47\,\log_2(n)\,\ell)}{172\,n\,\log_2(n)\,\ell+n+48\,\ell^2},$$

$$\frac{\mathrm{S}_{ct}}{\mathrm{S}_{sh}} \sim \frac{34\,\log_2(n)\,\log_2(\ell)+47\,\log_2(n)}{43\,\log_2(n)+16\,\log_2(\ell)},$$

$$\frac{\mathrm{O}_{ct}}{\mathrm{O}_{sh}} = \frac{8\,n\,(4\,\log_2(n)+\ell\,\log_2(\ell)-\log_2(\ell)-15)}{20\,n\,\log_2(n)+5\,n-4\,\ell},$$

where ℓ is the number of threads per thread-block.

- Both the work and span of the algorithm of Cooley & Tukey are increased by $\Theta(\log_2(\ell))$ factor w.r.t their counterparts in Stockham algorithm.

FFT: Cooley & Tukey vs Stockham (3/3)

The ratio $R = T_{ct}/T_{sh}$ of the estimated running times (using our Graham-Brent theorem) on $\Theta(\frac{n}{\ell})$ SMs is[3]:

$$R \sim \frac{\log_2(n)(2\,U\,\ell + 34\,\log_2(\ell) + 2\,U)}{5\,\log_2(n)\,(U + 2\,\log_2(\ell))},$$

when n escapes to infinity. This latter ratio is greater than 1 iff $\ell > 1$.

Table: Running time (secs) with input size n on GeForce GTX 670.

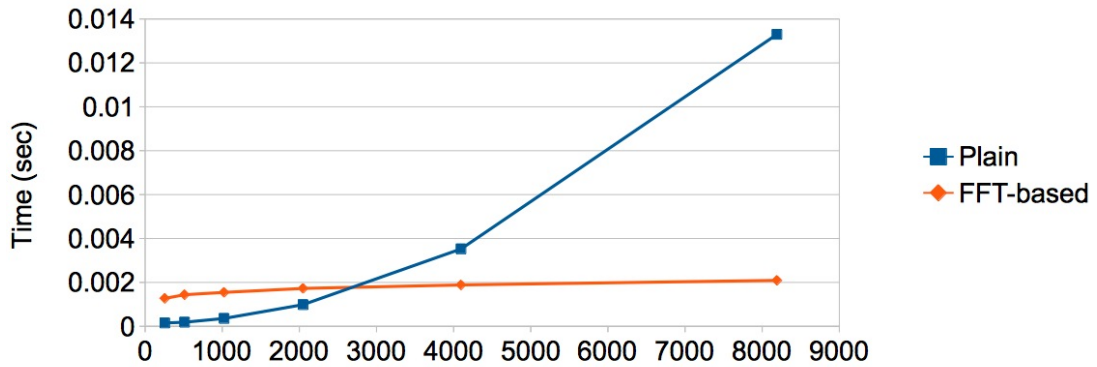
n	Cooley & Tukey	Stockham
2^{14}	0.583296	0.666496
2^{15}	0.826784	0.7624
2^{16}	1.19542	0.929632
2^{17}	2.07514	1.24928
2^{18}	4.66762	1.86458
2^{19}	9.11498	3.04365
2^{20}	16.8699	5.38781

[3] ℓ is the number of threads per thread-block.

Univariate polynomial multiplication: Plain vs FFT-based

Polynomial multiplication can be done either via the long (= plain) scheme or via FFT computations.
Let n be the largest size of an input polynomial and ℓ be the number of threads per thread-block.

- The theoretical analysis of our model indicates that the plain multiplication performs more work and parallelism overhead.
- However, on $O(\frac{n^2}{\ell})$ SMs, the ratio T_{plain}/T_{fft} of the estimated running times is essentially constant.
- On the other hand, the running time ratio T_{plain}/T_{fft} on $\Theta(\frac{n}{\ell})$ SMs suggests FFT-based multiplication outperforms plain multiplication for n large enough.



Outline

Optimizing cache complexity

- For all results discussed above, the key towards cache-oblivious or cache optimal algorithms is a *blocking strategy*.
- This blocking strategy may take different forms: from the *buckets* of counting sort to *matrix blocks* in dense linear algebra.
- While blocking strategies naturally lead to recursive algorithms, the implementation of the latter are often made in the form of *for-loop nests*, which is more suitable for compiler optimization.

In the context of multi/many-core processors

- When multiple threads are cooperating, cores executing those threads share a common physical address space, causing a *cache coherence problem*.
- Two well-known consequences of this problem are *true sharing* and *false sharing*:
  - ↳ In the former, two cores are accessing the same memory address, with at least one of them for writing.
  - ↳ In the latter, two cores are accessing the same cache-line (but not the same memory address), with at least one of them for writing.
- Other parallel overheads should be watched like *memory contention*, *scheduling and synchronization costs*, which are very hard to take into account in complexity analysis [15, 18, 21].
- Nevertheless, on multicore processors, a good practical indication about what to expect in $W(n)/Q(n; Z, L)$, in addition to the more standard ration $T_1(n)/T_\infty(n)$.

Data reshaping

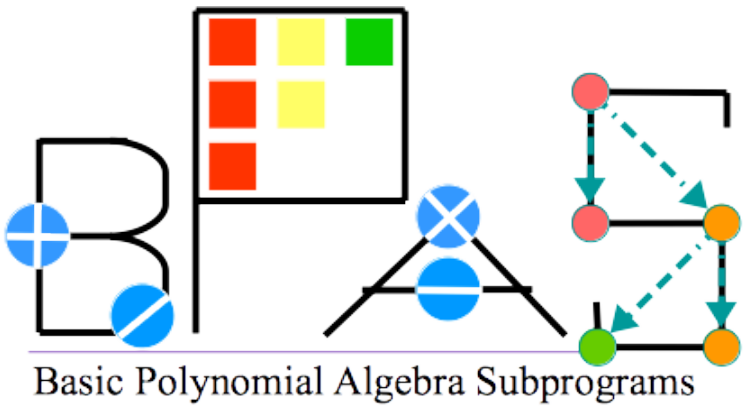
- Other performance degradation can come from *for-loop overheads*.
- If a loop has a few iterations, then overheads due to *branch misprediction* can have an impact, since a misprediction delay can be between 10 and 35 clock cycles [8].
- Trying to avoid those issues with for-loop nests has several advantages, including reducing overheads due to loop counter manipulation.

- In the context of dense multivariate polynomials over finite fields, this idea was studied in [22, 24] for *multi-threaded multi-dimensional FFTs (and TFTs)* and their application to polynomial multiplication.
- The authors systematically reduce multivariate polynomials to **balanced bivariate polynomials**. *Balanced* here means that partial degrees are equal or as close as possible.
- A theoretical study, supported by extensive experimentation, shows that this approach *minimizes cache misses* and *maximizes parallelism.*

Multi-measure models for many-core machines

- Two types of models:
  - ↳ predictor models (TMM, MCM) which can be used to design the implementation of a many-core programs; these models estimate running times from DAG characteristics.
  - ↳ profiler models (MWP-CWP) which can be used to understand performance issues; these models estimate running times from performance counters.
- Of course, those models (and their usage described above) have limitations that users should be aware of.
- Nevertheless, they can help understand experimental observations.

Thank You!



http://www.bpaslib.org/

References

[1] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, D. Talaashrafi, L. Wang, N. Xie, and Y. Xie. *Basic Polynomial Algebra Subprograms (BPAS)*. www.bpaslib.org. 2021.

[2] M. Asadi, A. Brandt, R. H. C. Moir, M. M. Maza, and Y. Xie. "On the parallelization of triangular decompositions". In: *International Symposium on Symbolic and Algebraic Computation (ISSAC '20), Kalamata, Greece, July 20-23, 2020*. ACM, 2020, pp. 22–29.

[3] M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Algorithms and Data Structures for Sparse Polynomial Arithmetic". In: *Mathematics* 7.5 (2019), p. 441.

[4] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. "Parallelization of Triangular Decompositions: Techniques and Implementation". In: *J. Symb. Comput.* (2021). *(to appear).*

[5] A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Employing C++ Templates in the Design of a Computer Algebra Library". In: *Mathematical Software - ICMS 2020, Braunschweig, Germany, July 13-16, 2020*. Vol. 12097. LNCS. Springer, 2020, pp. 342–352.

[6] C. Chen, M. Moreno Maza, and Y. Xie. "Cache Complexity and Multicore Implementation for Univariate Real Root Isolation". In: *J. of Physics: Conference Series* 341 (2011).

[7] M. F. I. Chowdhury, M. M. Maza, W. Pan, and É. Schost. "Complexity and Performance Results for non FFT-based Univariate Polynomial Multiplication.". In: *Proceedings of Advances in mathematical and computational methods: addressing modern of science, technology, and society, AIP conference proceedings*. volume 1368. 2011, pp. 259–262.

[8] S. Eyerman, J. E. Smith, and L. Eeckhout. "Characterizing the branch misprediction penalty". In: *2006 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2006, March 19-21, 2006, Austin, Texas, USA, Proceedings*. IEEE Computer Society, 2006, pp. 48–58.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-Oblivious Algorithms". In: *ACM Transactions on Algorithms* 8.1 (2012).

[10] M. Frigo and V. Strumpen. "The cache complexity of multithreaded cache oblivious algorithms". In: *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2006*. ACM, 2006, pp. 271–280.

[11] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra (3. ed.)* Cambridge University Press, 2013. ISBN: 978-1-107-03903-2.

[12] P. B. Gibbons. "A more practical PRAM model". In: *Proc. of SPAA*. 1989, pp. 158–168.

[13] P. B. Gibbons, Y. Matias, and V. Ramachandran. "The Queue-Read Queue-write asynchronous PRAM model". In: *Theoretical Computer Science* 196.1 (1998), pp. 3–29.

[14] P. B. Gibbons, Y. Matias, and V. Ramachandran. "The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms". In: *SIAM J. on Comput.* 28.2 (1998), pp. 733–769.

[15] S. A. Haque, M. M. Maza, and N. Xie. "A Many-Core Machine Model for Designing Algorithms with Minimum Parallelism Overheads". In: *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*. Vol. 27. Advances in Parallel Computing. IOS Press, 2015, pp. 35–44.

[16] S. A. Haque, M. M. Maza, and N. Xie. "A Many-core Machine Model for Designing Algorithms with Minimum Parallelism Overheads". In: *CoRR* abs/1402.0264 (2014). arXiv: 1402.0264. URL: http://arxiv.org/abs/1402.0264.

[17] J.-W. Hong and H. T. Kung. "I/O Complexity: The Red-Blue Pebble Game". In: *STOC*. ACM, 1981, pp. 326–333.

[18] S. Hong and H. Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness". In: *Proc. of ISCA*. 2009, pp. 152–163.

[19] L. Ma, K. Agrawal, and R. D. Chamberlain. "A Memory Access Model for Highly-threaded Many-core Architectures". In: *Proc. of ICPADS*. 2012, pp. 339–347.

[20] L. Ma and R. D. Chamberlain. "A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines". In: *Proc. of ASAP*. IEEE Computer Society, 2012, pp. 24–31.

[21] L. Ma, K. Agrawal, and R. D. Chamberlain. "A memory access model for highly-threaded many-core architectures". In: *Future Generation Comp. Syst.* 30 (2014), pp. 202–215.

[22] M. M. Maza and Y. Xie. "FFT-Based Dense Polynomial Arithmetic on Multi-cores". In: *High Performance Computing Systems and Applications, 23rd International Symposium, HPCS 2009*. Vol. 5976. LNCS. Springer, 2009, pp. 378–399.

[23] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[24] M. Moreno Maza and Y. Xie. "Balanced Dense Polynomial Multiplication on Multi-Cores". In: *Int. J. Found. Comput. Sci.* 22.5 (2011), pp. 1035–1055.

[25] M. Moreno Maza and H. Yuan. "Balanced Dense Multivariate Multiplcation: The General Case". In: *SYNASC*. 2022.

[26] N. Satish, M. Harris, and M. Garland. "Designing efficient sorting algorithms for manycore GPUs". In: *Proc. of IPDPS 2009*. IEEE, 2009, pp. 1–10.

[27] J. E. Savage. *Models of computation - exploring the power of computing*. Addison-Wesley, 1998. ISBN: 978-0-201-89539-1.