Improving the Space-Time Efficiency of Matrix Multiplication Algorithms

Yuan Tang

yuantang@fudan.edu.cn

School of Computer Science, Fudan University

Motivation

- Matrix Multiplication and fast algorithms (e.g. Strassen) is a fundamental computation and building block in algorithm design.
- Classic Processor-Aware (PA) approaches may not utilize all processor effectively unless the processor number matches well the structure of algorithm. E.g.
  - The Communication-Avoiding Parallel Strassen (CAPS) by Ballard et al. [5] requires p to be an exact power of 7. Lipshitz et al. [32] improved it to a multiple of 7 with no large prime factors, i.e. $p = m * 7^x$, where $1 \le m < 7$ and $1 \le x$ are integers by a hybrid of Strassen and classic MM.
  - Communication–Avoiding parallel Recursive rectangular Matrix Multiplication (CARMA) by Demmel et al. [20] assumes p is an exact power of 2, or any of p's prime factor can be bounded by a small constant.
- Classic Processor-Oblivious (PO) and Cache-Oblivious MM algorithms achieve optimality either in time or space, but not both. E.g.
  - CO2: $O(n)$ depth, $O(n^3/(B\sqrt{M}) + p\, n\, M/B)$ w.h.p.
  - CO3: $O(\log n)$ depth, $O(n^3/B + p \log n\, M/B)$ w.h.p.

CO3(C, A, B)

// $C \leftarrow A \times B$
if (sizeof(C) $\leq$ BASE_SIZE)
    BASE-KERNEL(C, A, B)
    return
$D \leftarrow$ alloc(sizeof(C))
// Run all 8 sub-MMs concurrently
CO3(C_{00}, A_{00}, B_{00}) || CO3(C_{01}, A_{00}, B_{01})
|| CO3(C_{10}, A_{10}, B_{00}) || CO3(C_{11}, A_{10}, B_{01})
|| CO3(D_{00}, A_{01}, B_{10}) || CO3(D_{01}, A_{01}, B_{11})
|| CO3(D_{10}, A_{11}, B_{10}) || CO3(D_{11}, A_{11}, B_{11})
; // **sync**
// Merge matrix D into C by addition
madd(C, D)
free (D)
return

(a) Recursive MM algorithm with $O(n^3)$ space

CO2(C, A, B)

// $C \leftarrow A \times B$
if (sizeof(C) $\leq$ BASE_SIZE)
    BASE-KERNEL(C, A, B)
    return
// Run the first 4 sub-MMs concurrently
CO2(C_{00}, A_{00}, B_{00}) || CO2(C_{01}, A_{00}, B_{01})
|| CO2(C_{10}, A_{10}, B_{00}) || CO2(C_{11}, A_{10}, B_{01})
; // **sync**
// Run the next 4 sub-MMs concurrently
CO2(C_{00}, A_{01}, B_{10}) || CO2(C_{01}, A_{01}, B_{11})
|| CO2(C_{10}, A_{11}, B_{10}) || CO2(C_{11}, A_{11}, B_{11})
; // **sync**
return

(b) Recursive MM algorithm with $O(n^2)$ space

Figure 2: Recursive MM algorithms. "||" and ";" are linguistic constructs of the Nested Parallel model (Sect. 2).

Our Contributions

Algo.	Work (T_1)	Time (T_∞)	Space (S_p)	Sequential Cache (Q_1)
CO2 [19]	$O(n^3)$	$O(n)$	$O(n^2)$	$O(n^3/(B\sqrt{M}) + n^2/B)$
CO3 [19]	$O(n^3)$	$O(\log n)$	$O(n^3)$	$O(n^3/B)$
TAR-MM	$O(n^3)$	$O(n)$	$O(n^2 + pb^2)$	$O(n^3/(B\sqrt{M}) + n^2/B)$
SAR-MM	$O(n^3)$	$O(\log n)$	$O(p^{1/3}n^2)$	$O(n^3/(B\sqrt{M}) + n^2/B)$
STAR-MM	$O(n^3)$	$O(\sqrt{p}\log n)$	$O(n^2)$	$O(n^3/(B\sqrt{M}) + n^2/B)$
Straightforward Strassen	$O(n^{\omega_0})$	$O(\log n)$	$O(n^{\omega_0})$	$O(n^{\omega_0}/B)$
SAR-Strassen	$O(n^{\omega_0})$	$O(\log n)$	$O(pn^2)$	$O(n^{\omega_0}/(BM^{(1/2)\omega_0-1}) + n^2/B)$
STAR-MM-Strassen	$O(p^{0.09}n^{\omega_0})$	$O(p^{1/2}\log n)$	$O(n^2)$	$O(p^{0.09}n^{\omega_0}/(BM^{(1/2)\omega_0-1}) + p^{1/2}n^2/B)$
STAR-Strassen	$O(n^{\omega_0})$	$O(\log n)$	$O(p^{(1/2)\omega_0}n^2)$	$O(n^{\omega_0}/(BM^{(1/2)\omega_0-1}) + p^{(1/2)\omega_0-1}n^2/B)$

Figure 1: Main results of this paper, with comparisons to typical prior works. CO2 stands for the MM (Matrix Multiplication) algorithm with $O(n^2)$ space (Fig. 2b); CO3 stands for the MM algorithm with $O(n^3)$ space (Fig. 2a); p denotes processor count, b is the base-case dimension. $\omega_0 = \log_2 7$

Cost Models and Programming Model

- Parallel Performance Model:
  - Work-span model, aka work-time model
  - Views a parallel computation as a DAG. Each vertex stands for a computation and each edge some control or data dependency. Each arithmetic op is counted uniformly as an $O(1)$ op.
  - Only calculates total work (T_1) and critical-path length (T_∞)
  - Parallel running time : $T_p = O(T_1 / p + T_\infty)$ w.h.p. [3]
- Memory Model
  - Sequential cache complexity (Q_1) in the ideal cache model [22]
  - Parallel cache complexity : $Q_p = Q_1 + O(p T_\infty M / B)$ w.h.p. under RWS scheduler [1, 37]
- Programming Model:
  - Nested Parallel Model, aka Fork-Join Model
  - Parallel: a || b
  - Serial: a ; b

Time Adaptive and Reductive (TAR) Algorithm

- Problems of CO_2:
  - It imposes more control dependency than necessary data dependency to keep the algorithm correct. E.g. all-to-all sync between the 2 parallel steps (8 sub-MMs) of CO_2.
  - The all-to-all sync separating the 2 parallel steps actually serializes all n muls targeting the same cell. However, muls by itself are independent of each other and should be parallelized, serialization only makes sense for the later adds.
- Our improvements:
  - TAR to remove unnecessary control dependency from a critical path, parallelizes all muls and serializes only adds.

```
TAR-MM(C, A, B)
// C ← A × B
if (sizeof(C) ≤ BASE_SIZE)
    // Request space from the program-managed memory pool
    D ← GET-STORAGE(sizeof(C))
    BASE-KERNEL(D, A, B)
    // Write the intermediate results in D to C atomically
    ATOMIC-MADD(C, D)
    // Return storage to the memory pool
    free(D)
    return
// Run all 8 sub-MMs concurrently
TAR-MM(C00, A00, B00) || TAR-MM(C01, A00, B01)
|| TAR-MM(C10, A10, B00) || TAR-MM(C11, A10, B01)
|| TAR-MM(C00, A01, B10) || TAR-MM(C01, A01, B11)
|| TAR-MM(C10, A11, B10) || TAR-MM(C11, A11, B11)
return
```

Figure 3: TAR-MM algorithm

Time Adaptive and Reductive (TAR) Algorithm

- Memory Allocator:
  - Space for base-case computation gets reused on each processor
  - A task on base-case computation cannot block or be preempted.
- Theorem 1:
  - The TAR-MM algorithm computes classic square MM of dimension n on a semiring in O(n)time, with O(n^2+p b^2) space, and optimal O(n^3/(B√M) + n^2/B) cache misses, where b denotes the dimension of base case. If assuming b is some small constant, the space bound reduces to O(n^2+p).

```
TAR-MM(C, A, B)
// C ← A × B
if (sizeof(C) ≤ BASE_SIZE)
    // Request space from the program-managed memory pool
    D ← GET-STORAGE(sizeof(C))
    BASE-KERNEL(D, A, B)
    // Write the intermediate results in D to C atomically
    ATOMIC-MADD(C, D)
    // Return storage to the memory pool
    free(D)
    return
// Run all 8 sub-MMs concurrently
TAR-MM(C00, A00, B00) || TAR-MM(C01, A00, B01)
|| TAR-MM(C10, A10, B00) || TAR-MM(C11, A10, B01)
|| TAR-MM(C00, A01, B10) || TAR-MM(C01, A01, B11)
|| TAR-MM(C10, A11, B10) || TAR-MM(C11, A11, B11)
return
```

Figure 3: TAR-MM algorithm

Space Adaptive and Reductive (SAR) Algorithm

- Problems of CO3:
  - It allocates space irrespective of the availability of processors, i.e. it is designed for an infinite number of processors, or proportional to T_1 / T_\infty.
- Our improvements:
  - Generalization of ``busy-leaves” property: each depth of each processor can have only one copy of memory and will be reused across function calls.
  - Lazy Allocation: allocate space iff it runs simultaneously on a different processor from the sub-MM updating the same output region.

```
SAR-MM(C, A, B, d)
// Computes SAR-MM at recursion level d
// Run all 8 sub-MMs concurrently
HLP(C00, A00, B00, d + 1) || HLP(C01, A00, B01, d + 1)
|| HLP(C10, A10, B00, d + 1) || HLP(C11, A10, B01, d + 1)
|| HLP(C00, A01, B10, d + 1) || HLP(C01, A01, B11, d + 1)
|| HLP(C10, A11, B10, d + 1) || HLP(C11, A11, B11, d + 1)
return
```

Figure 5: SAR-MM algorithm

```
HLP(Parent, A, B, d)
if (parent.trylock())
    // work right on parent's storage
    D ← parent
else
    // request space for depth d
    D ← GET-STORAGE(sizeof(n/2^d))
if (sizeof(n/2^d) ≤ BASE_SIZE)
    BASE-KERNEL(D, A, B)
else
    SAR-MM(D, A, B, d)
if (D ≠ parent)
    // Update D to parent atomically
    ATOMIC-MADD(parent, D)
    // Return storage to the memory pool
    free(D)
else
    parent.unlock()
return
```

Figure 4: The helper function request temporary storage from the program-managed memory pool *iff* parent's storage is occupied. If computation is on a local temporary storage, the helper function will write back results to parent by atomic addition.

Space Adaptive and Reductive (SAR) Algorithm

- Theorem 3. The SAR-MM algorithm computes general MM of dimension n on a semi-ring in optimal O(logn) time, O(p^{1/3}n^2) space, and optimal O(n^3/B √ M + n^2/B) cache complexities, assuming p=o(n).

```
HLP(Parent, A, B, d)
if (parent.trylock())
    // work right on parent's storage
    D ← parent
else
    // request space for depth d
    D ← GET-STORAGE(sizeof(n/2^d))
if (sizeof(n/2^d) ≤ BASE_SIZE)
    BASE-KERNEL(D, A, B)
else
    SAR-MM(D, A, B, d)
if (D ≠ parent)
    // Update D to parent atomically
    ATOMIC-MADD(parent, D)
    // Return storage to the memory pool
    free(D)
else
    parent.unlock()
return
```

Figure 4: The helper function request temporary storage from the program-managed memory pool *iff* parent's storage is occupied. If computation is on a local temporary storage, the helper function will write back results to parent by atomic addition.

```
SAR-MM(C, A, B, d)
// Computes SAR-MM at recursion level d
// Run all 8 sub-MMs concurrently
HLP(C00, A00, B00, d + 1) || HLP(C01, A00, B01, d + 1)
|| HLP(C10, A10, B00, d + 1) || HLP(C11, A10, B01, d + 1)
|| HLP(C00, A01, B10, d + 1) || HLP(C01, A01, B11, d + 1)
|| HLP(C10, A11, B10, d + 1) || HLP(C11, A11, B11, d + 1)
return
```

Figure 5: SAR-MM algorithm

Space-Time Adaptive and Reductive (STAR) Algorithm

- The TAR algorithm remove muls from a critical path without using much more space, while the SAR algorithm reduces the space complexity without increasing the time complexity
- STAR = TAR + SAR
- Theorem 4. The STAR-MM algorithm computes the general MM of dimension n on a semi-ring in $O(\sqrt{p}\log n)$ time, optimal $O(n^2)$ space, and optimal $O(n^3/(B\sqrt{M})+n^2/B)$ cache bounds, assuming $p=o(n^2/\log^2 n)$
- Theorem 7.The STAR-MM-Strassen algorithm has an $O(p^{1/2} \log n)$ time, $O(p^{0.09} n^{\omega_0})$work, optimal $O(n^2)$ space, and $O(p^{0.09}\cdot n^{\omega_0}/(BM^{(1/2)\omega_0-1}) + p^{1/2}\cdot n^2/B)$ sequential cache complexities, where $\omega_0=\log_2 7$
- Theorem 8. The STAR-Strassen algorithm has an optimal $O(n^{\omega_0})$ work, optimal $O(\log n)$ time, near-optimal $O(n^{\omega_0}/(BM^{(1/2)\omega_0-1}) + p^{(1/2)\omega_0-1} n^2 / B)$ cache and an $O(p^{(1/2)\omega_0} n^2)$ space complexities, where $\omega_0=\log_2 7$.

Experiments

Table 1: Experiemental Machine

Name	24-core machine
CPU type Clock Freq # sockets # cores / socket	Intel Xeon E5-2670 v3 2.30 GHz 2 12
Dual Precision FLOPs / cycle Hyper-Threading	16 disabled
OS Compiler	CentOS 7 x86_64 ICC 19.0.3
L1 dcache / core L2 cache / core L3 cache (shared) memory	32 KB 256 KB 30 MB 132 GB

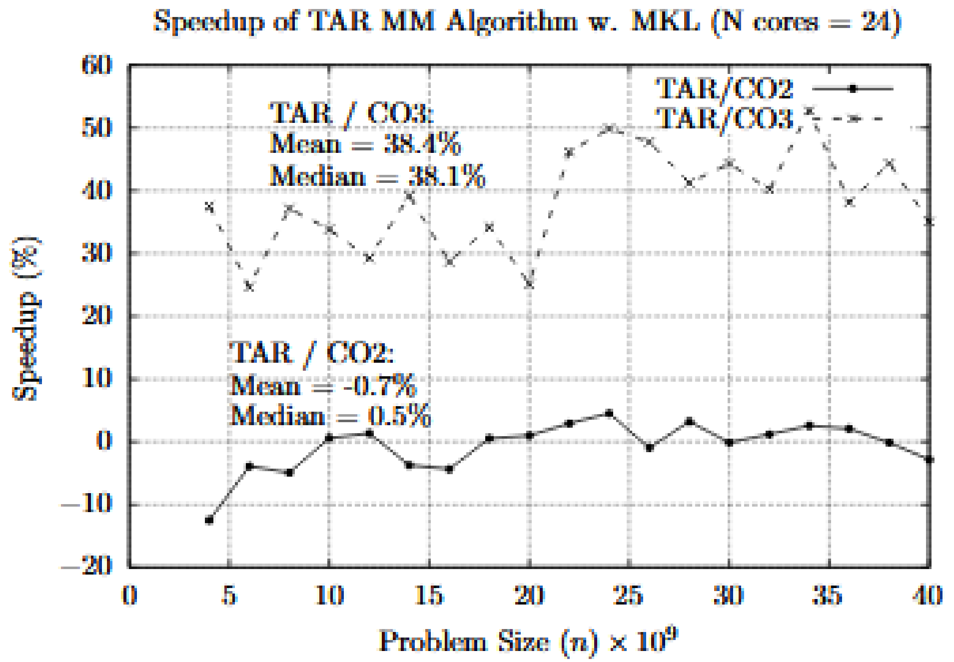

Figure 6: TAR-MM's speedup over CO2 and CO3 with MKL kernel

with MKL kernel			
Mean/Median Spdp (%)	TAR	SAR	STAR
CO2	−0.7/0.5	−2.0/−1.0	−2.6/−1.8
CO3	38.4/38.1	36.6/36.5	35.8/34.0
with manual kernel			
Mean/Median Spdp (%)	TAR	SAR	STAR
CO2	11.8/9.2	9.3/6.8	10.5/8.0
CO3	1.6/1.5	−0.6/−0.5	0.5/0.5