A predictable SIMD library for GEMM routines

Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, Victor Jegu, Claire Pagetti

RTAS, May 14th 2024

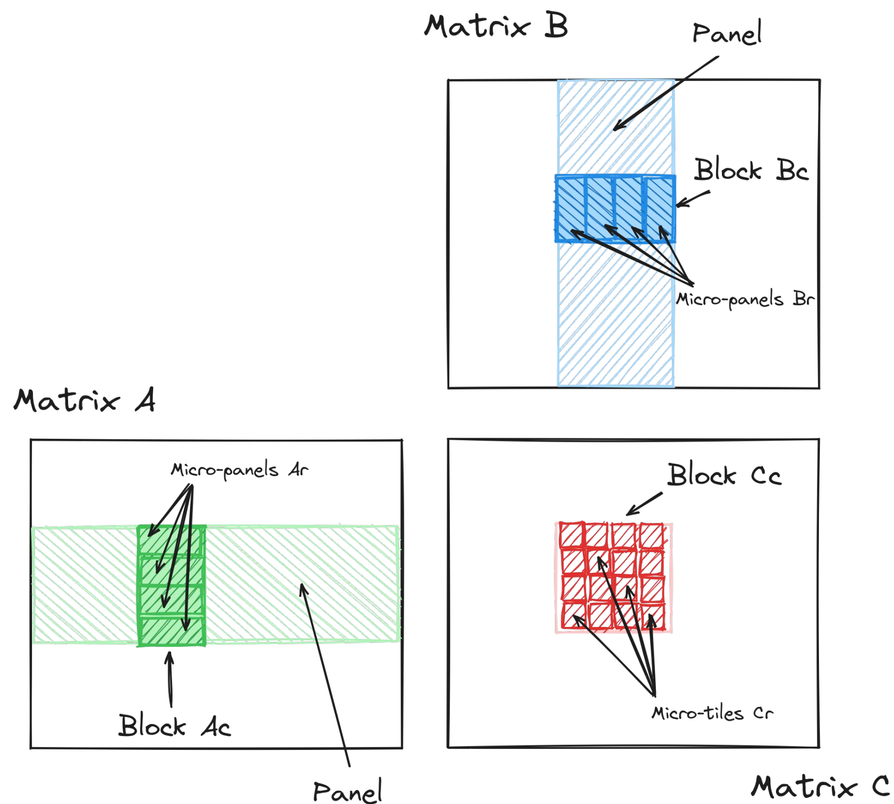
Introduction

- General Matrix Multiplication (GEMM) used for scientific simulations and ML-based applications
- Heavily optimized in HPC domain
  - Blocked matrix multiplication (improved cache usage)
  - SIMD extensions (vector instructions)
- Safety-critical real-time systems ⇒ certification standards
  - Traceability
  - Timing-predictability
- Objective: optimized and certifiable GEMM library
  - Analyzable and traceable code
    - No dynamic memory management
    - No compiler optimization
  - Choice of blocking parameters to improve predictability

Blocked GEMM

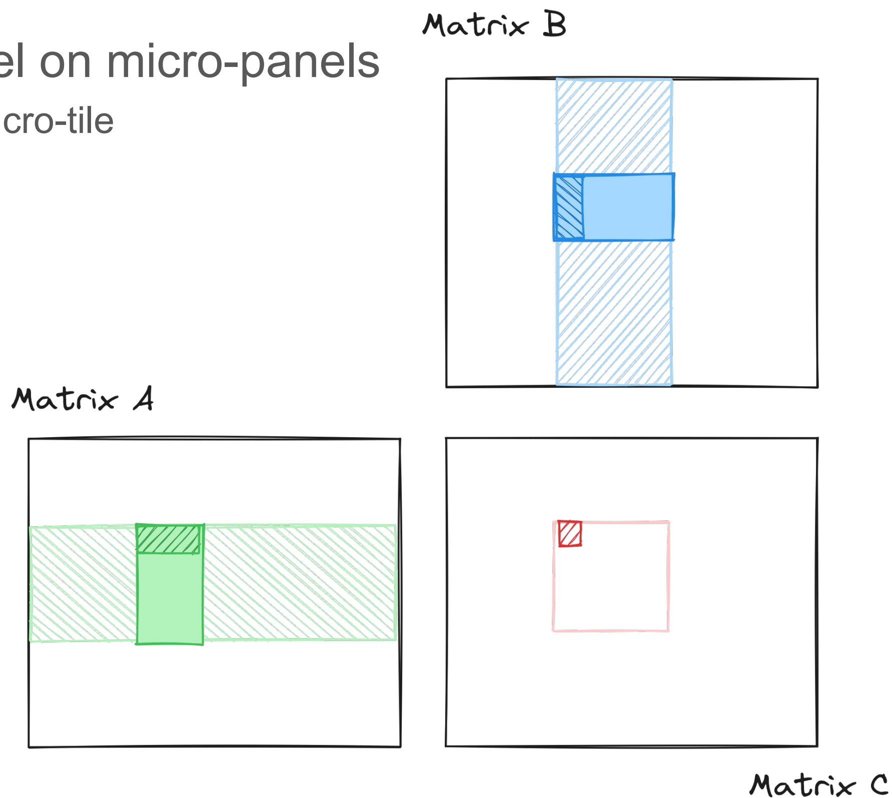
- Goal : compute C = A . B + C
- Improve spatial and temporal locality in caches
  - Divide A, B and C in blocks (submatrices) that fit in the various levels of cache
- Use vector instructions
- 3 routines:
  - Macro-kernel (shaping the blocks, driving the algorithm)
  - Micro-kernel (computing partial products)
  - Packing (copying and reordering elements in the blocks)

Blocked GEMM (macro-kernel)



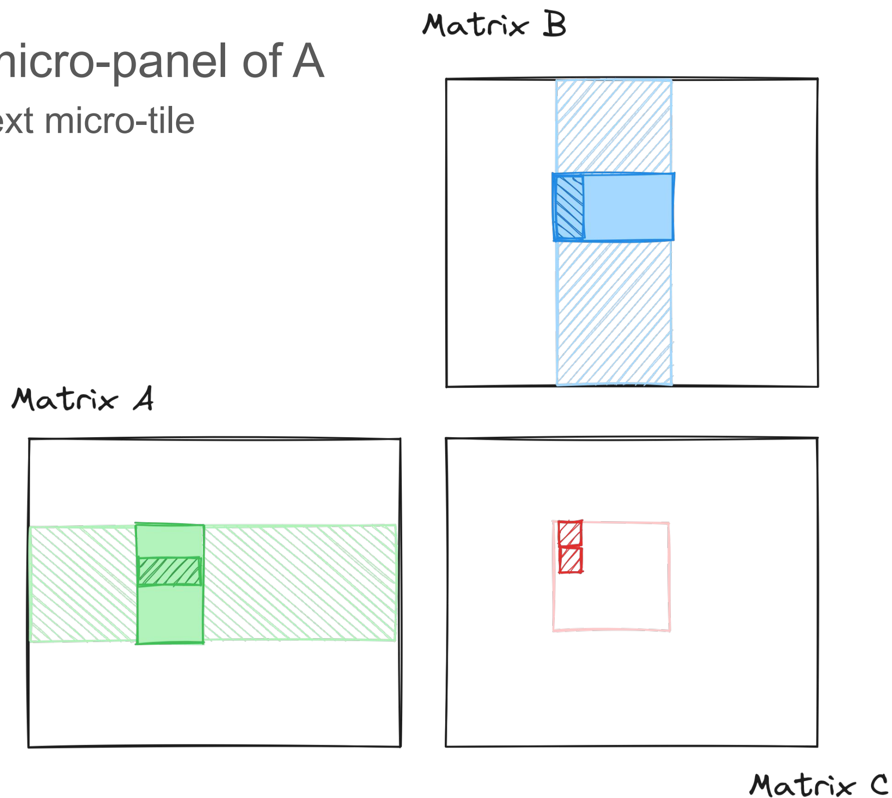
Blocked GEMM (macro-kernel)

- Call micro-kernel on micro-panels
  - → compute micro-tile



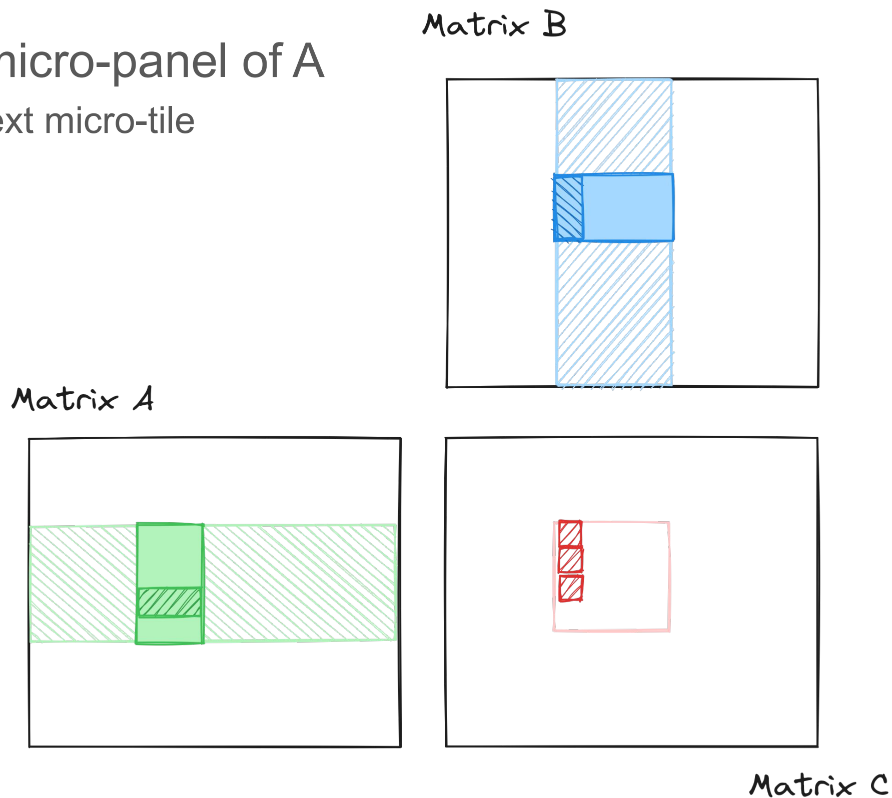
Blocked GEMM (macro-kernel)

- Switch to next micro-panel of A
  - → compute next micro-tile



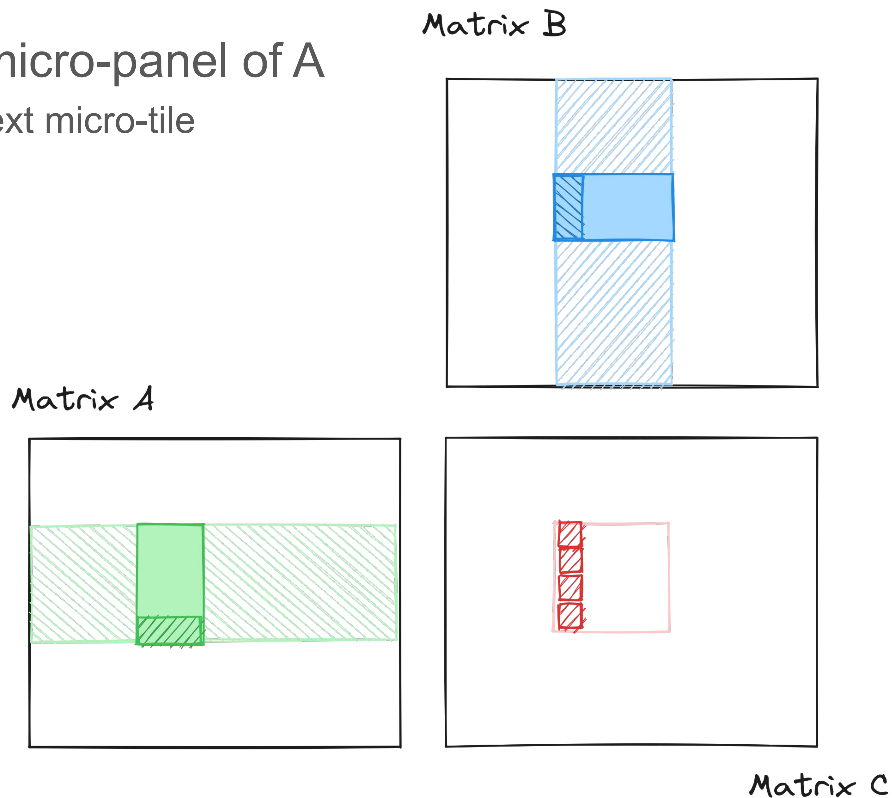
Blocked GEMM (macro-kernel)

- Switch to next micro-panel of A
  - → compute next micro-tile



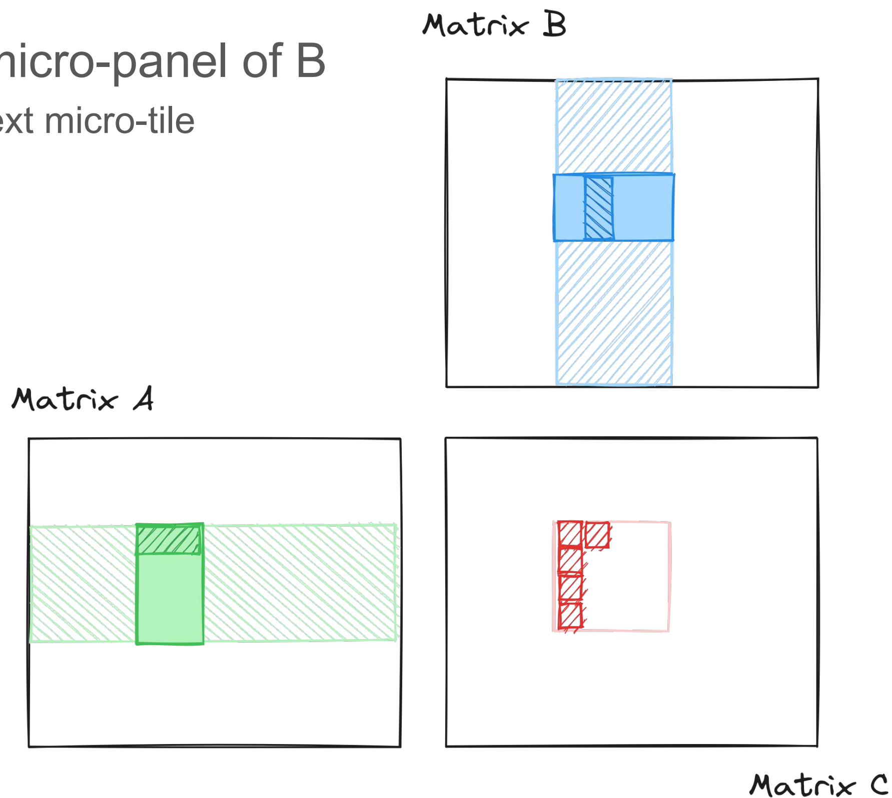
Blocked GEMM (macro-kernel)

- Switch to next micro-panel of A
  - → compute next micro-tile



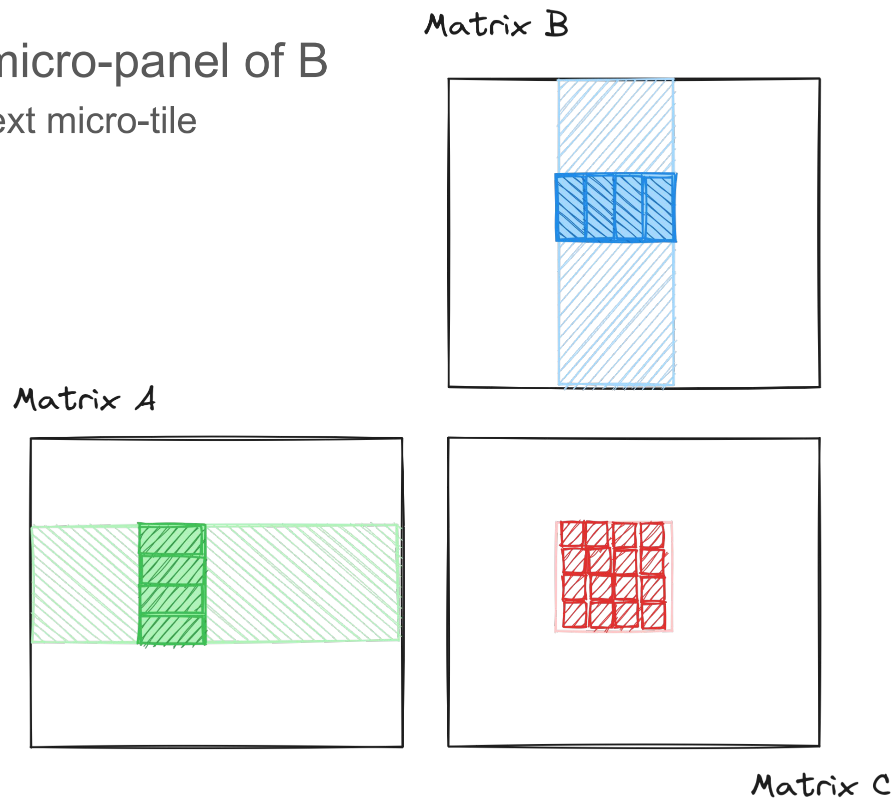
Blocked GEMM (macro-kernel)

- Switch to next micro-panel of B
  - → compute next micro-tile

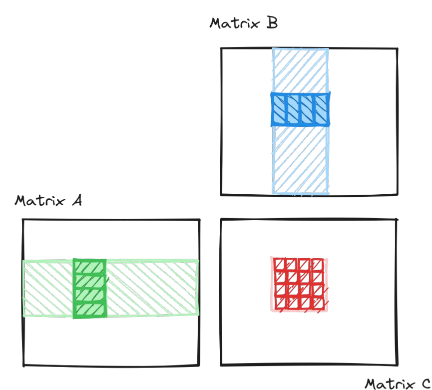


Blocked GEMM (macro-kernel)

- Switch to next micro-panel of B
  - → compute next micro-tile



Blocked GEMM (Micro-kernel)



Matrix B

Vector register

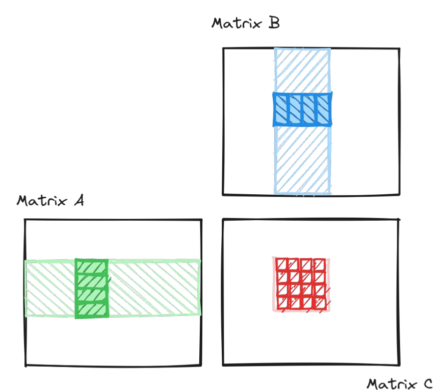
Vector
Multiply-
Accumulate

Matrix A

Vector register

Matrix C

Blocked GEMM (Micro-kernel)



Matrix B

Vector register

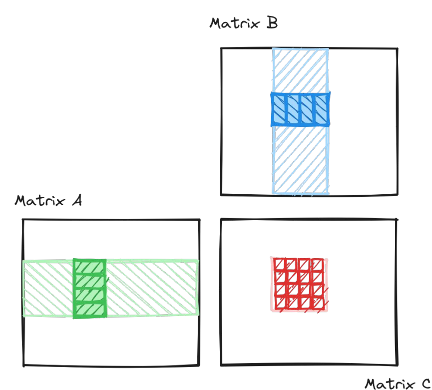
Vector Multiply-Accumulate

Matrix A

Vector register

Matrix C

Blocked GEMM (Micro-kernel)



- Loading elements in SIMD registers
  - Need to re-order them
  - → Packing

Matrix B

Vector register

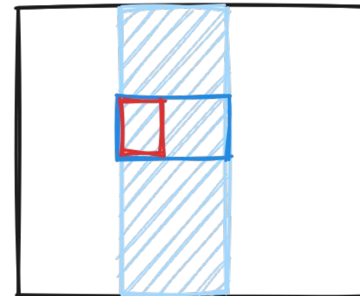
Vector Multiply-Accumulate

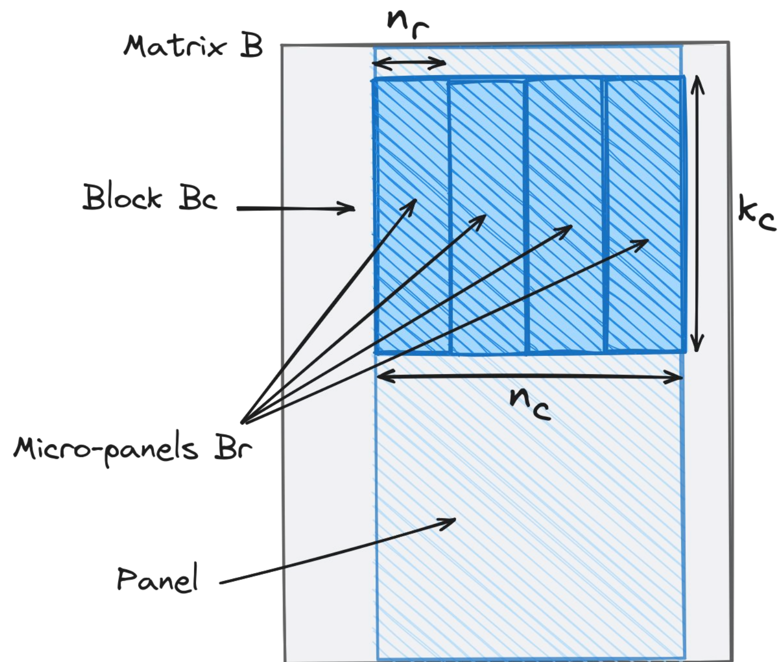
Matrix A

Vector register

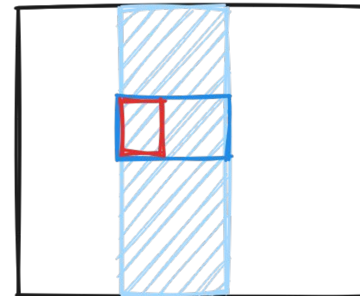
Matrix C

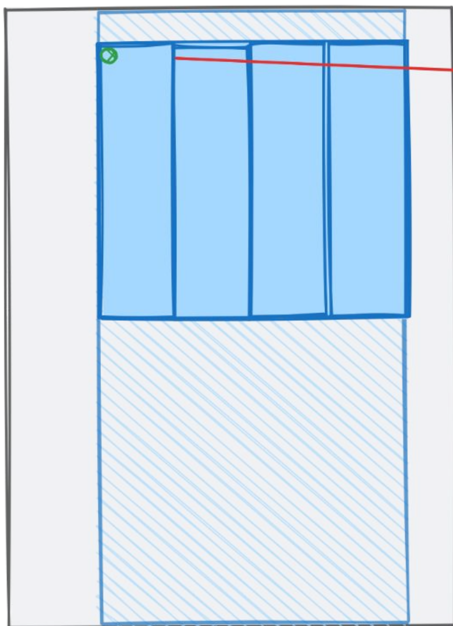
Blocked GEMM (Packing B)





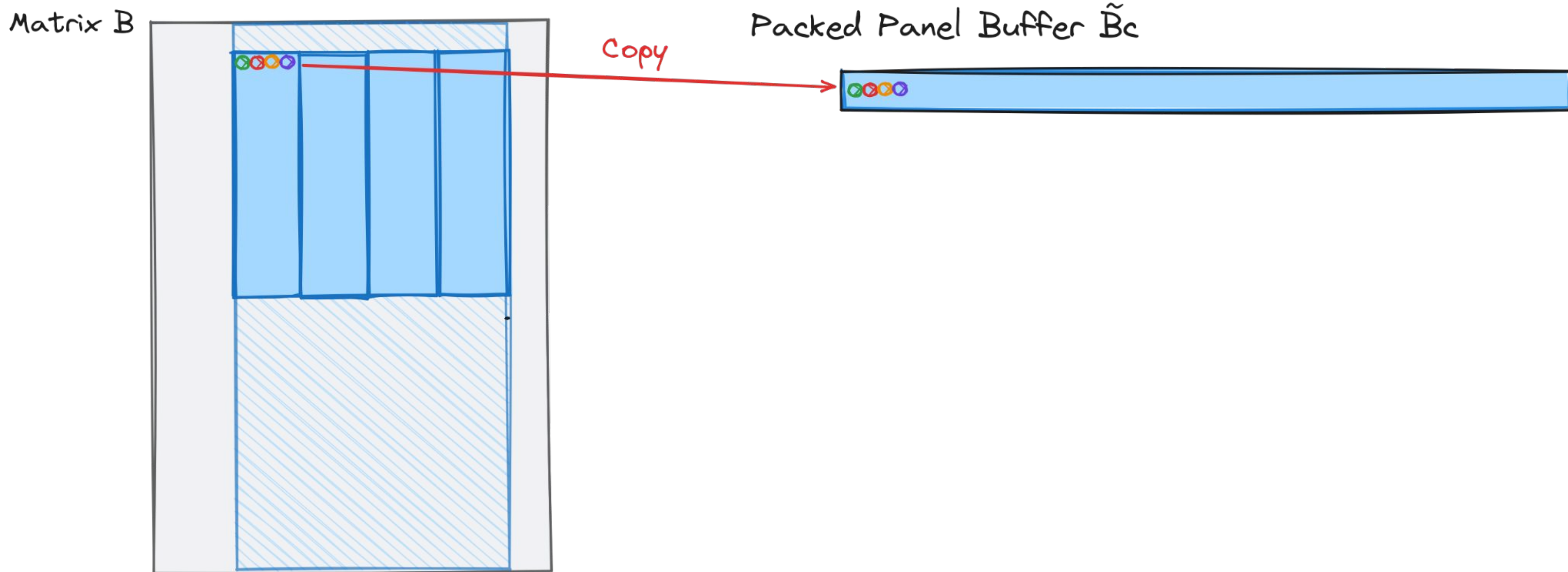
Blocked GEMM (Packing B)





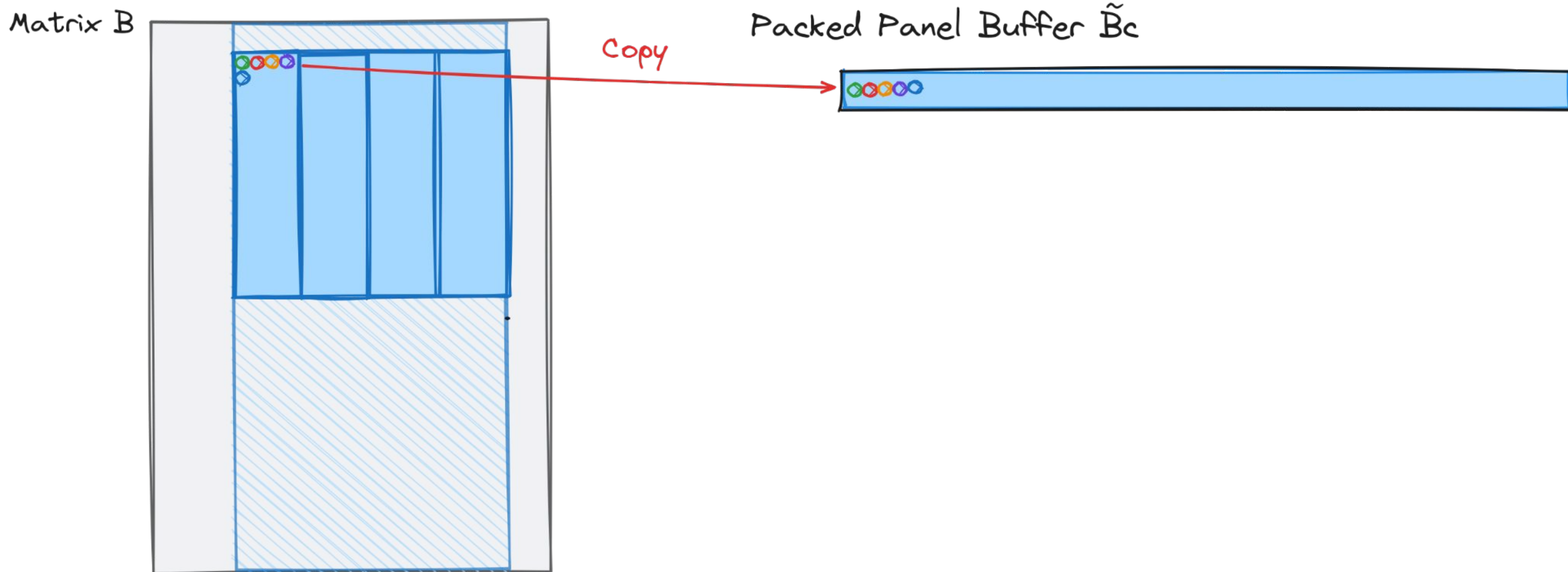
Blocked GEMM (Packing B)

Matrix B



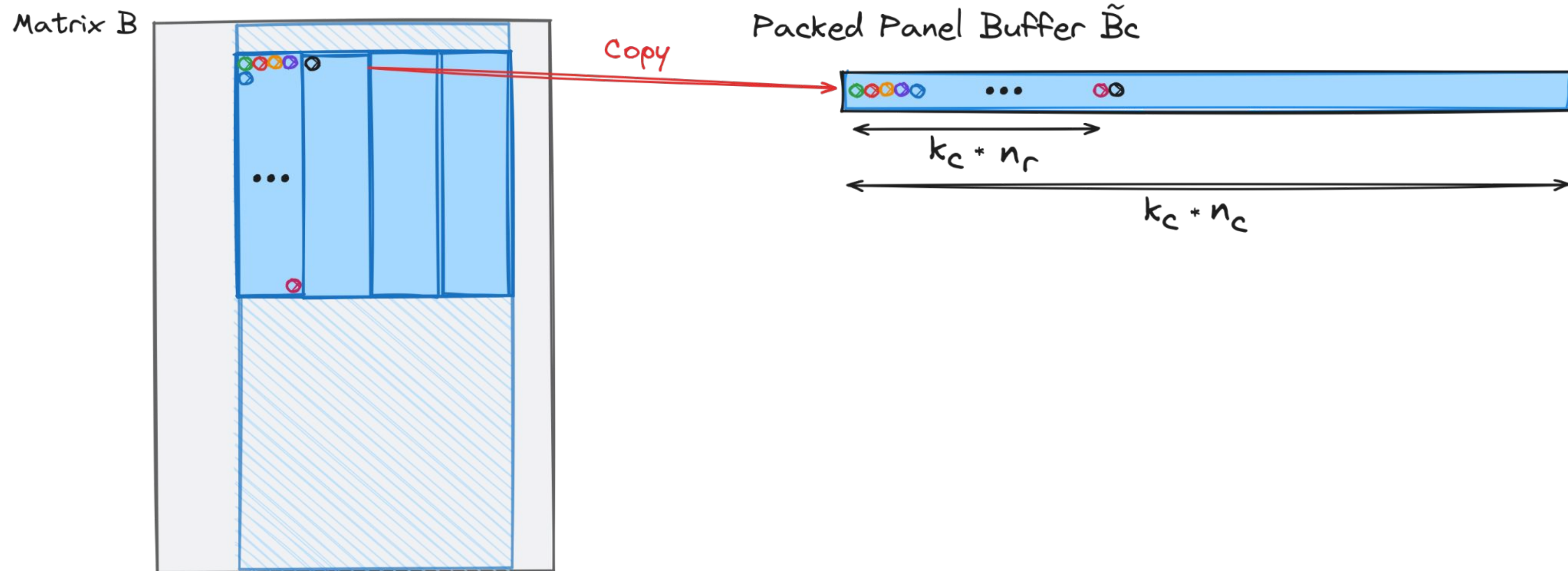
Blocked GEMM (Packing B)

Matrix B



Blocked GEMM (Packing B)

Matrix B



Contributions

- Contribution 1: Traceable Blocked GEMM library
  - No compiler optimization
  - Optimized NEON code, no intrinsics
- Contribution 2: Cache miss prediction
  - Implementation rules for predictability
  - Formulas for worst-case cache misses
- Target = Texas Keystone II
  - 1 ARM Cortex A15 (ARM v7)
  - NEON extension: 16 128-bits registers (**4 FP32 elements each**)
  - L1D: 32KB, 64 Bytes cache line, **2-ways associative (256 sets), LRU**
  - L2D: 4MB, 64 Bytes cache line, 16-ways associative (4096 sets), random

Library code optimization

- Force register usage in C code
  - Reduce pressure on the stack
- Select NEON instructions in assembly code blocks
  - With -O0, ARM NEON intrinsics translation by compiler is not efficient

TABLE I
MEASURED EXECUTION TIMES, IN CYCLES, ON AN ARM CORTEX-A15 WITH -O0 FLAG. DIFFERENCE W.R.T. *Base* IS GIVEN IN PARENTHESES.

Matrix configuration				**GEMM implementation**						
m	n	k		**Base**	**First optimized**		**Optimized with intrinsics**		**Optimized in assembly**	
272	272	272	mean	2 543 708 367	988 682 239	(-61.13%)	2 220 393 123	(-12.71%)	235 168 329	(-90.75%)
			min	2 543 704 067	988 676 103		2 220 386 698		235 153 669	
			max	2 543 711 485	988 692 654		2 220 398 214		235 181 181	
			min-max var.	2.92e−4%	1.67e−3%		5.19e−4%		1.17e−2%	
528	528	528	mean	6 920 673 125	2 883 613 750	(-58.33%)	3 312 934 607	(-52.13%)	1 566 589 440	(-77.36%)
			min	6 920 653 603	2 883 100 642		3 312 863 994		1 566 565 684	
			max	6 920 698 753	2 888 548 217		3 312 956 562		1 566 603 968	
			min-max var.	6.52e−4%	1.89e−1%		2.79e−3%		2.44e−3%	
192	736	528	mean	5 789 246 930	3 622 514 818	(-37.43%)	3 875 069 543	(-33.06%)	778 924 693	(-85.55%)
			min	5 789 237 191	3 622 514 793		3 875 069 520		778 923 061	
			max	5 789 266 393	3 622 514 892		3 875 069 565		778 926 162	
			min-max var.	5.02e−4%	2.73e−6%		1.16e−6%		3.98e−4%	
256	784	2,016	mean	2 120 705 709	579 964 569	(-72.65%)	857 163 412	(-59.58%)	134 562 545	(-93.65%)
			min	2 120 700 674	579 962 833		857 161 851		134 561 357	
			max	2 120 709 745	579 966 116		857 165 391		134 563 809	
			min-max var.	4.28e−4%	5.66e−4%		4.13e−4%		1.82e−3%	

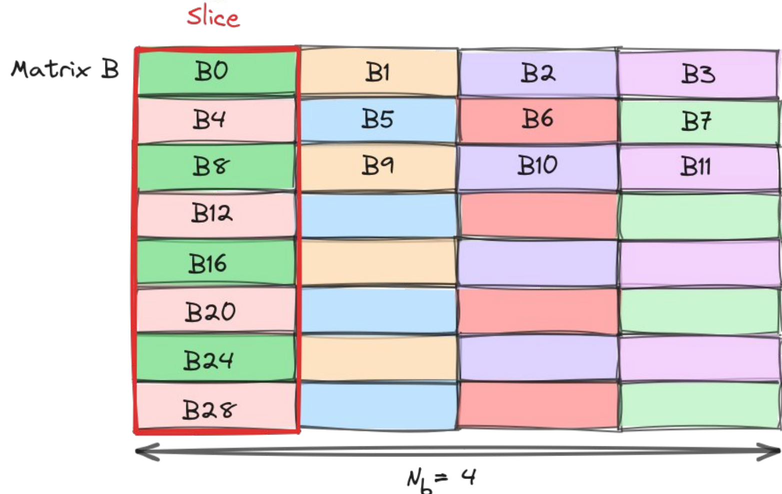
Improving predictability

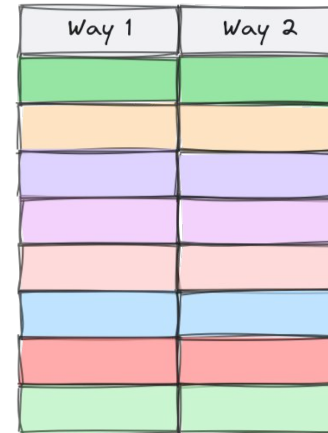
- Objective: statically bound memory accesses and cache misses
  - For each library routine (packing A, packing B, macro kernel)
  - Take advantage of regular code (no if-then-else)
- These values can then be used as input to static WCET analyser
  - Out of scope

Improving predictability

- Corollary (accesses to matrix slices = complete blocks in column)

Corollary 1. *If N_b is an odd integer, any slice of shape $s_{Li} \times X_{Li}$ of matrix M, whose first row is aligned to the start of a cache block, has each of its s_{Li} cache blocks mapped to a different cache set.*

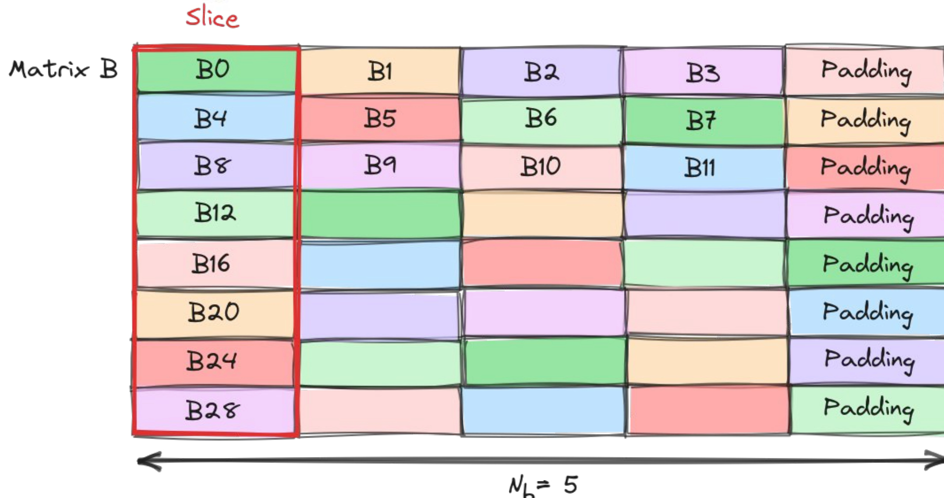


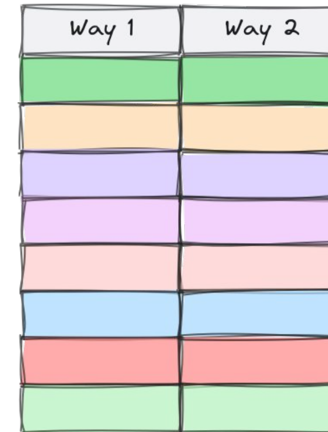


Improving predictability

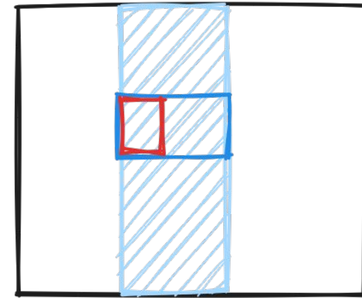
- Corollary (accesses to matrix slices = complete blocks in column)

Corollary 1. *If N_b is an odd integer, any slice of shape $s_{Li} \times X_{Li}$ of matrix M, whose first row is aligned to the start of a cache block, has each of its s_{Li} cache blocks mapped to a different cache set.*

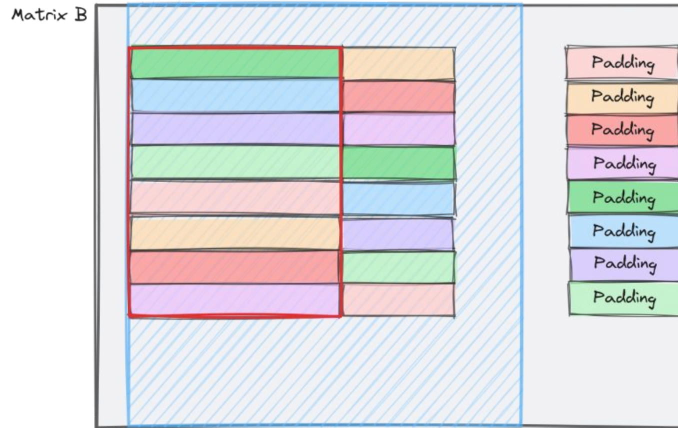


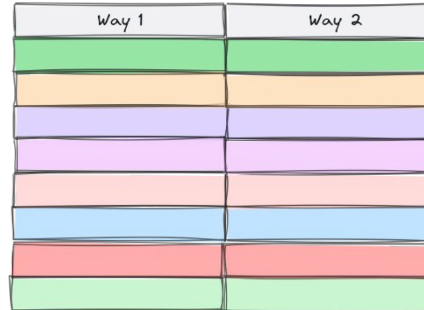


Improving predictability



- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose slice height equal to number of cache sets

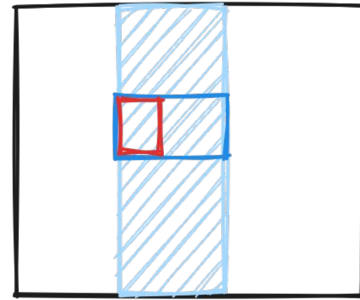




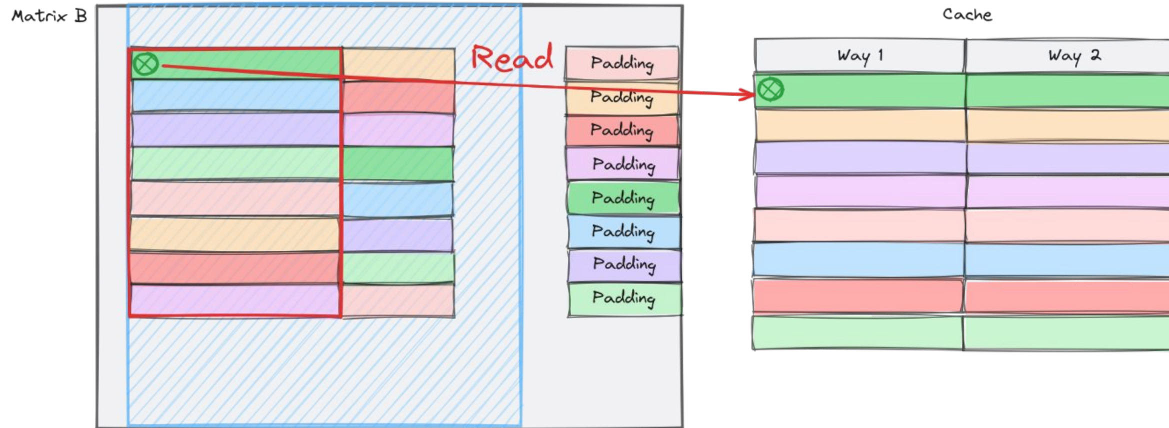
Buffer $\tilde{B}_c$

...

Improving predictability

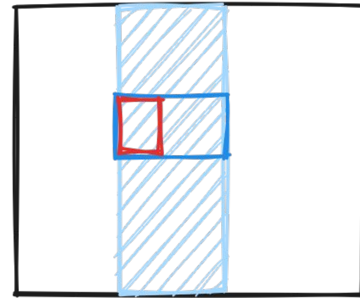


- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose slice height equal to number of cache sets

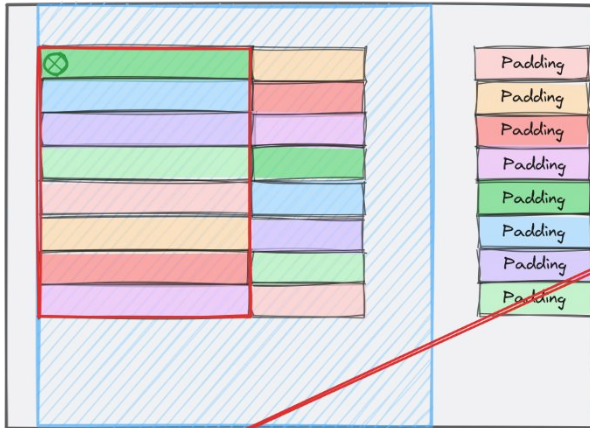


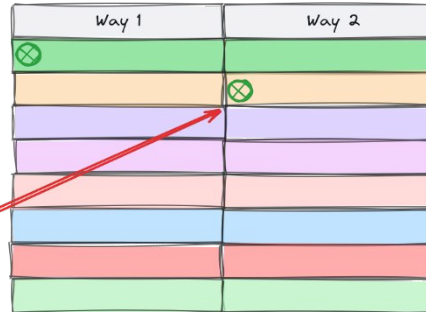
1 miss

Improving predictability



- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose slice height equal to number of cache sets

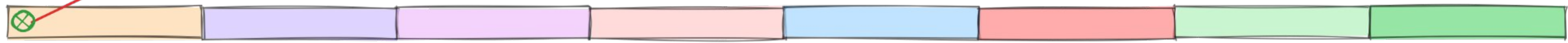




1 miss + 1 miss

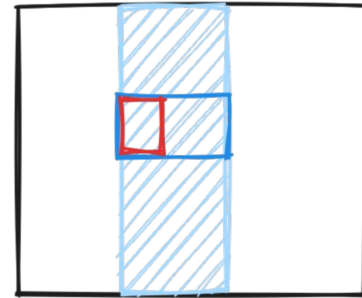
Buffer $\tilde{B}_c$

Write

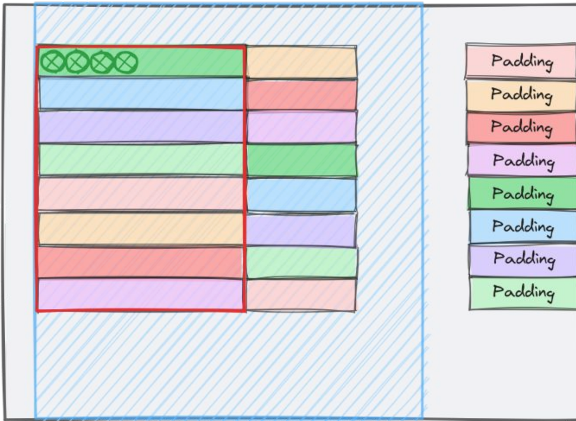


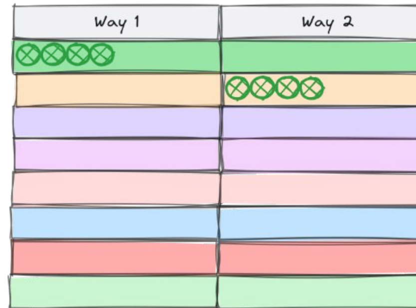
...

Improving predictability

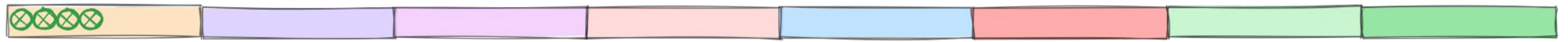


- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose slice height equal to number of cache sets

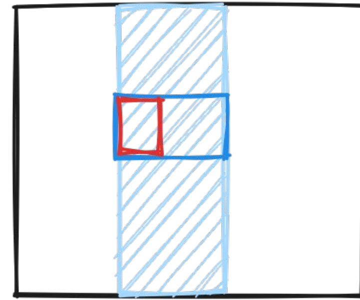




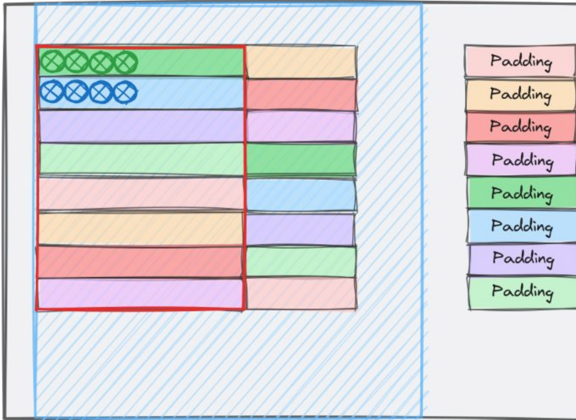
1 miss + 1 miss

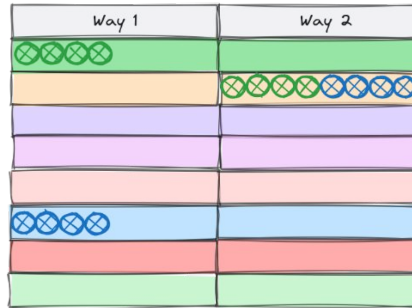


Improving predictability

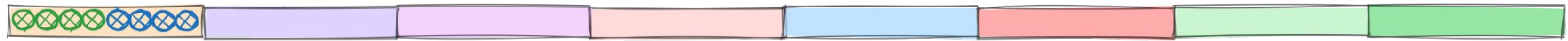


- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose slice height equal to number of cache sets



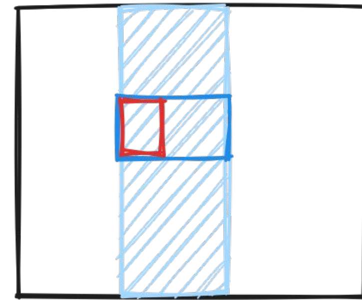


2 misses + 1 miss

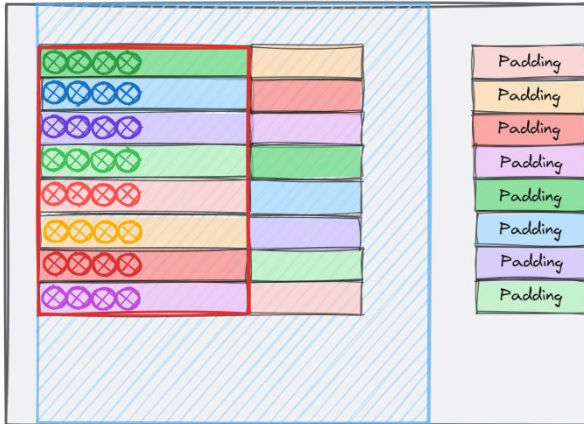


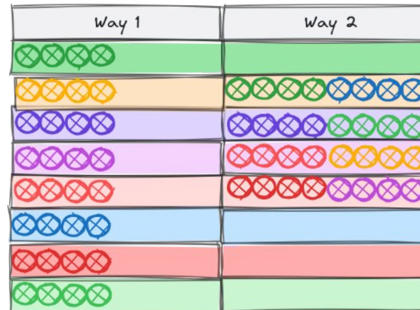
...

Improving predictability



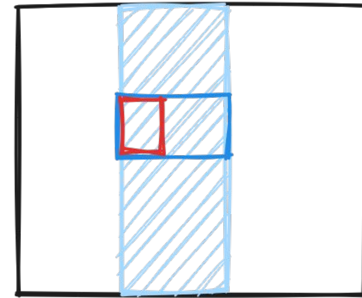
- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose slice height equal to number of cache sets



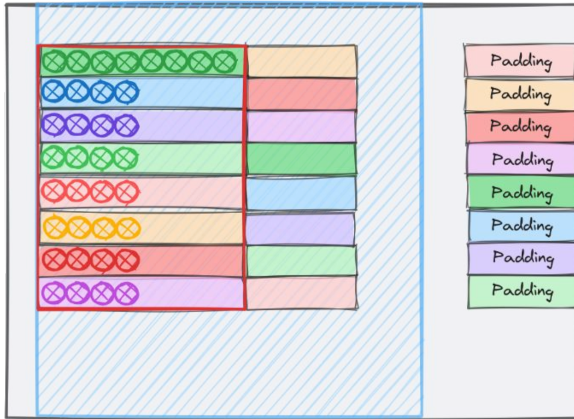


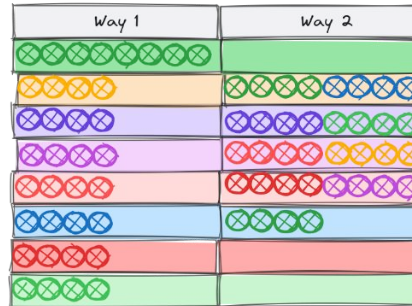
8 misses + 4 misses

Improving predictability



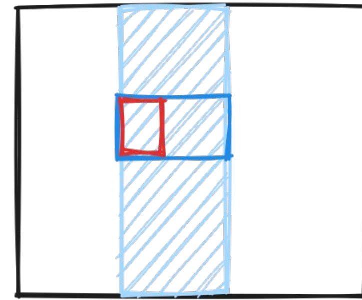
- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose slice height equal to number of cache sets



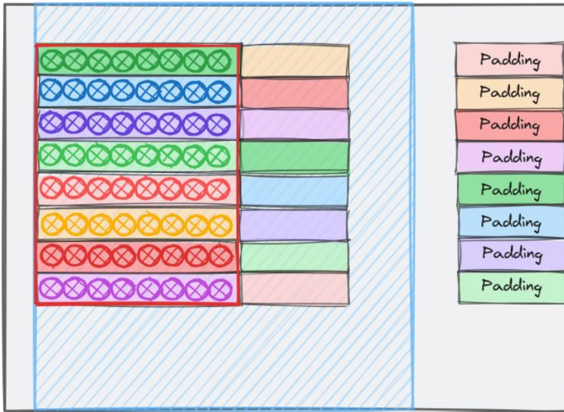


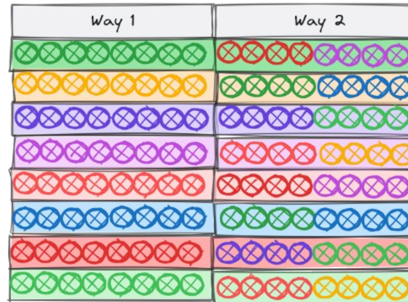
8 misses + 5 misses

Improving predictability

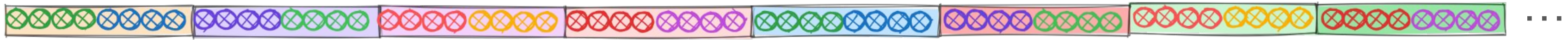


- Packing B
  - Exact prediction of number of memory accesses and L1 cache misses
  - Cache miss bound is theoretical minimum
- Packing A is handled in a similar fashion



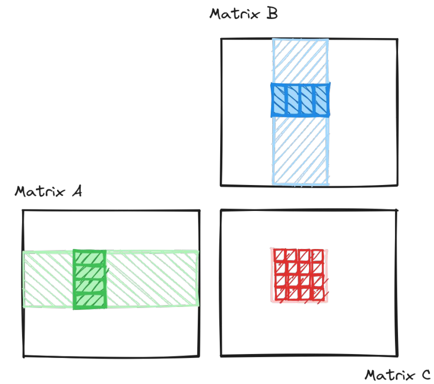


8 misses + 8 misses
i.e. number of blocks



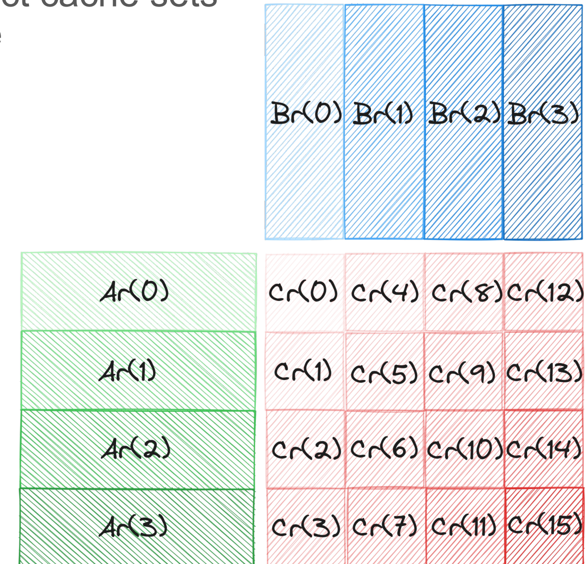
...

Improving predictability



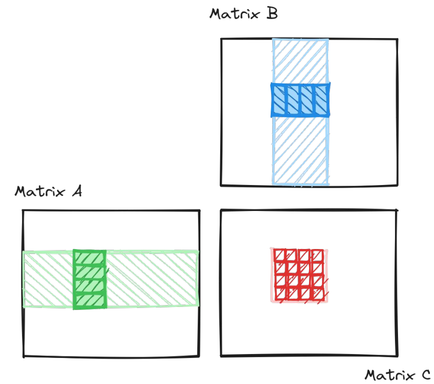
- Macro kernel
  - Rule 1: Matrix C is aligned to a cache block boundary
  - Rule 2: Matrix C is padded if necessary
  - Prop. 1: Buffer A, Buffer B aligned to a cache block boundary
  - Prop. 2: Micro-panels of A are contiguous in memory (same for B)
  - Prop. 3: Successive rows of each $C_r(i)$ mapped to distinct cache sets
  - Prop. 4: All micro-panels of A and B have the same size

- $C_r(0)$ += $A_r(0)$*$B_r(0)$;
- $C_r(1)$ += $A_r(1)$*$B_r(0)$;
- $C_r(2)$ += $A_r(2)$*$B_r(0)$;
- $C_r(3)$ += $A_r(3)$*$B_r(0)$;
- …
- $C_r(9)$ += $A_r(1)$*$B_r(2)$;
- …



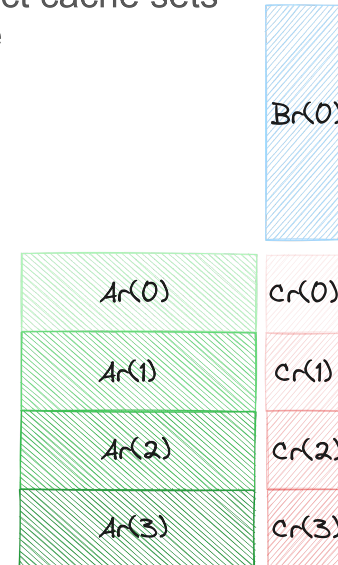
Improving predictability

- Macro kernel
  - Rule 1: Matrix C is aligned to a cache block boundary
  - Rule 2: Matrix C is padded if necessary
  - Prop. 1: Buffer A, Buffer B aligned to a cache block boundary
  - Prop. 2: Micro-panels of A are contiguous in memory (same for B)
  - Prop. 3: Successive rows of each $C_r(i)$ mapped to distinct cache sets
  - Prop. 4: All micro-panels of A and B have the same size



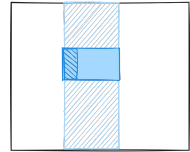
- $C_r(0)$ += $A_r(0)$*$B_r(0)$;
- $C_r(1)$ += $A_r(1)$*$B_r(0)$;
- $C_r(2)$ += $A_r(2)$*$B_r(0)$;
- $C_r(3)$ += $A_r(3)$*$B_r(0)$;
- …
- $C_r(9)$ += $A_r(1)$*$B_r(2)$;
- …

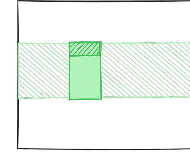
Reuse $B_r(0)$

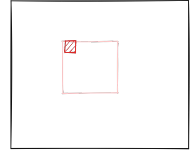


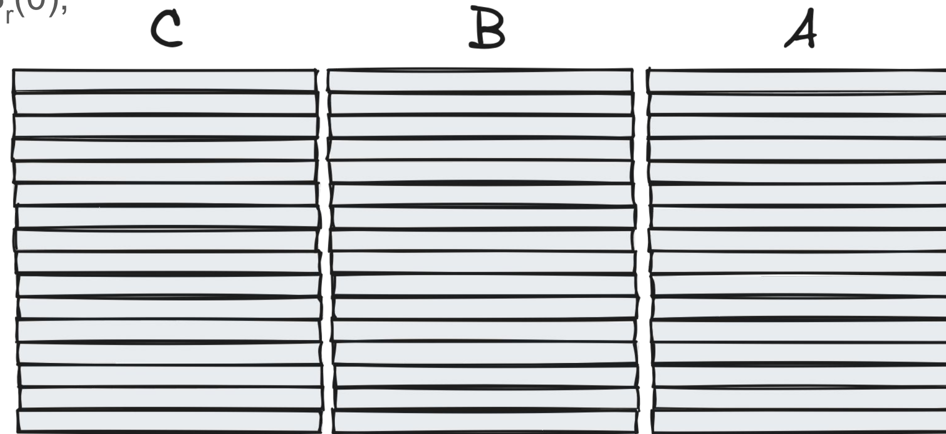
Improving predictability

- Macro kernel
  - $C_r(0) += A_r(0)*B_r(0);$ // R-$C_r(0)$; R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$;
  - $C_r(1) += A_r(1)*B_r(0);$ // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2) += A_r(2)*B_r(0);$
  - $C_r(3) += A_r(3)*B_r(0);$



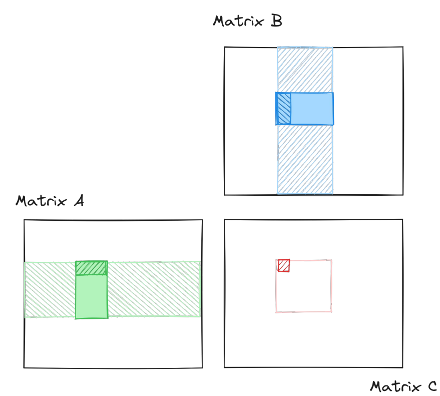


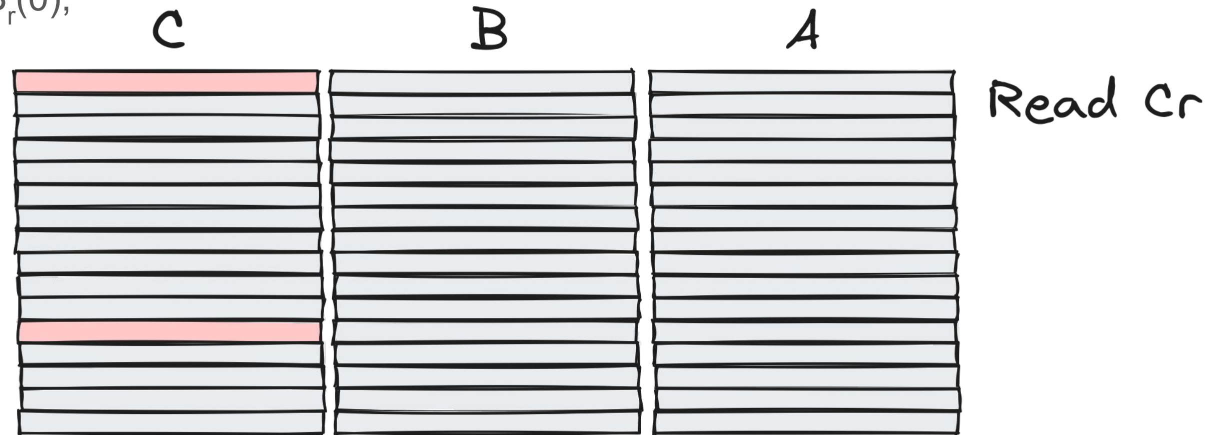




Improving predictability

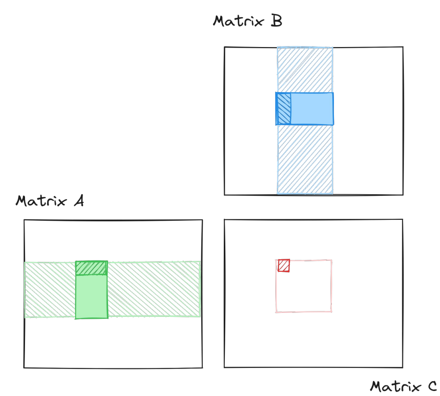
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // **R-C_r(0)**; R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

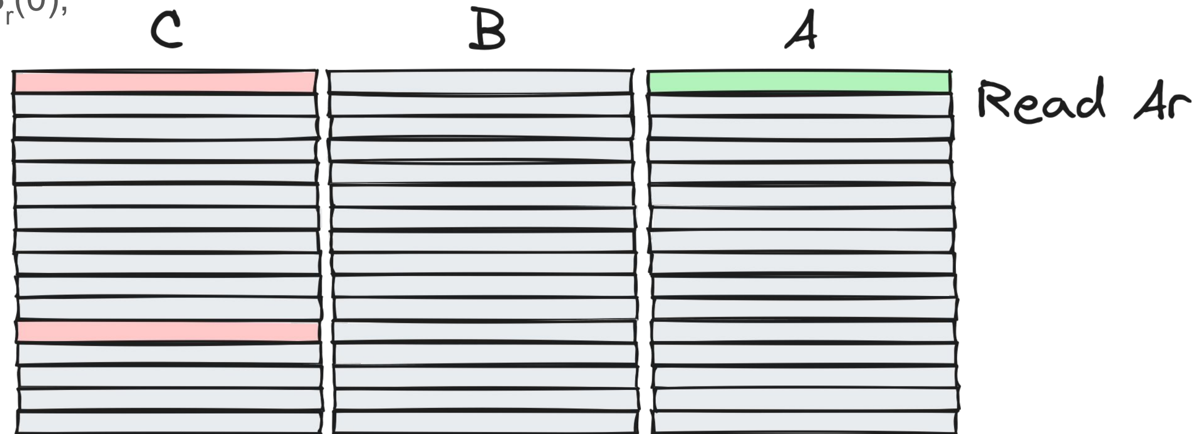




Improving predictability

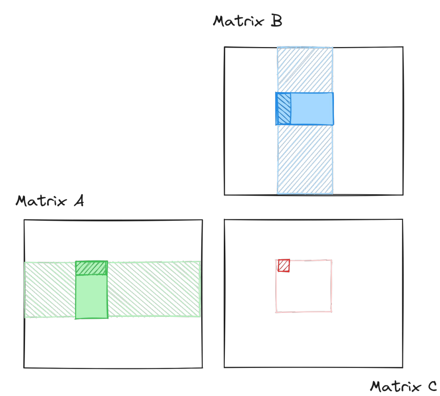
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; **R-A_r(0)||R-B_r(0)**;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

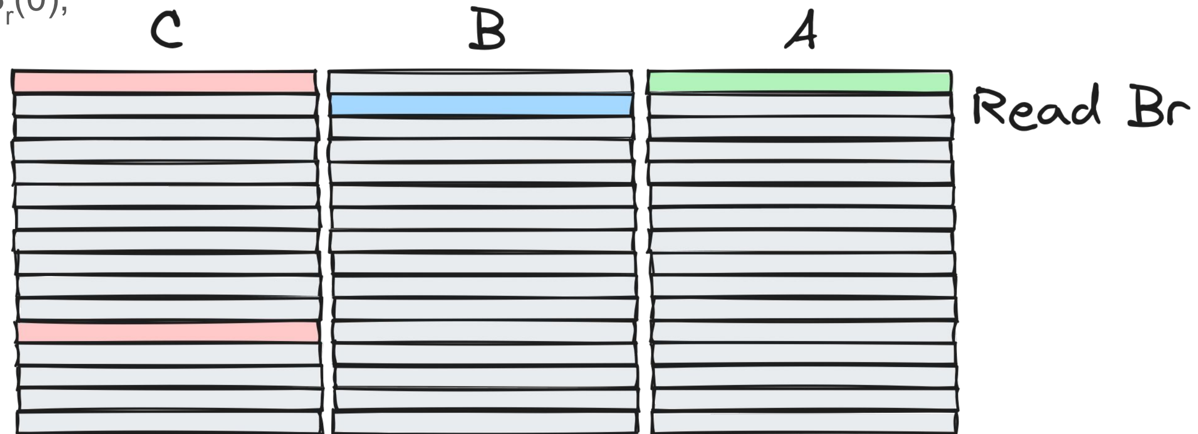




Improving predictability

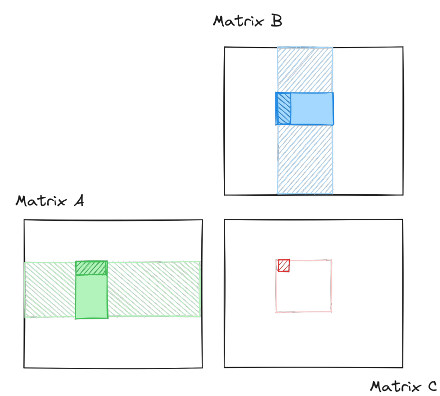
- Macro kernel
  - $C_r(0) \mathrel{+}= A_r(0)*B_r(0)$; // R-$C_r(0)$; **R-$A_r(0)$||R-$B_r(0)$**;W-$C_r(0)$;
  - $C_r(1) \mathrel{+}= A_r(1)*B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2) \mathrel{+}= A_r(2)*B_r(0)$;
  - $C_r(3) \mathrel{+}= A_r(3)*B_r(0)$;

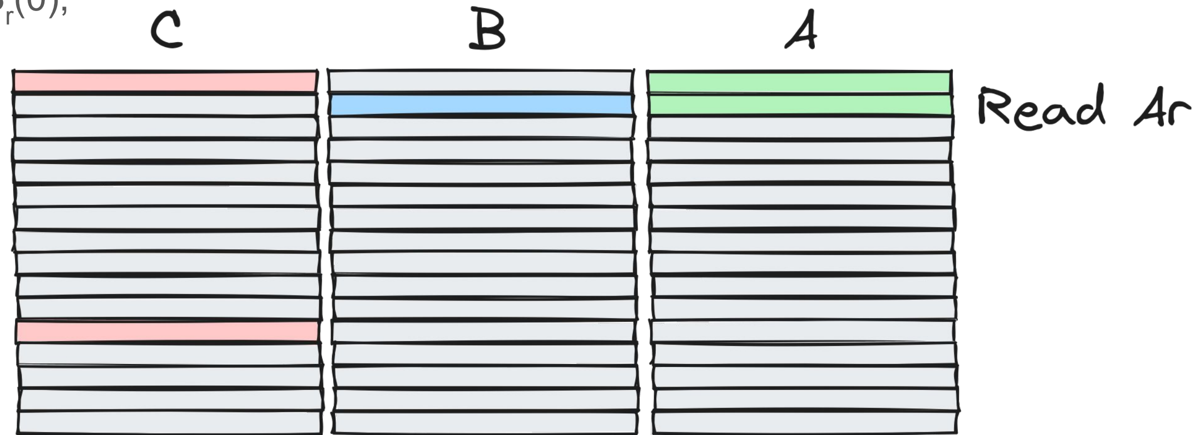




Improving predictability

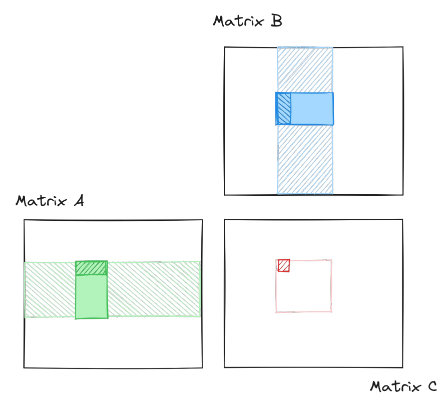
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; **R-$A_r(0)$||R-$B_r(0)$**;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

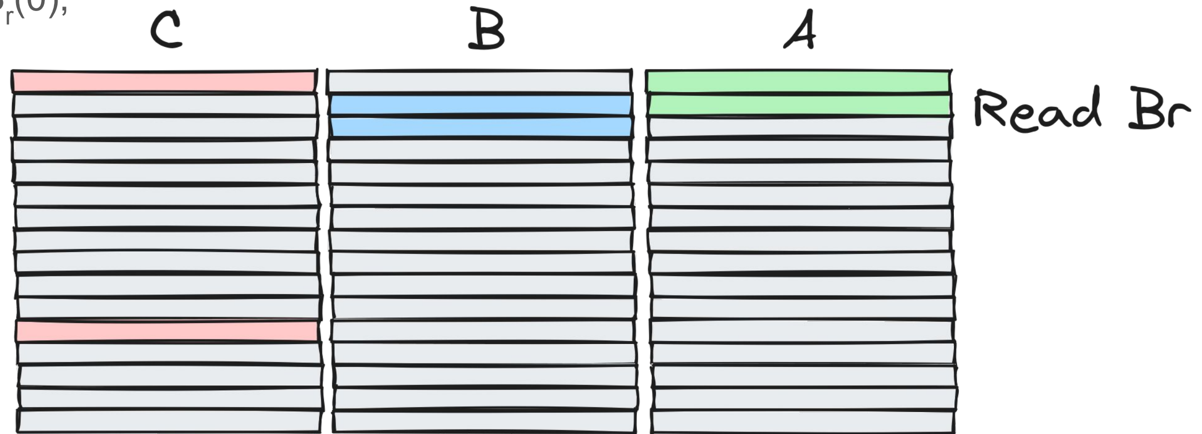




Improving predictability

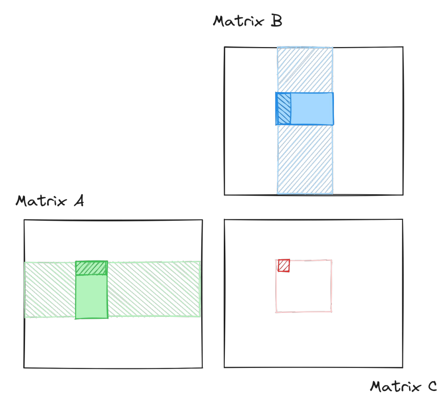
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; **R-A_r(0)||R-B_r(0)**;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

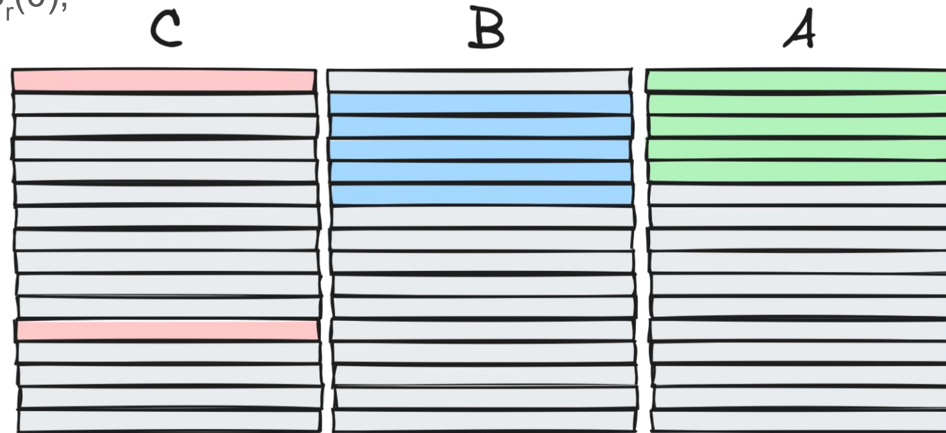




Improving predictability

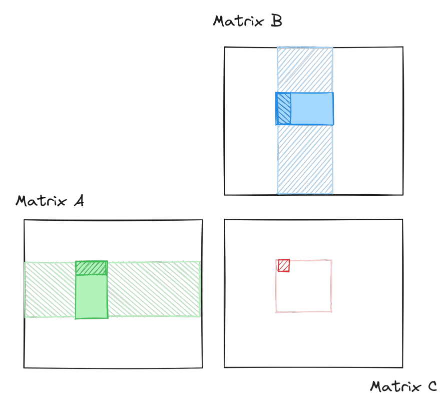
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; **R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$**;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

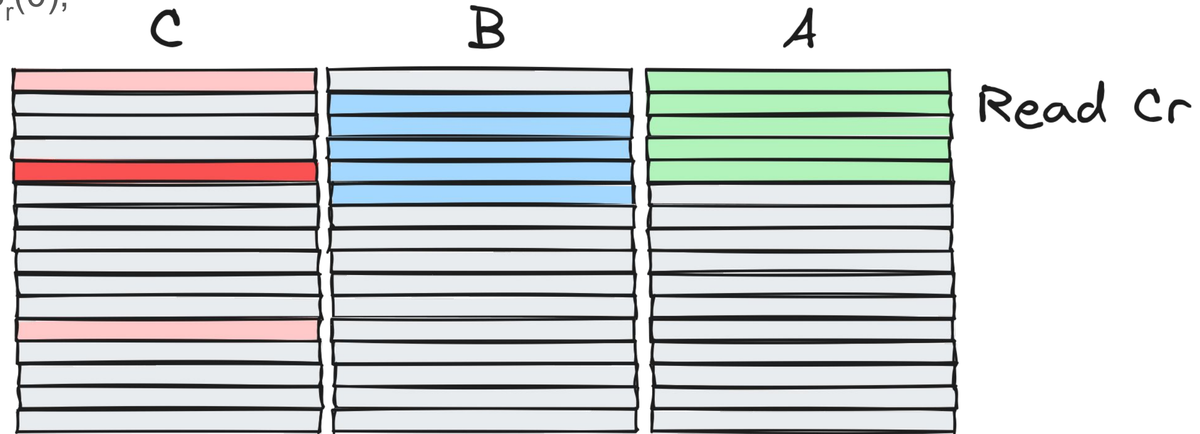




Improving predictability

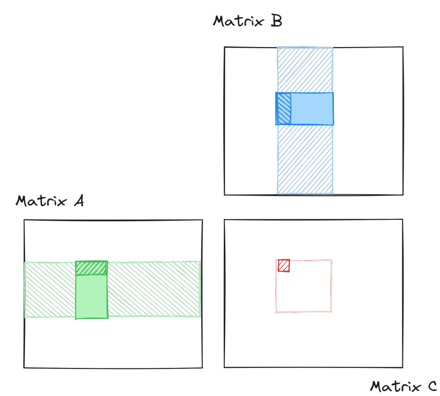
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // **R-C_r(1)**; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;



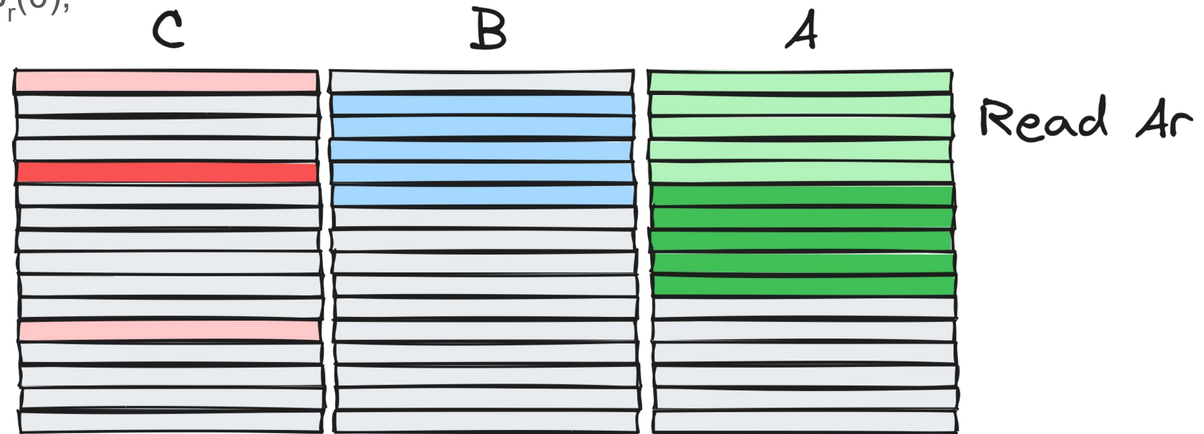


Improving predictability

- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; **R-$A_r(1)$||R-$B_r(0)$**;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

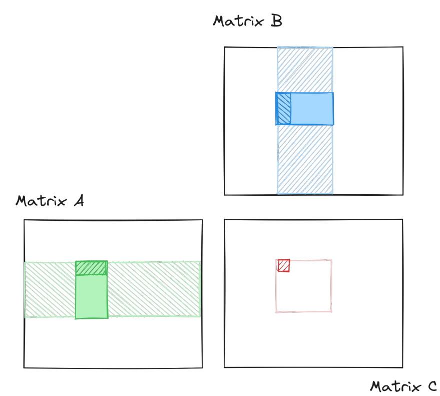


Number of cache misses: Load Cr, Load Br, Load Ar + reload evicted blocks

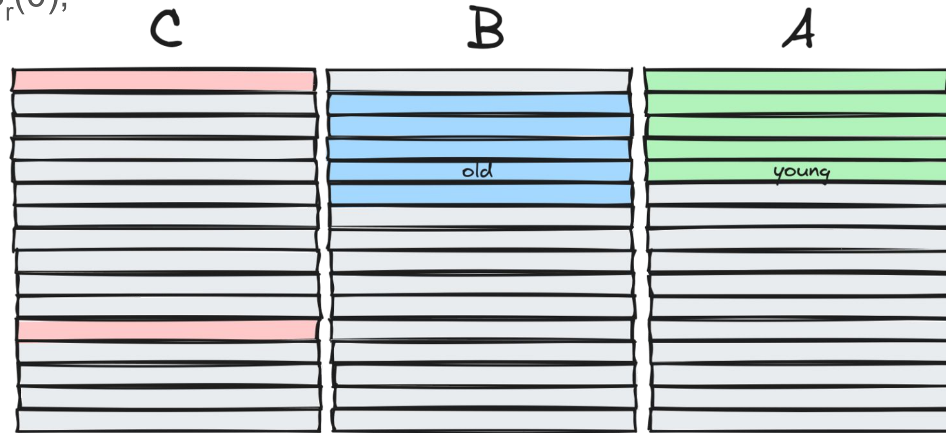


Improving predictability

- Macro kernel
  - $C_r(0) \mathrel{+}= A_r(0) * B_r(0)$; // R-$C_r(0)$; **R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$**;
  - $C_r(1) \mathrel{+}= A_r(1) * B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2) \mathrel{+}= A_r(2) * B_r(0)$;
  - $C_r(3) \mathrel{+}= A_r(3) * B_r(0)$;

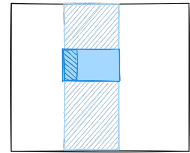


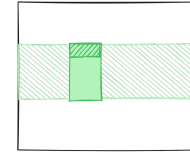
Case 1: Br gets old because of Ar and is evicted by Cr

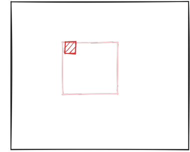


Improving predictability

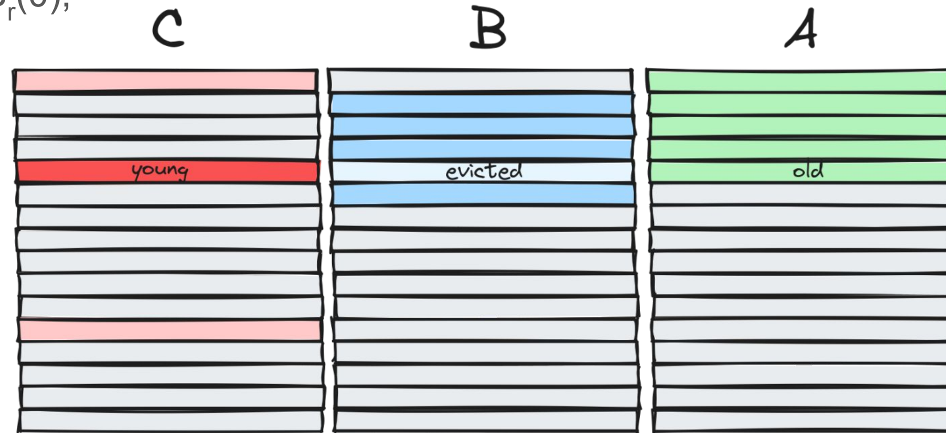
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // **R-$C_r(1)$**; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;





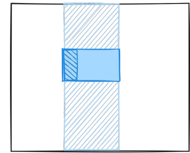


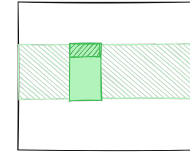
Case 1: Br gets old because of Ar and is evicted by Cr

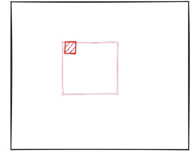


Improving predictability

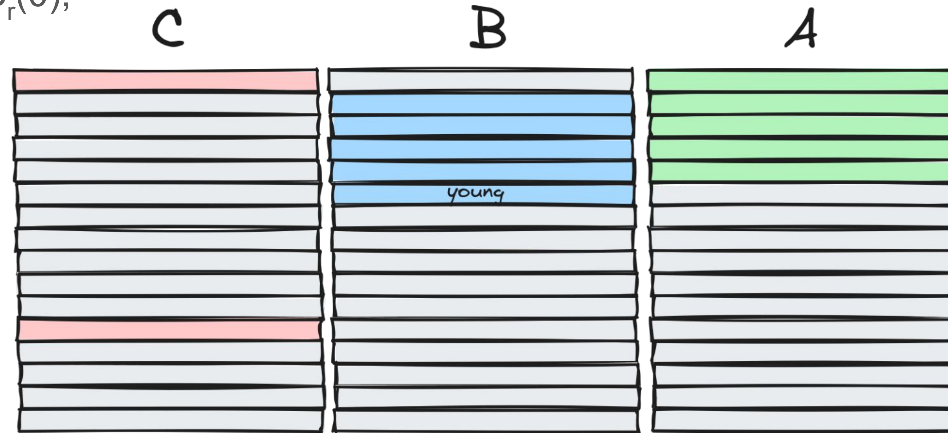
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; **R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$**;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;





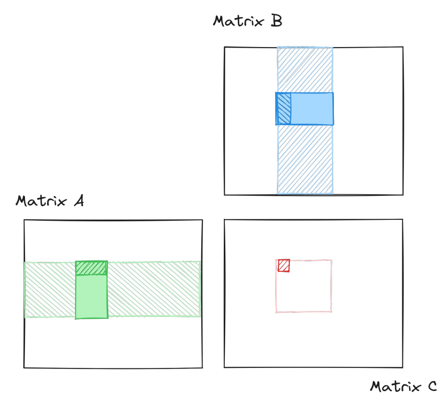


Case 2: Br gets old because of Cr and evicted by Ar

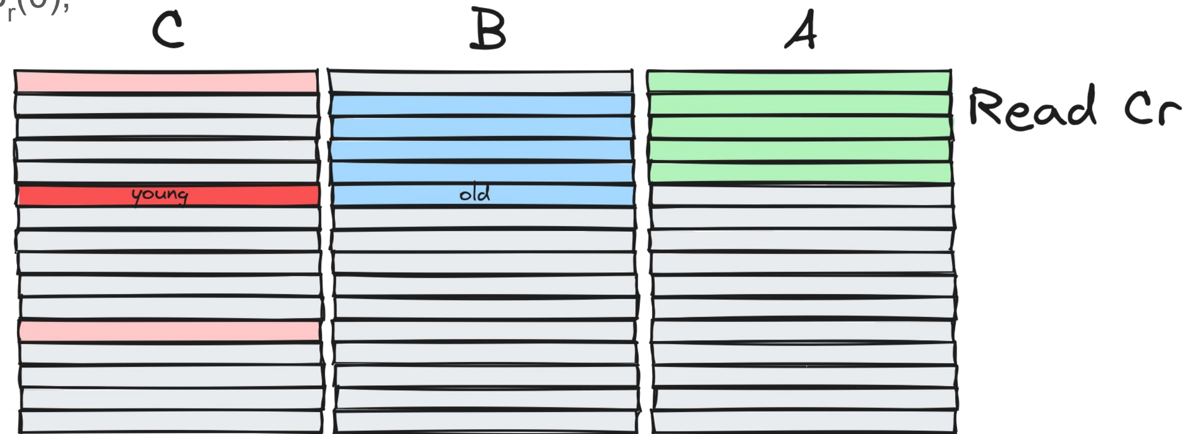


Improving predictability

- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // **R-$C_r(1)$**; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

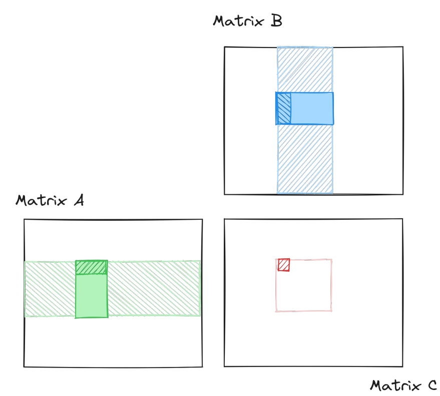


Case 2: Br gets old because of Cr and evicted by Ar

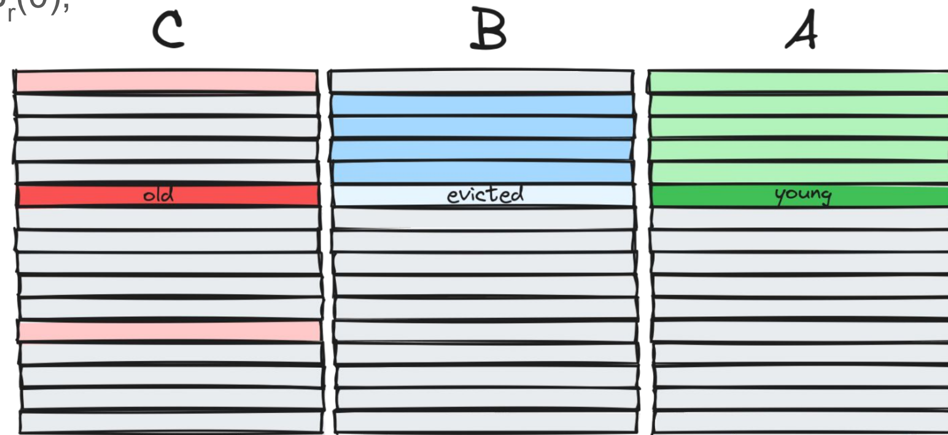


Improving predictability

- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; R-$A_r(0)$||R-$B_r(0)$;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; **R-$A_r(1)$||R-$B_r(0)$**;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

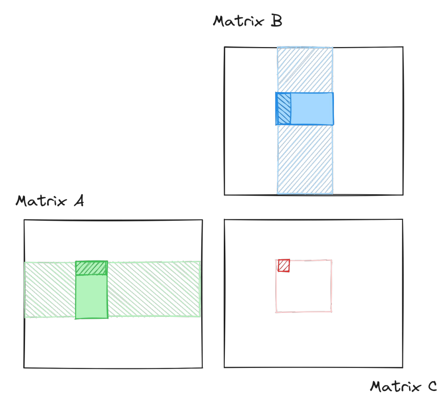


Case 2: Br gets old because of Cr and evicted by Ar

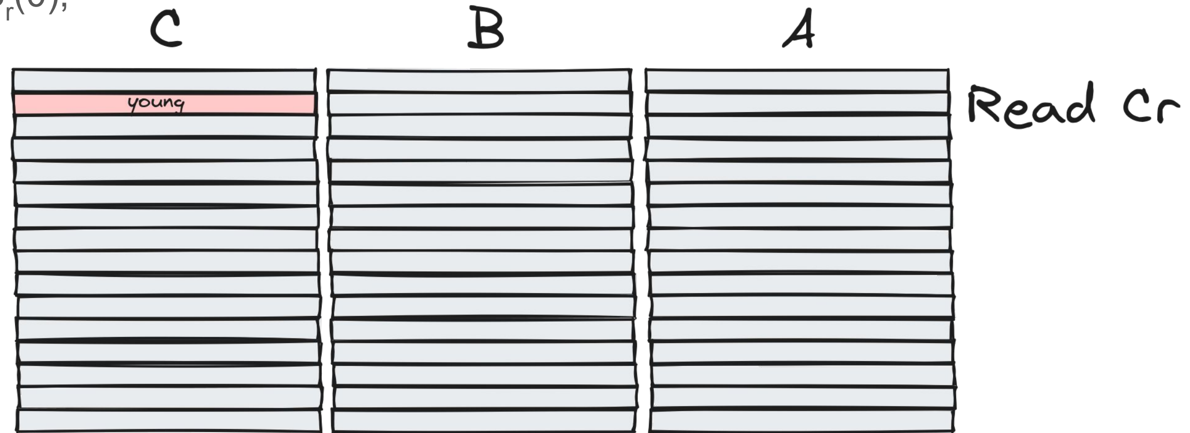


Improving predictability

- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // **R-C_r(0)**; R-A_r(0)||R-B_r(0);W-C_r(0);
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-C_r(1); R-A_r(1)||R-B_r(0);W-C_r(1);
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

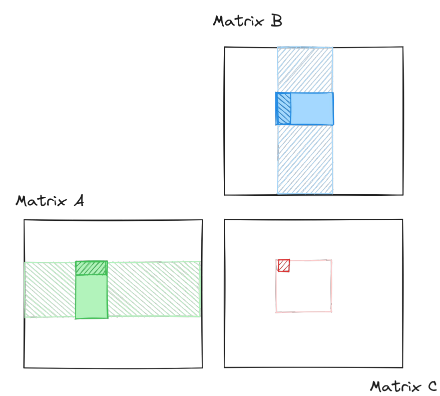


Case 3: Cr gets evicted because of Ar and Br

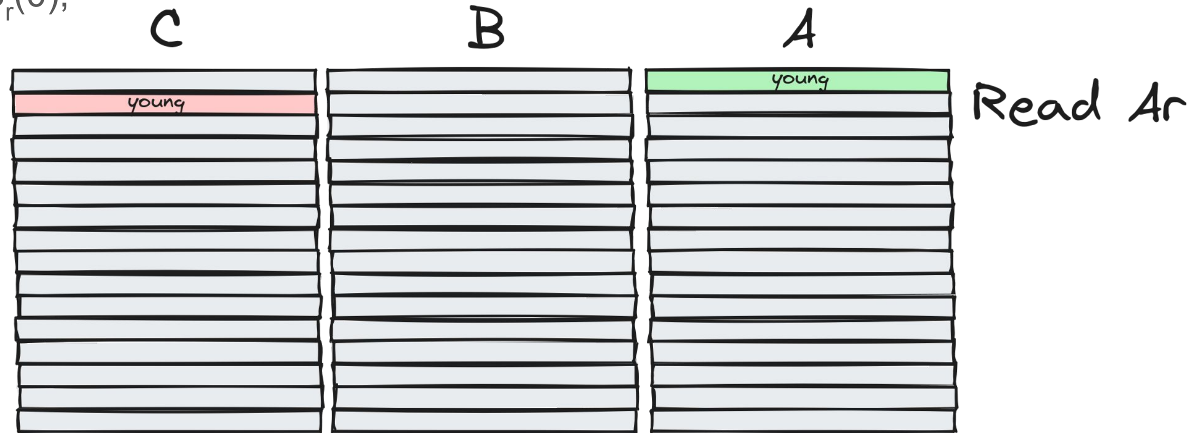


Improving predictability

- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; **R-A_r(0)**||R-$B_r(0)$;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;

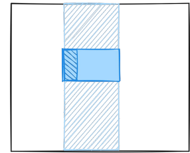


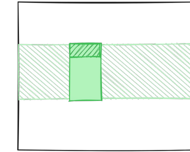
Case 3: Cr gets evicted because of Ar and Br

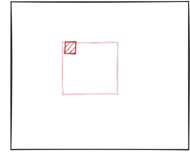


Improving predictability

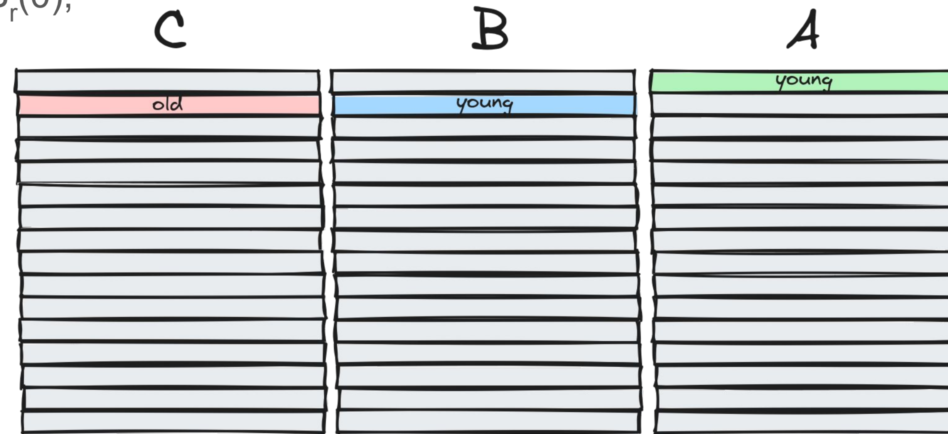
- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; **R-$A_r(0)$||R-$B_r(0)$**;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;





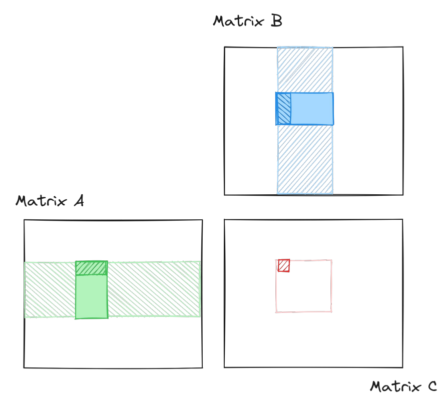


Case 3: Cr gets evicted because of Ar and Br

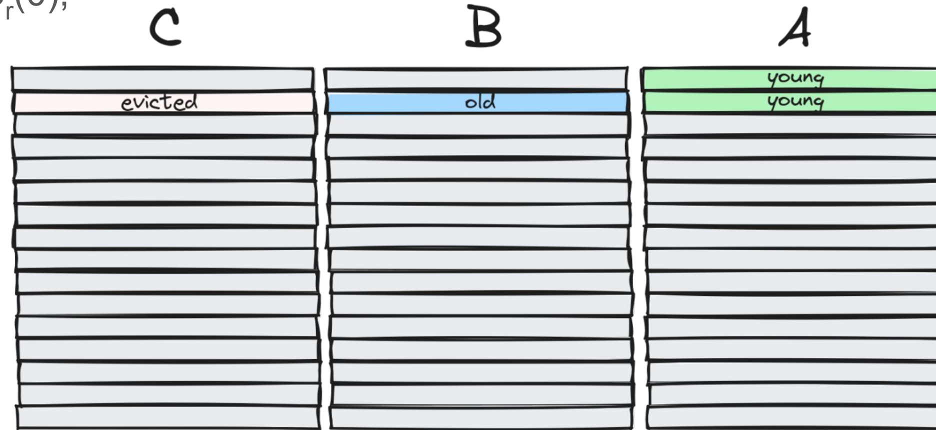


Improving predictability

- Macro kernel
  - $C_r(0)$ += $A_r(0)$*$B_r(0)$; // R-$C_r(0)$; **R-$A_r(0)$||R-$B_r(0)$**;W-$C_r(0)$;
  - $C_r(1)$ += $A_r(1)$*$B_r(0)$; // R-$C_r(1)$; R-$A_r(1)$||R-$B_r(0)$;W-$C_r(1)$;
  - $C_r(2)$ += $A_r(2)$*$B_r(0)$;
  - $C_r(3)$ += $A_r(3)$*$B_r(0)$;



Case 3: Cr gets evicted because of Ar and Br



Improving predictability

- For packing A and B, formulas predict misses with max 0.03% overhead
- For macro-kernel, overhead of 5% on average
  - Except particular case with small blocks A_c and C_c (up to 13% overhead)
- Less than 5% overhead in average execution time
- For packing B, L1D refill reduced by up to 60% in the best case

TABLE III
MEASURES FOR THE NUMBER OF MEMORY ACCESS AND L1D REFILL OF THE GEMM ROUTINE.

Matrix configuration					Memory accesses			L1D refill				
	m	n	k		***Ours***	***Low et al.***		***Ours***	***Low et al.***		**Theoretical**	
(i)	272	272	272	Packing B	148 032	148 000	(-0.02%)	9 253	10 612	(12.81%)	9 254	(0.01%)
				Packing A	148 032	148 000	(-0.02%)	9 252	9 251	(-0.01%)	9 254	(0.02%)
				Macro-kernel	2 811 414	2 663 435	(-5.26%)	333 950	347 307	(3.85%)	384 886	(13.23%)
				Total	3 107 478	2 959 435	(-4.76%)	352 455	367 170	(4.01%)	403 394	(12.63%)
(ii)	528	528	528	Packing B	557 664	557 632	(-0.01%)	34 856	78 145	(55.40%)	34 857	(0.00%)
				Packing A	557 664	557 632	(-0.01%)	34 854	34 852	(-0.01%)	34 857	(0.01%)
				Macro-kernel	20 072 481	19 514 902	(-2.78%)	2 552 600	2 509 309	(-1.73%)	2 703 897	(5.60%)
				Total	21 187 809	20 630 166	(-2.63%)	2 622 310	2 622 306	(0.00%)	2 773 611	(5.46%)
(iii)	256	784	2,016*	Packing B	3 161 344	3 161 216	(0.00%)	197 591	341 213	(42.09%)	197 592	(0.00%)
				Packing A	1 032 448	1 032 320	(-0.01%)	64 528	64 520	(-0.01%)	64 536	(0.01%)
				Macro-kernel	53 788 760	52 183 084	(-2.99%)	6 894 927	6 766 186	(-1.90%)	7 217 528	(4.47%)
				Total	57 982 552	56 376 620	(-2.77%)	7 157 046	7 171 919	(0.21%)	7 479 656	(4.31%)
(iv)	192	736*	528	Packing B	777 312	777 248	(-0.01%)	48 584	121 948	(60.16%)	48 585	(0.00%)
				Packing A	202 848	202 784	(-0.03%)	12 677	12 675	(-0.02%)	12 681	(0.03%)
				Macro-kernel	10 174 497	9 891 862	(-2.78%)	1 248 905	1 258 791	(0.79%)	1 322 057	(5.53%)
				Total	11 154 657	10 871 894	(-2.53%)	1 310 166	1 393 414	(5.97%)	1 383 323	(5.29%)

* Padding applied.

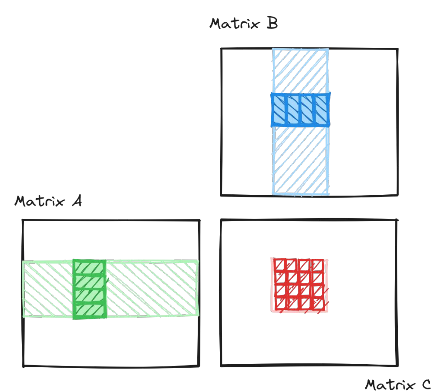
Conclusion

- Predictable and traceable yet efficient implementation of a blocked GEMM algorithm
  - Execution time reduced by up to 99% compared to unoptimized code, with no compiler optimization
  - L1D cache misses over-estimated by 5% on average, except for one matrix configuration
  - Cost of predictability is less than 5%
- Future work
  - Extend our formulas to predictable L2 caches
  - Refine analysis for the problematic case
  - Automatic code generation targeting particular architectures and matrix configurations
    - Instead of generic library

Thank you for your attention

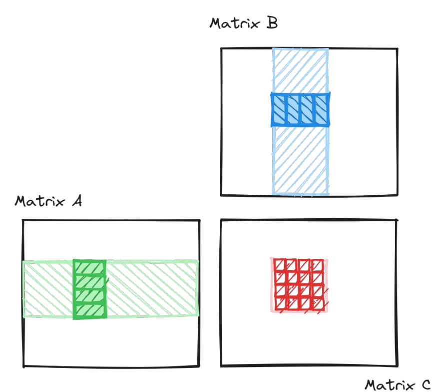
Questions ?

Blocked GEMM (Micro-kernel)



Outer product

Blocked GEMM (Micro-kernel)



Matrix B

Matrix A

Product

Matrix C

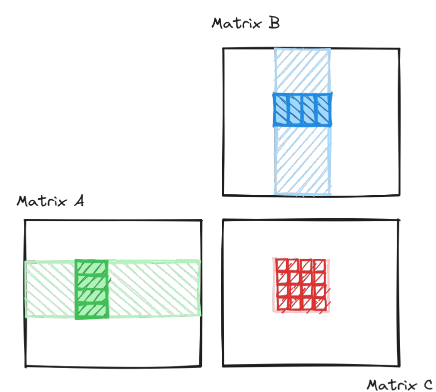
Blocked GEMM (Micro-kernel)

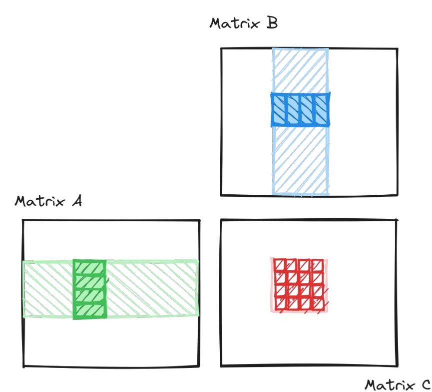
Matrix B

Product

Matrix A

Matrix C





Blocked GEMM (Micro-kernel)

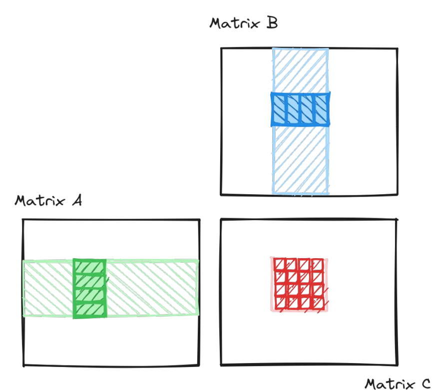
Matrix B

Product

Matrix A

Matrix C

Blocked GEMM (Micro-kernel)



Matrix B

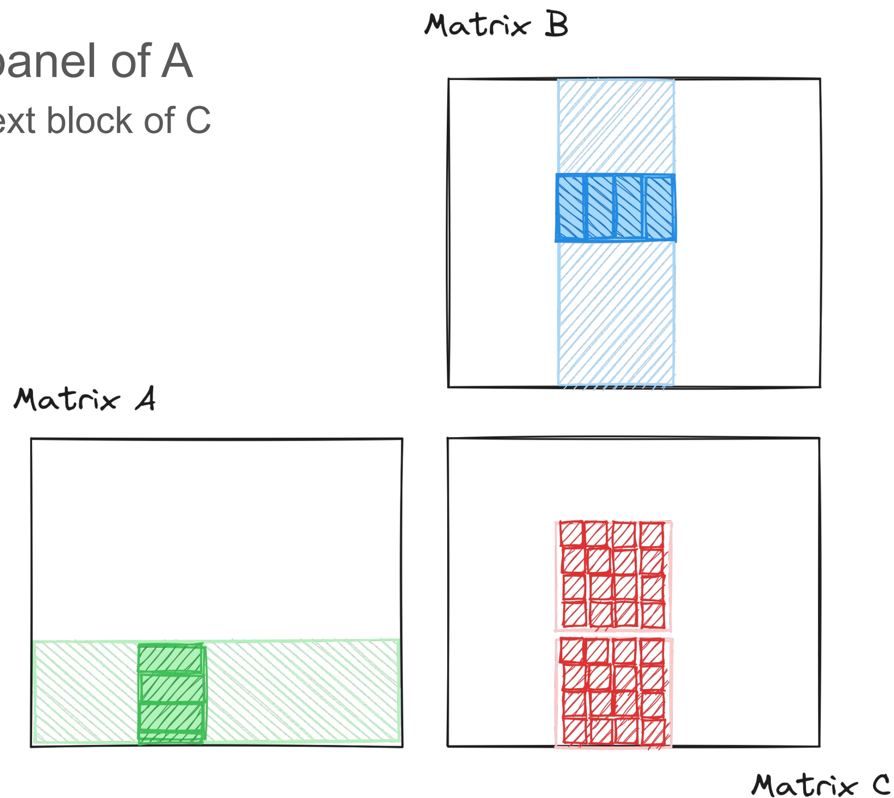
Matrix A

Multiply-Accumulate

Matrix C

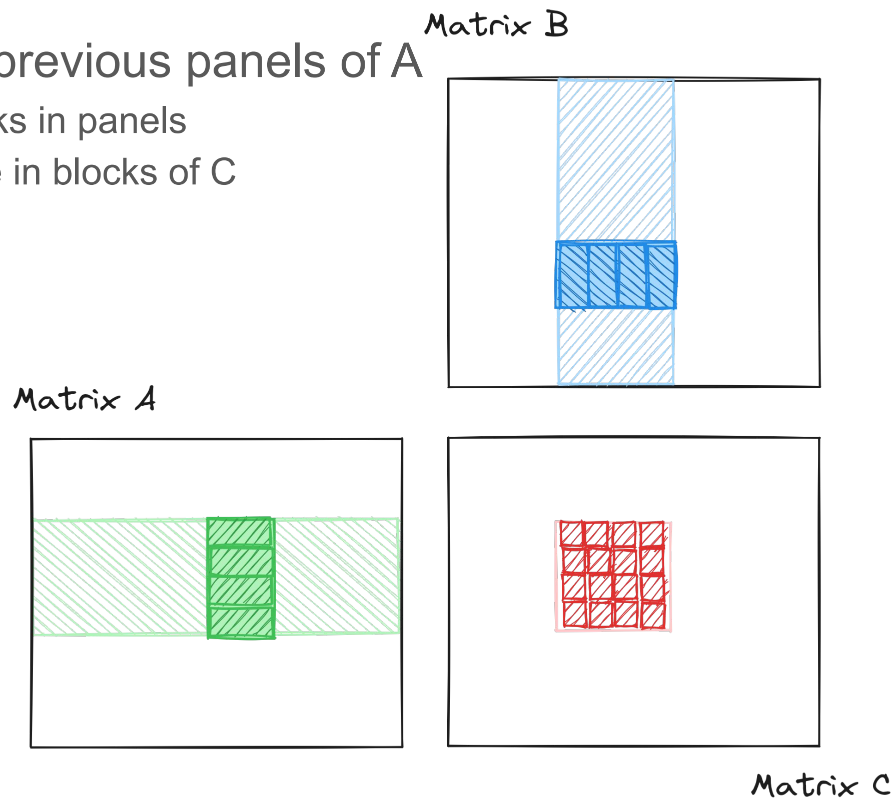
Blocked GEMM (macro-kernel)

- Switch to next panel of A
  - → compute next block of C

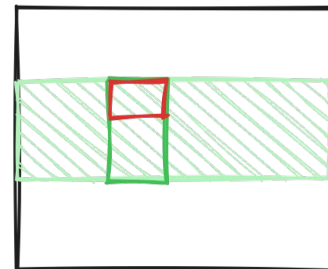


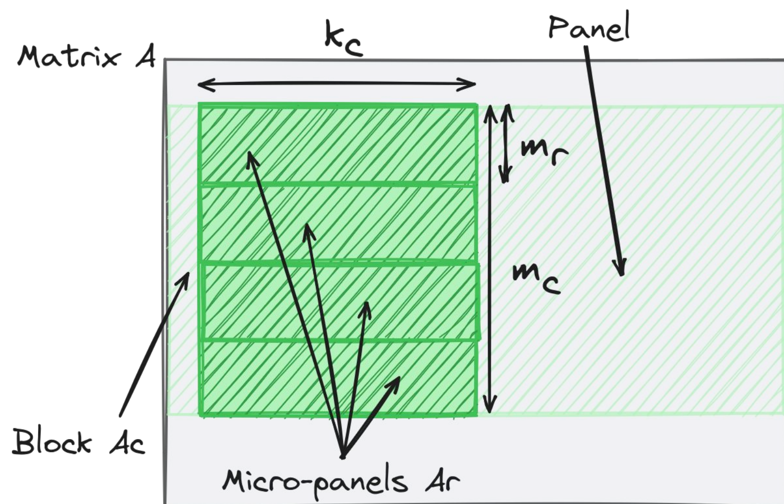
Blocked GEMM (macro-kernel)

- Switch back to previous panels of A
  - Use next blocks in panels
  - → accumulate in blocks of C

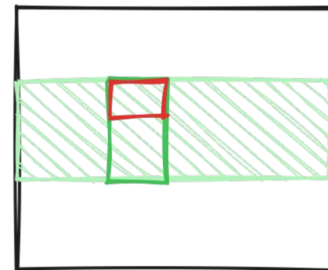


Blocked GEMM (Packing A)



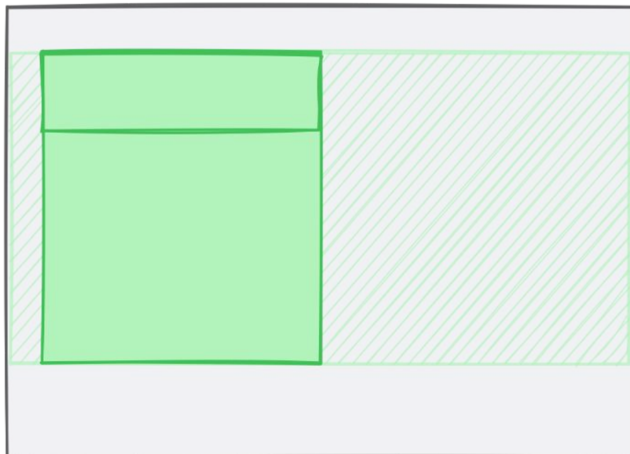


Matrix A



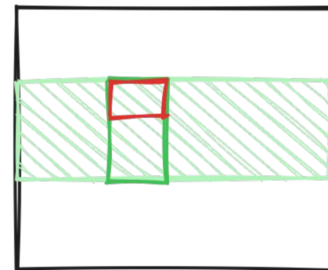
Blocked GEMM (Packing A)

Matrix A

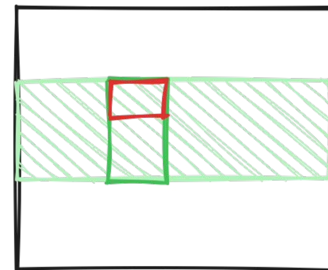


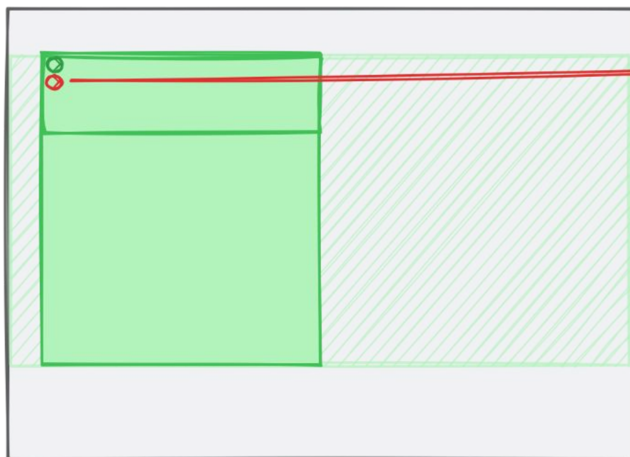
Packed Panel Buffer $\tilde{A}_c$

Blocked GEMM (Packing A)

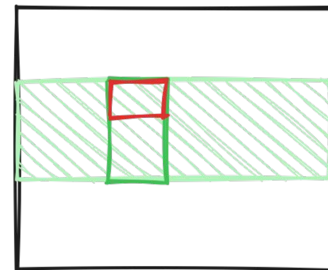


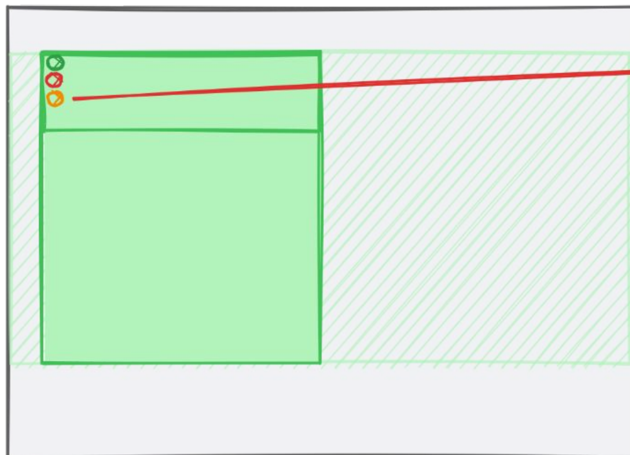
Blocked GEMM (Packing A)



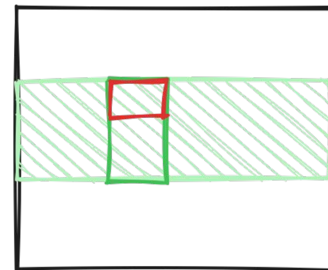


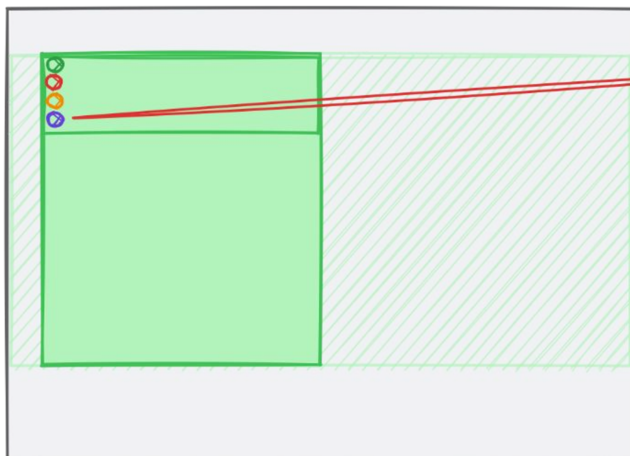
Blocked GEMM (Packing A)

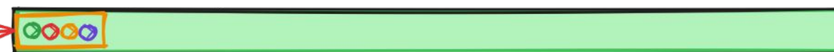




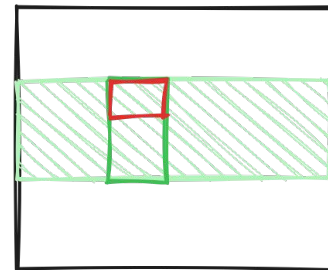
Blocked GEMM (Packing A)



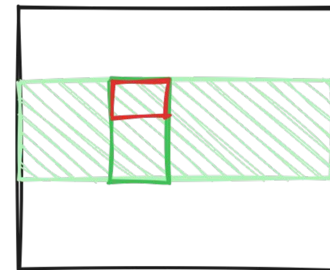




Blocked GEMM (Packing A)



Blocked GEMM (Packing A)



Matrix A

k_c

m_r

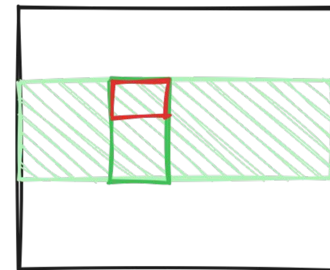
Copy

Packed Panel Buffer $\tilde{A}_c$

•••

$k_c * m_r$

Blocked GEMM (Packing A)



Matrix A

k_c

m_c

Copy

Packed Panel Buffer $\tilde{A}_c$

$k_c * m_c$

Parameter tuning

- Traditionally, the blocks dimensions are chosen empirically so that:
  - Micro-panels B_r stay in the L1 cache, one at a time
  - The packed buffer $\tilde{A}_c$ stays in the L2 cache
  - The packed buffer $\tilde{B}_c$ stays in the L3 cache, if any

- Low et al. [1] proposed an analytical method to tune the parameters automatically to reduce execution time
  - Here, same philosophy for predictability

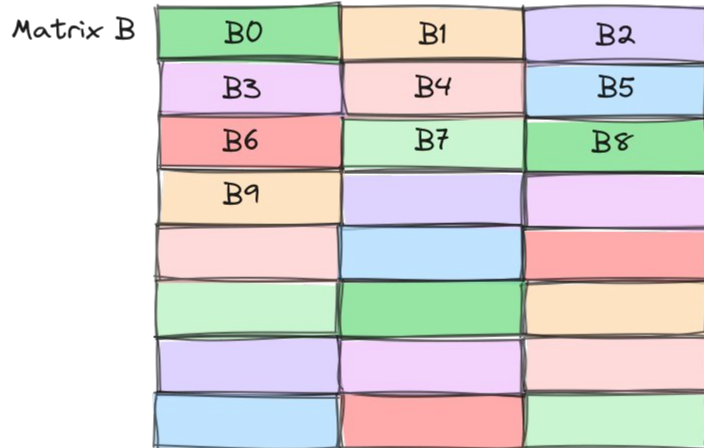
[1] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, “Analytical modeling is enough for high-performance BLIS”

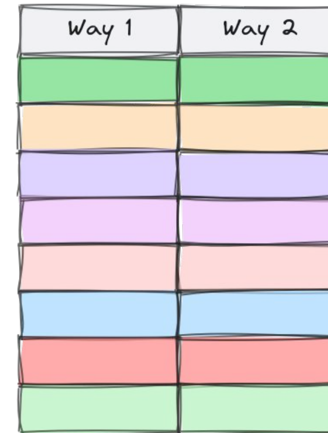
ACM Transactions on Mathematical Software, vol. 43, no. 2, pp. 1–18, Aug. 2016

Improving predictability

- General theorem (strided accesses in a matrix)

Theorem 1. *Let $b, n \in \mathbb{N}$. If n is odd, then the cache blocks of indices $b, b+n, b+2\cdot n, \ldots, b+(s_{Li}-1)\cdot n$ are all mapped to different cache sets.*

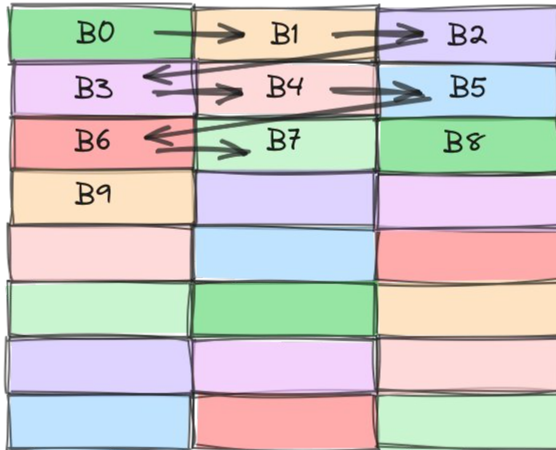




Improving predictability

- General theorem (strided accesses in a matrix)

Theorem 1. *Let $b, n \in \mathbb{N}$. If n is odd, then the cache blocks of indices $b, b+n, b+2\cdot n, \ldots, b+(s_{Li}-1)\cdot n$ are all mapped to different cache sets.*



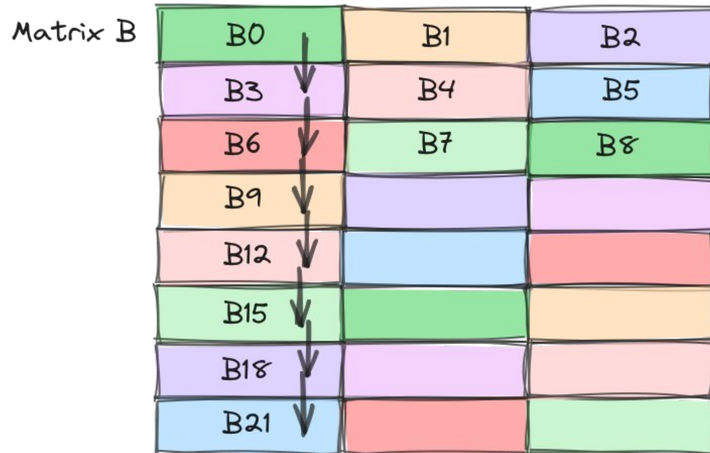
Ex. 1: b=0, n=1

Way 1	Way 2
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	

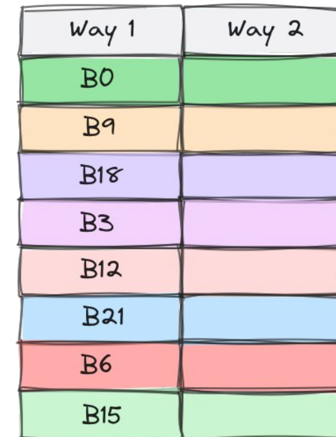
Improving predictability

- General theorem (strided accesses in a matrix)

Theorem 1. *Let* $b, n \in \mathbb{N}$. *If* n *is odd, then the cache blocks of indices* $b, b+n, b+2\cdot n, \ldots, b+(s_{Li}-1)\cdot n$ *are all mapped to different cache sets.*



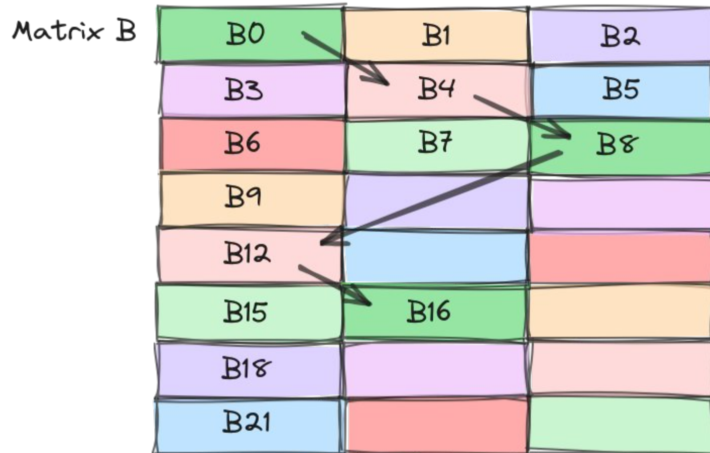
Ex. 2: b=0, n=3



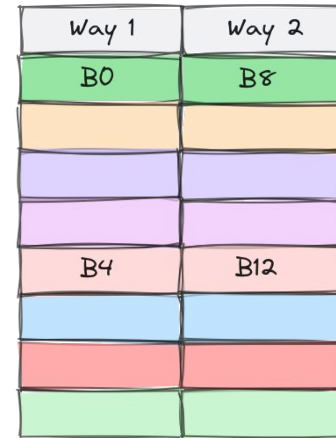
Improving predictability

- General theorem (mapping a matrix to a data cache with s_{Li} sets)

Theorem 1. *Let $b, n \in \mathbb{N}$. If n is odd, then the cache blocks of indices $b, b+n, b+2 \cdot n, \ldots, b+(s_{Li}-1) \cdot n$ are all mapped to different cache sets.*

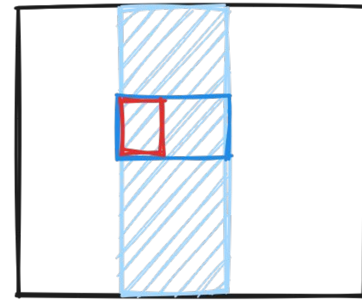


Counter ex.: b=0, n=4

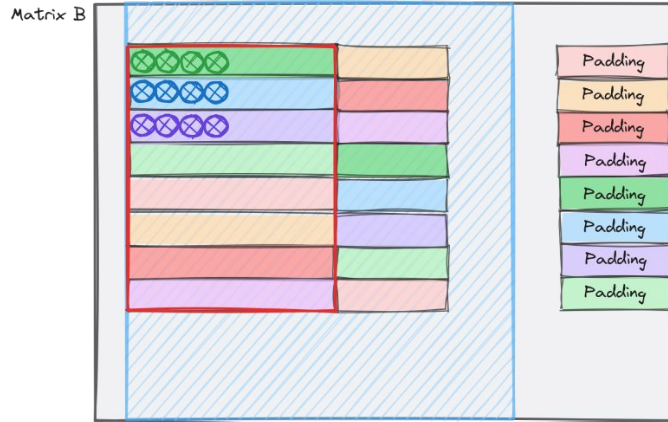


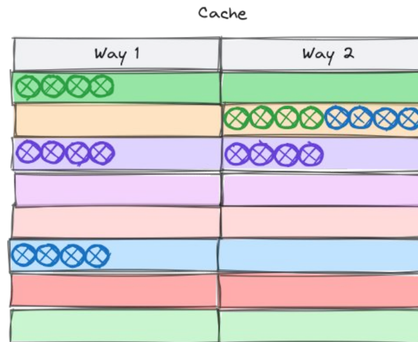
B16

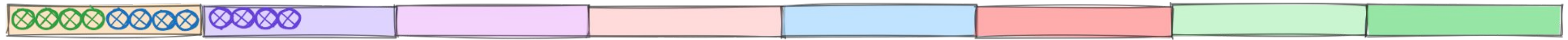
Improving predictability



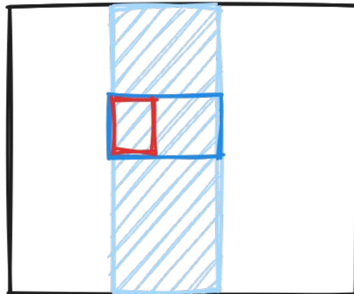
- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose vector width as a divisor of cache block size
  - Rule 4: Choose slice height equal to number of cache sets



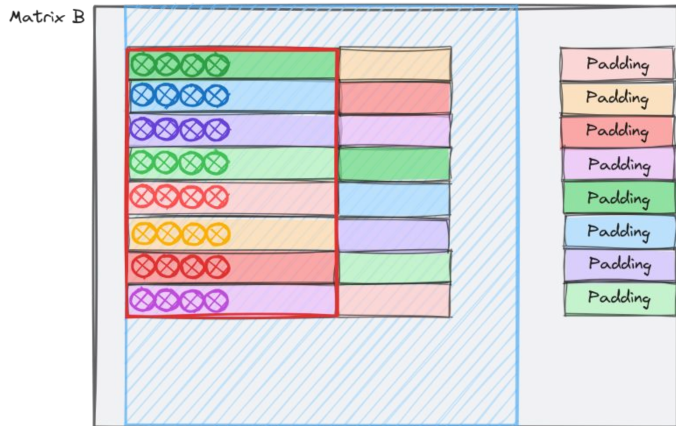


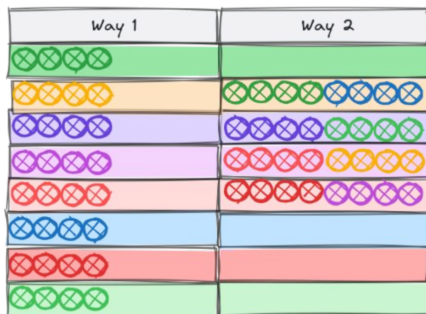


Improving predictability

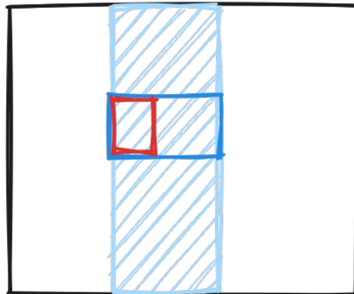


- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose vector width as a divisor of cache block size
  - Rule 4: Choose slice height equal to number of cache sets

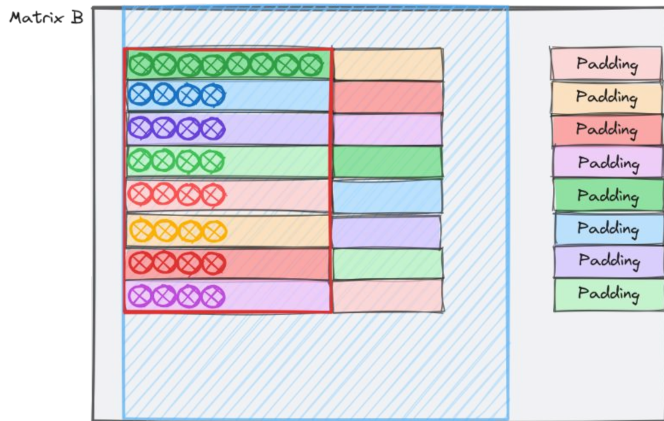


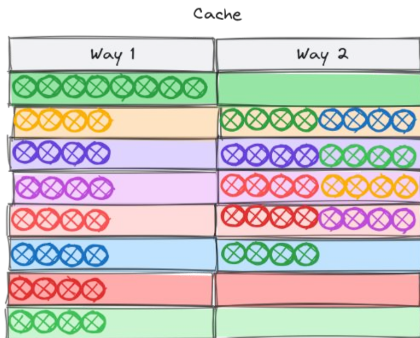


Improving predictability

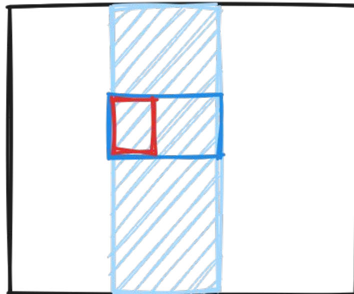


- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose vector width as a divisor of cache block size
  - Rule 4: Choose slice height equal to number of cache sets

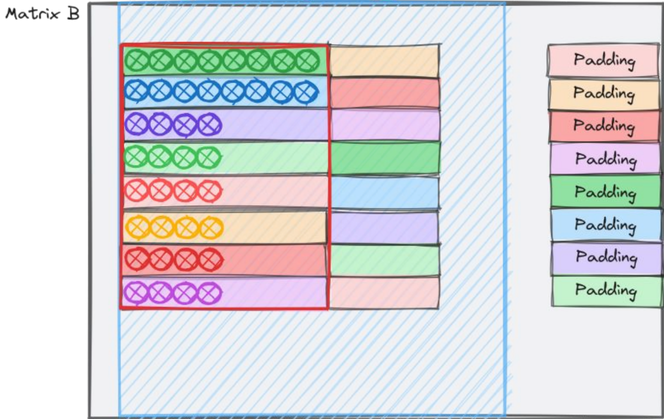


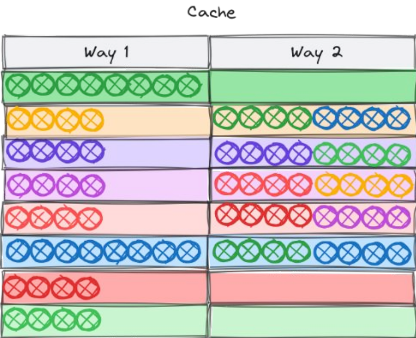


Improving predictability



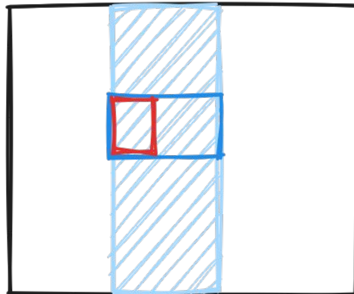
- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose vector width as a divisor of cache block size
  - Rule 4: Choose slice height equal to number of cache sets





Buffer $\tilde{B}_c$

Improving predictability



- Packing B
  - Rule 1: Align Matrix B and packing buffer to cache block boundaries
  - Rule 2: Pad rows of matrix B to an odd number of complete cache blocks (if necessary)
  - Rule 3: Choose vector width as a divisor of cache block size
  - Rule 4: Choose slice height equal to number of cache sets