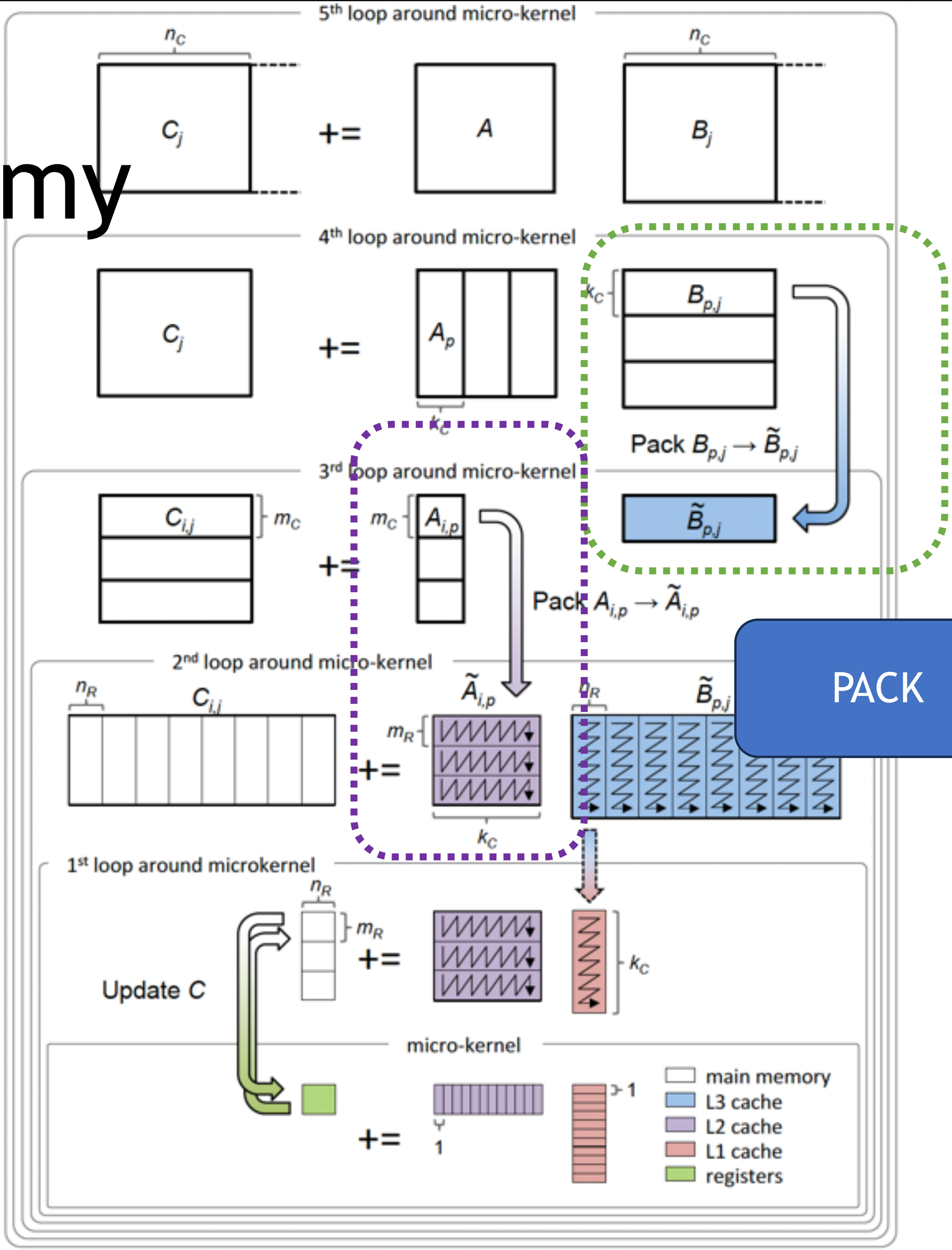
GEMMFIP - Towards a Unified Implementation of GEMM in BLIS & An Attempt towards Code-Gen Kernels

Aug. 2023

Xu, Ruqing G.

Univ. Tokyo
NVIDIA (Intern)

BLIS' Autonomy



PACK

then GEMM

The Pack-then-Gemm Way

Pros:

- Packing cost $O(n^2)$ is much smaller than Gemm kernels' $O(n^3)$ cost;
- Packing kernels is easier to write than Gemm kernels.

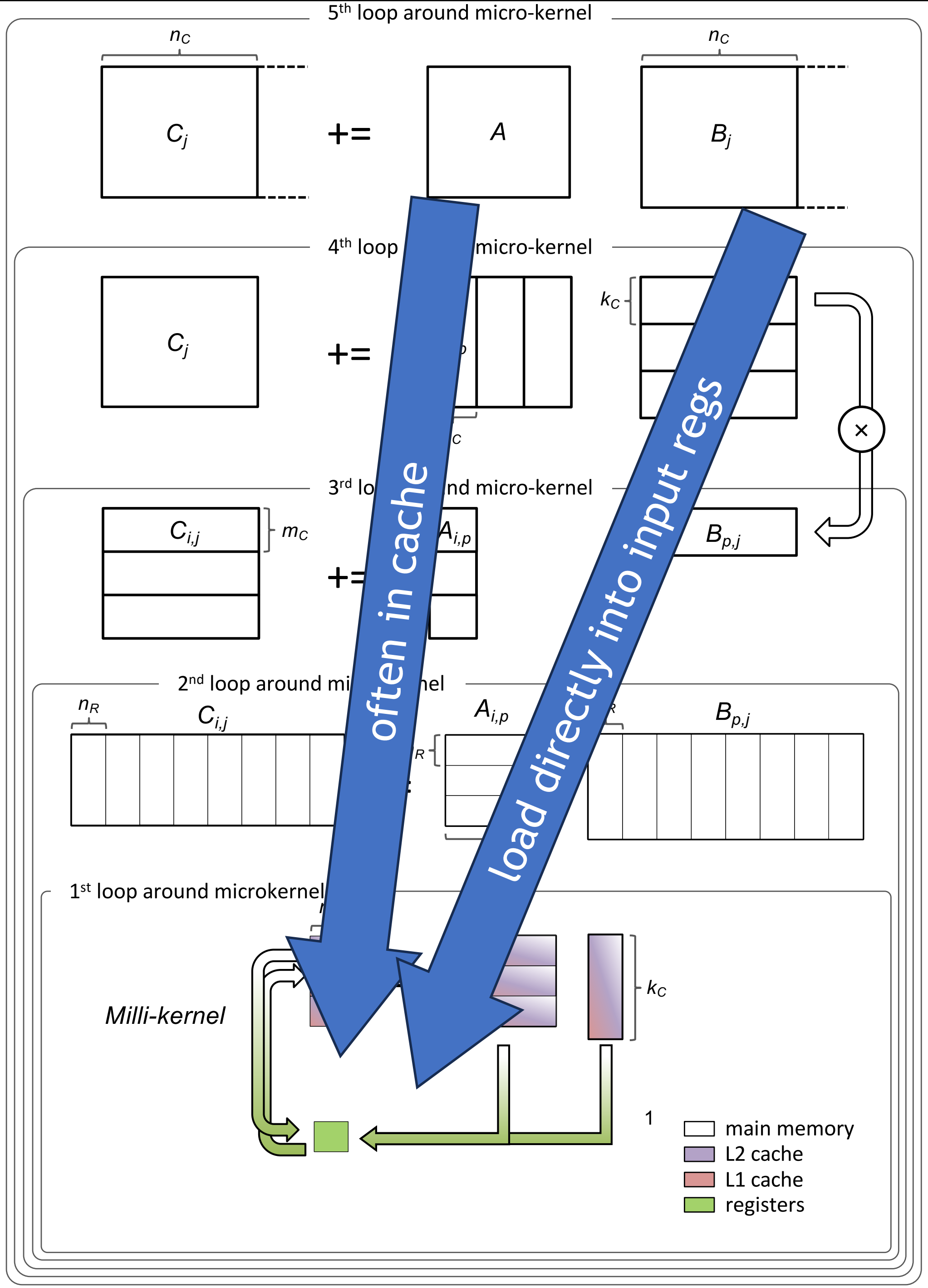
Cons:

- Not perfect for smaller matrices (e. g. $m = k = 2000, n = 20$).

Supporting small-ish matrices: **GemmSUP**

GemmSUP

Utilize the fact: small input A/B can fit directly into the cache



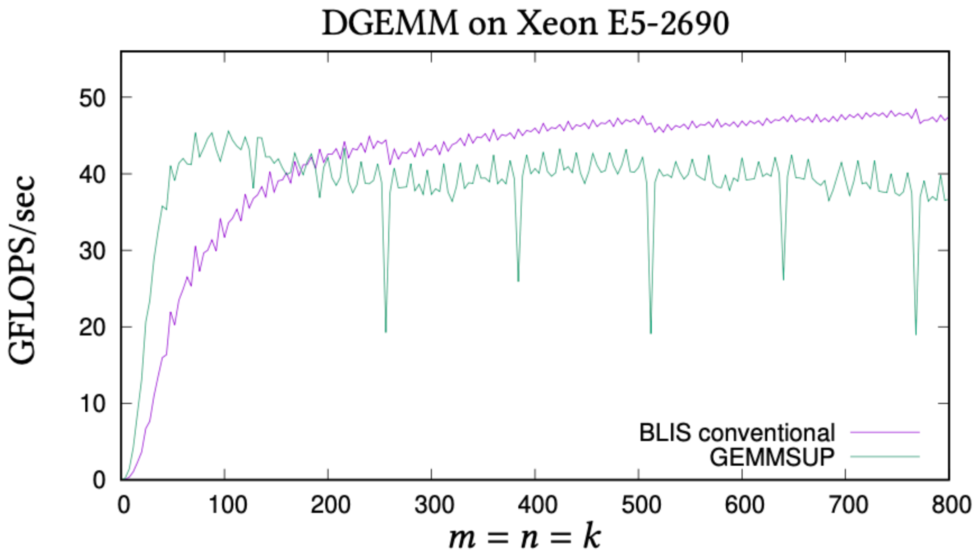
NB: All numbers shown here are warm cache results.
In real apps A & B can have one of them warm (e.g. as previous-step's result) and another cold (parameters)

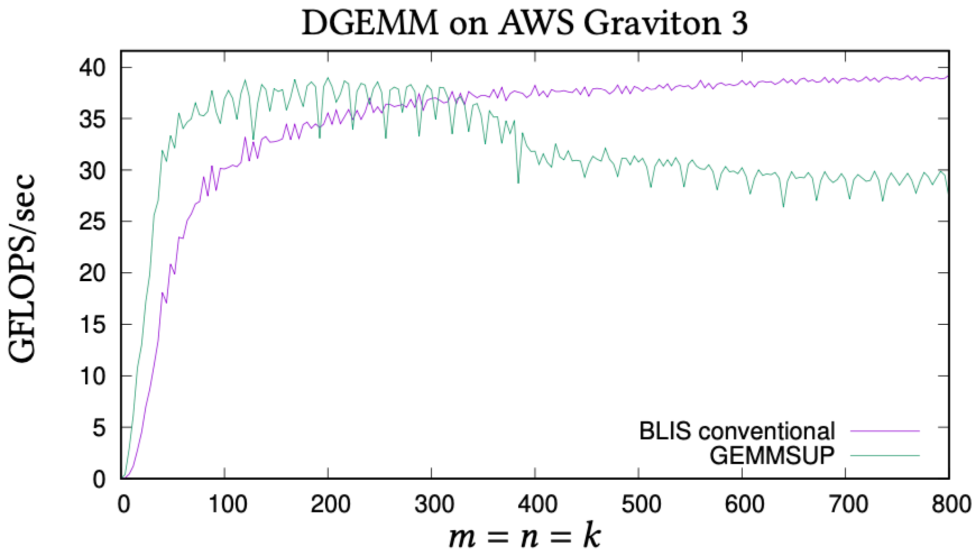
GemmSUP

- Slowdown due to cache competition.
- Still, cache can manage to store & prefetch *some* data
- Max performance ~80% theoretical peak.
- No extra ***store*** overhead: good for tall and skinny matrices.
    - *Good for LAPACK performance*

Conventional BLIS vs SUP

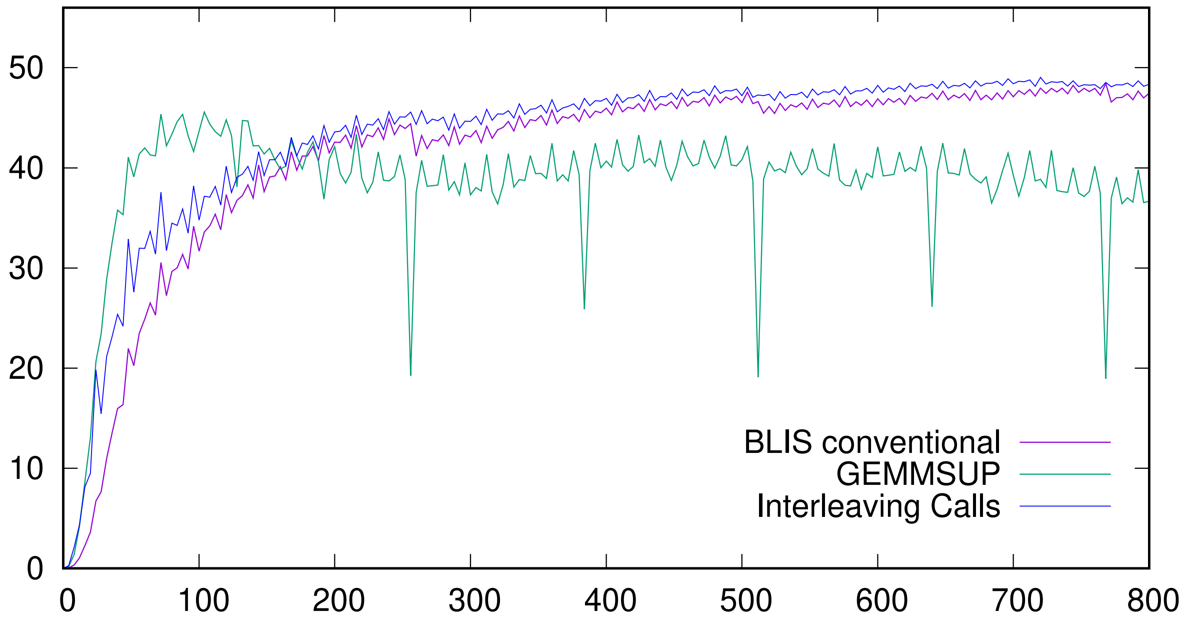
- Conventional BLIS works well for largg-ish, while SUP surges in small-ish
- Crossover points are hardware-dependent.
  - Caches differ there anyways.





Trying to fill the gap: Can we do packing *just before needed?*

- Interleaving packing with microkernel calls does produce a curve in between.
- Still fails to match SUP for small-ish cases.

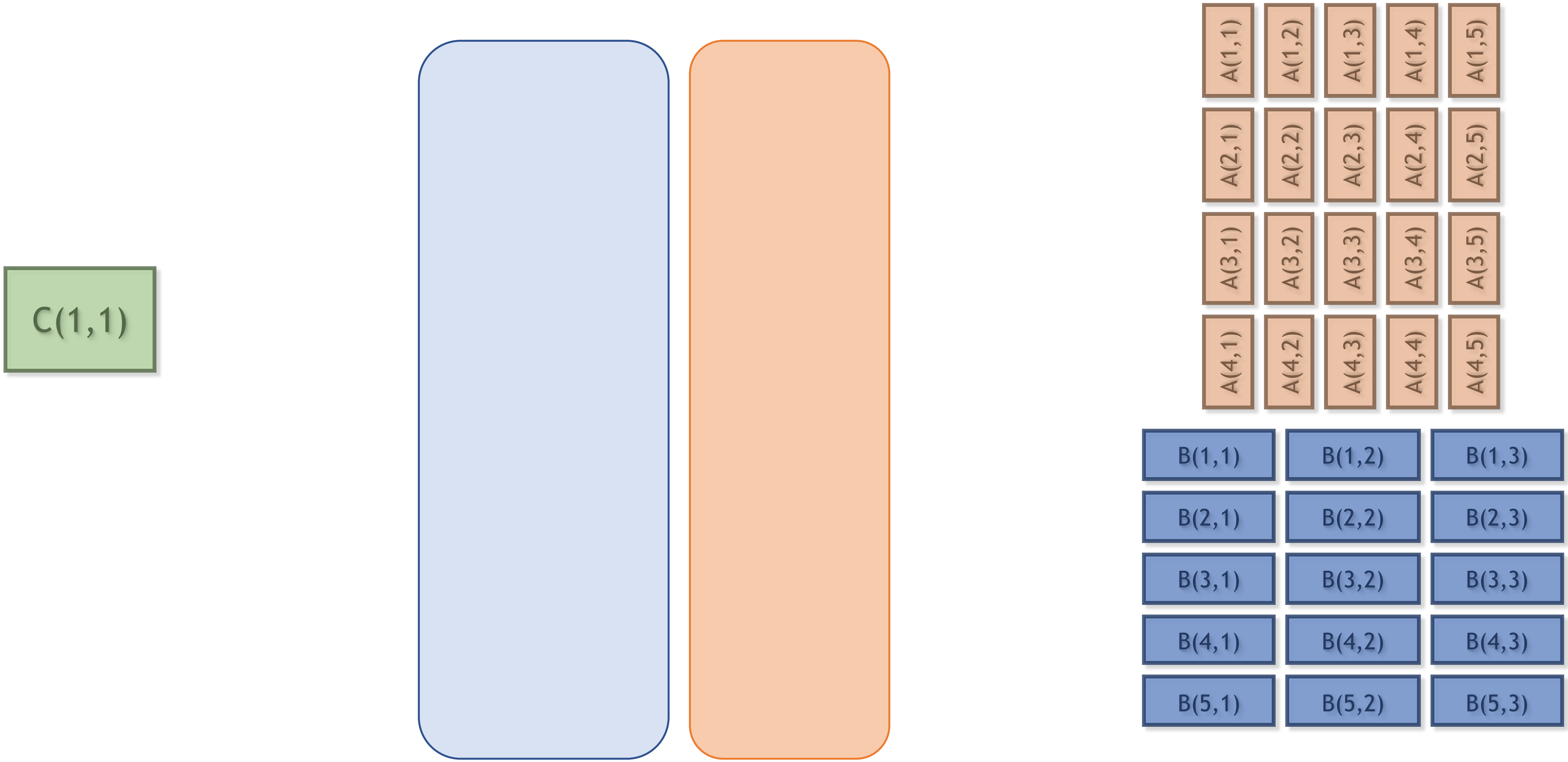


Unifying the two approaches by **F**using-**I**n the **P**acking

Registers
x86 or ARMv8

Packed Memory
usually present in cache

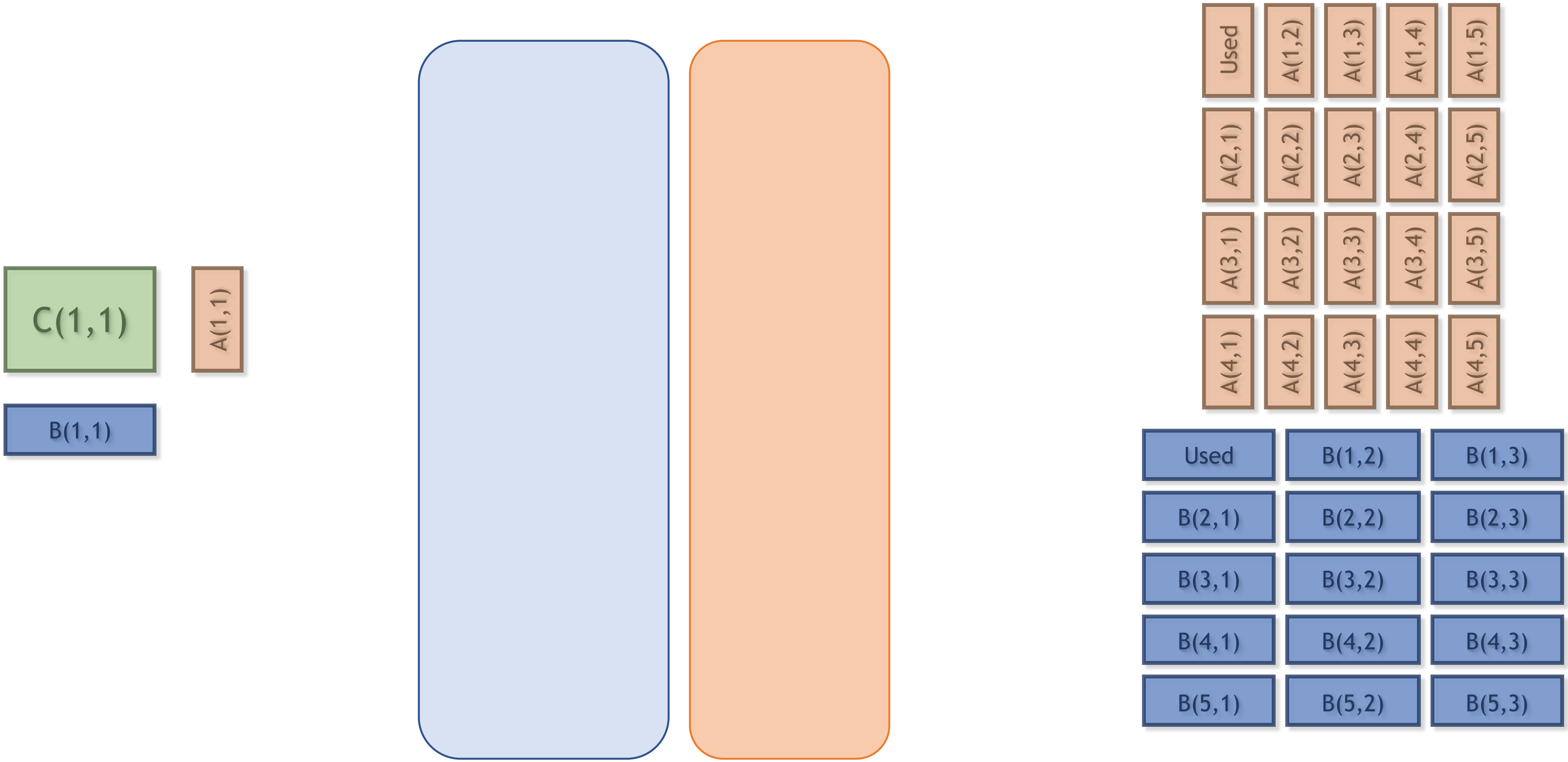
Unpacked Memory
usually not present in cache



Registers
x86 or ARMv8

Packed Memory
usually present in cache

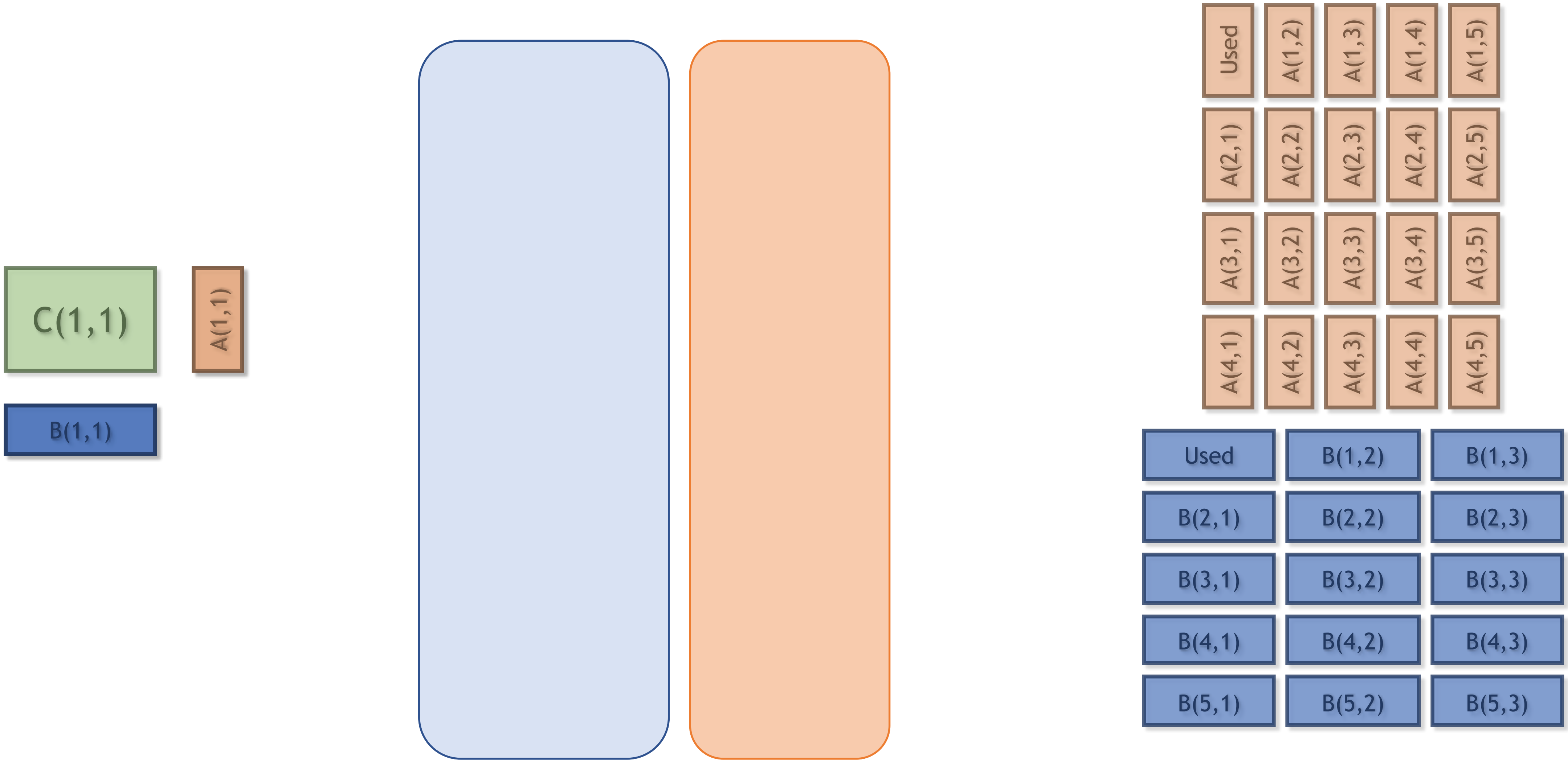
Unpacked Memory
usually not present in cache



Registers
x86 or ARMv8

Packed Memory
usually present in cache

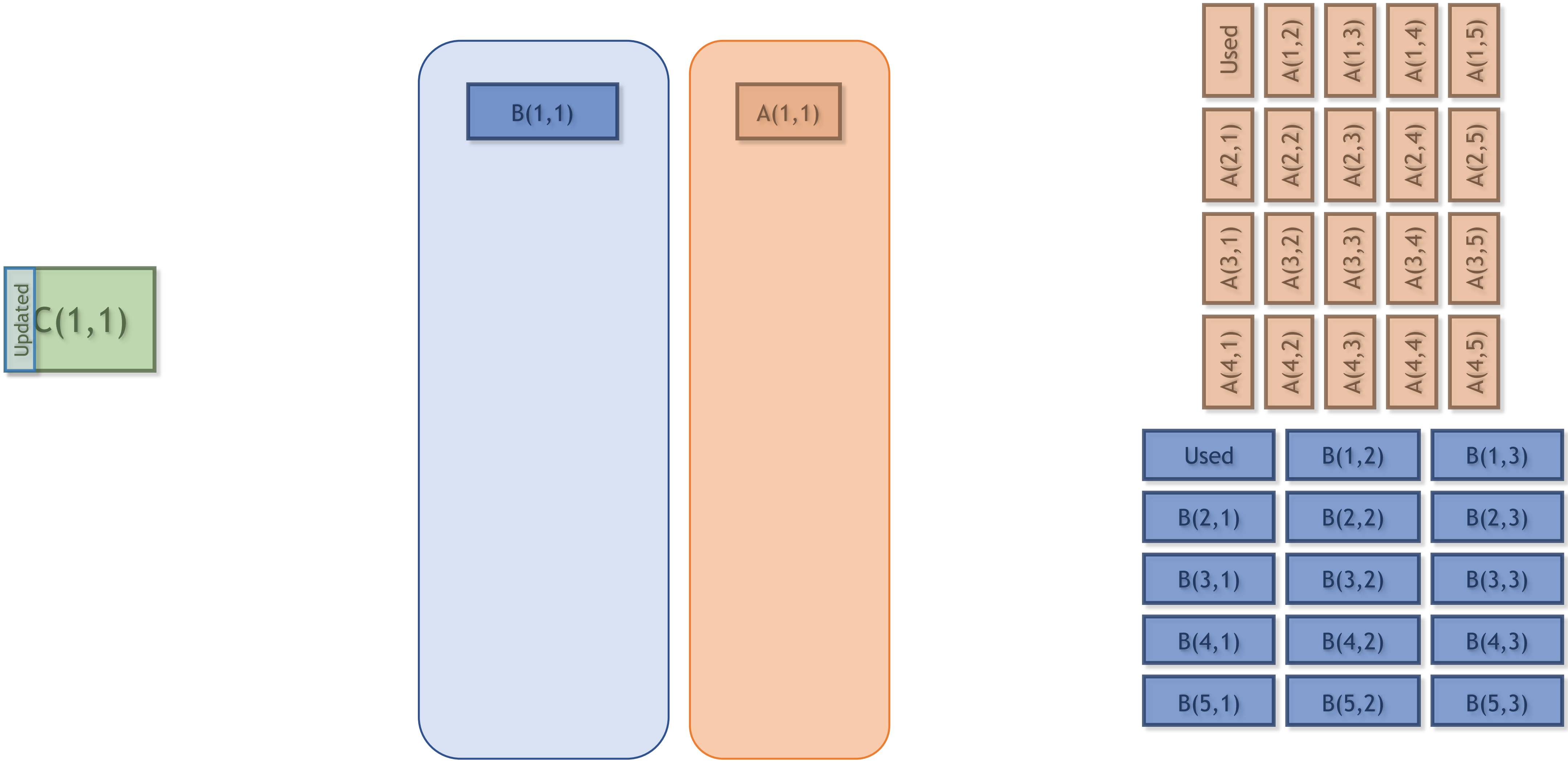
Unpacked Memory
usually not present in cache



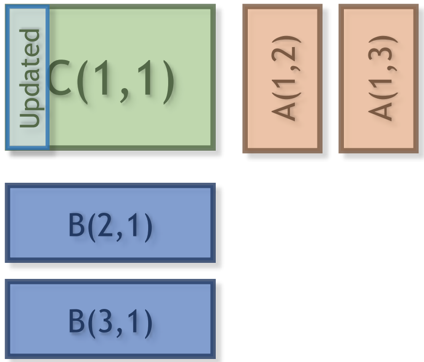
Registers
x86 or ARMv8

Packed Memory
usually present in cache

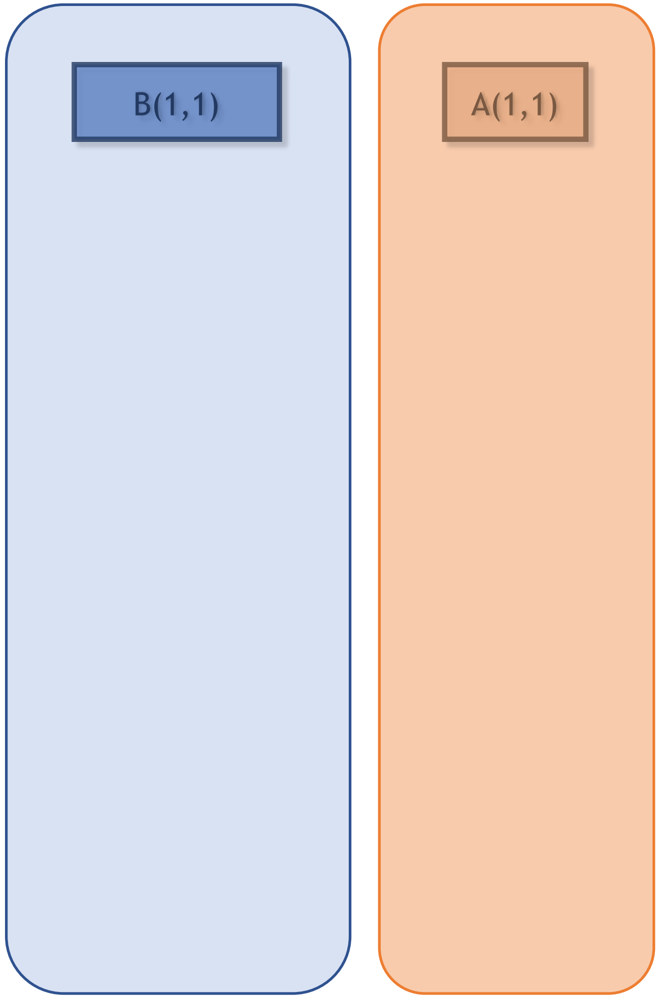
Unpacked Memory
usually not present in cache



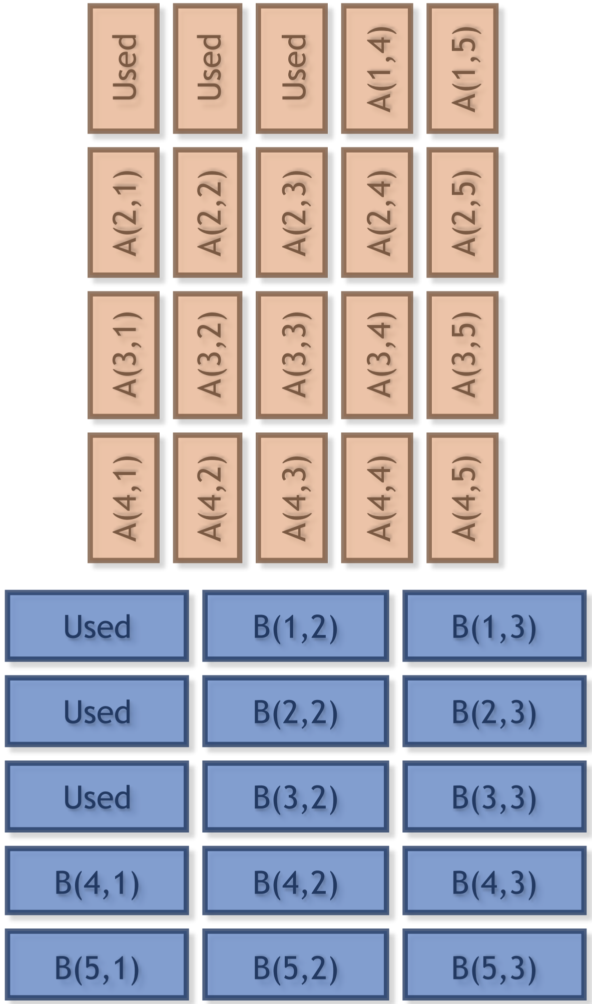
Registers
x86 or ARMv8



Packed Memory
usually present in cache



Unpacked Memory
usually not present in cache



Registers

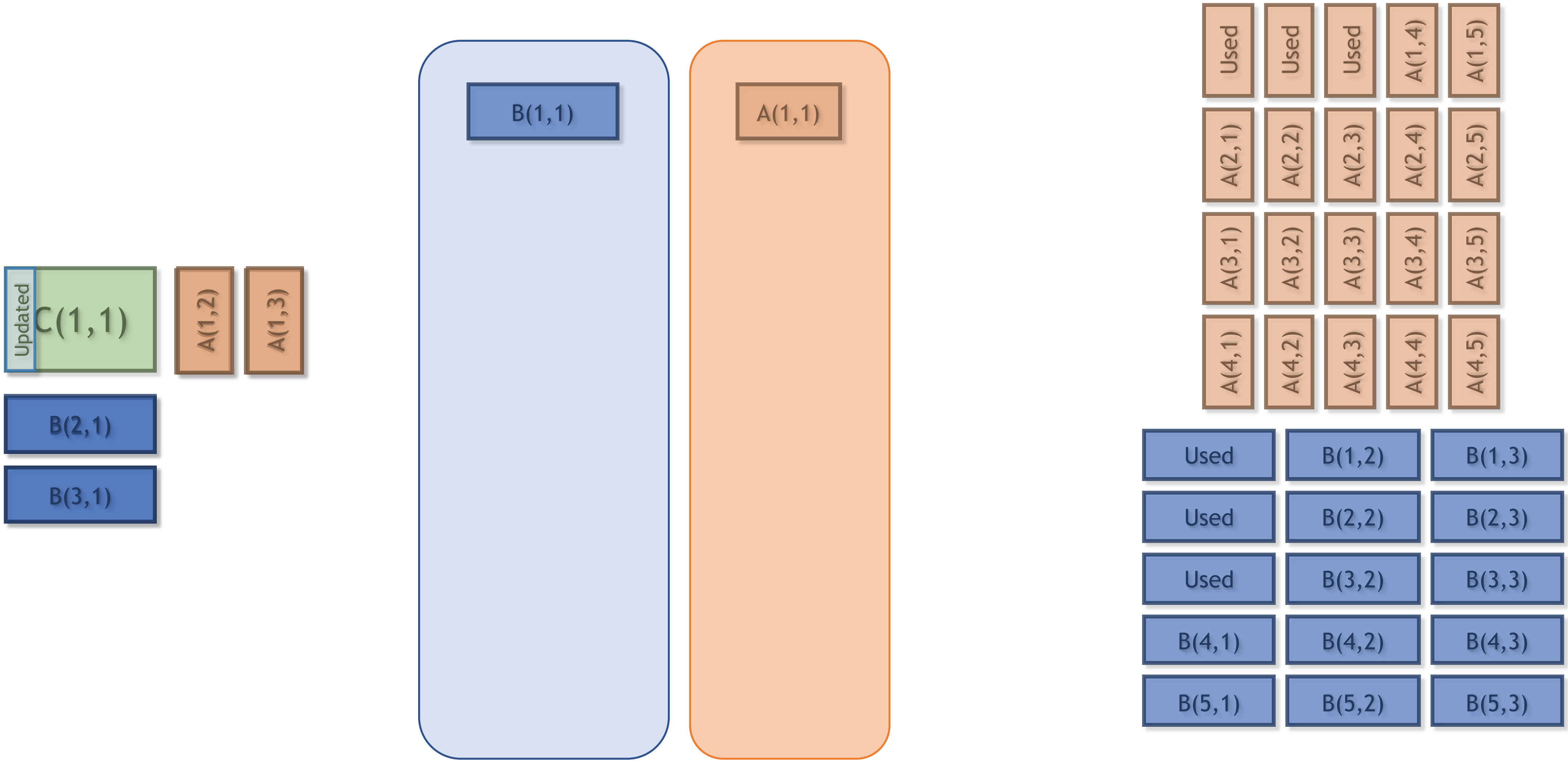
x86 or ARMv8

Packed Memory

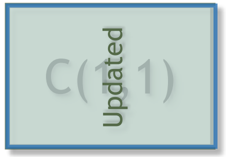
usually present in cache

Unpacked Memory

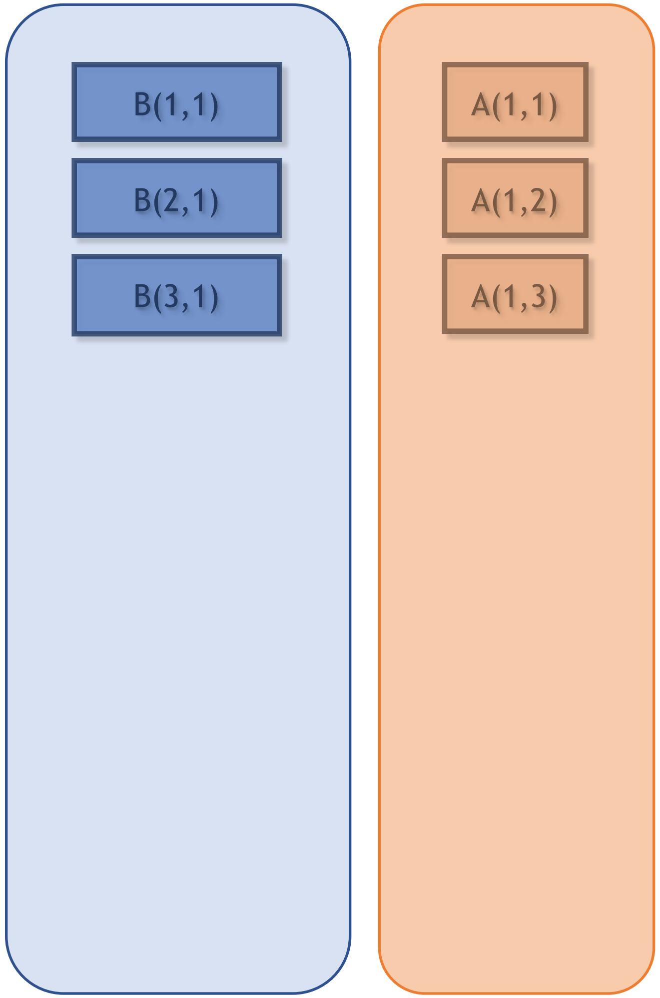
usually not present in cache



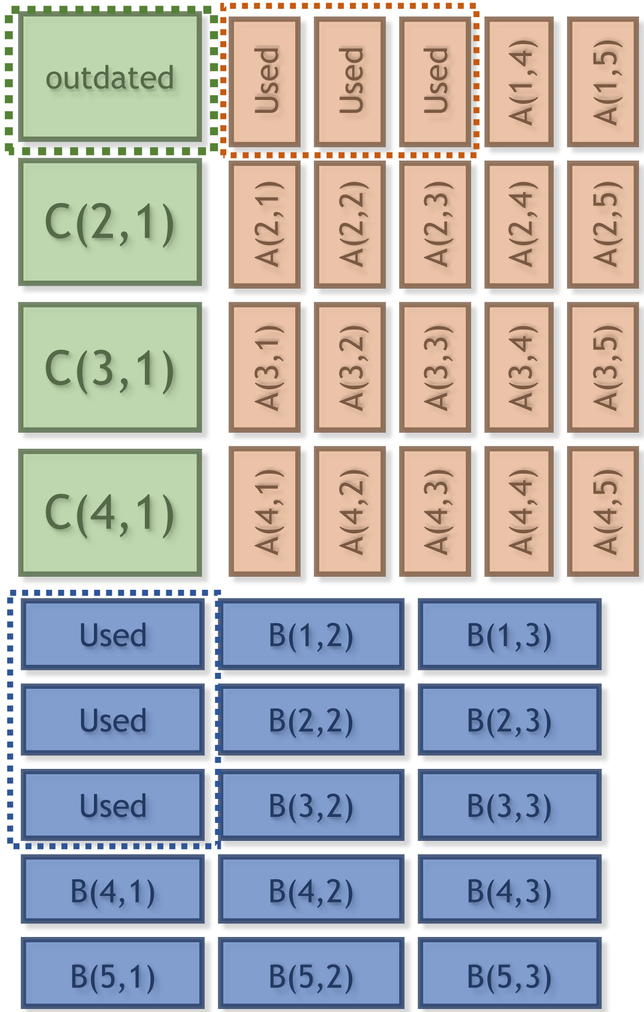
Registers
x86 or ARMv8



Packed Memory
usually present in cache



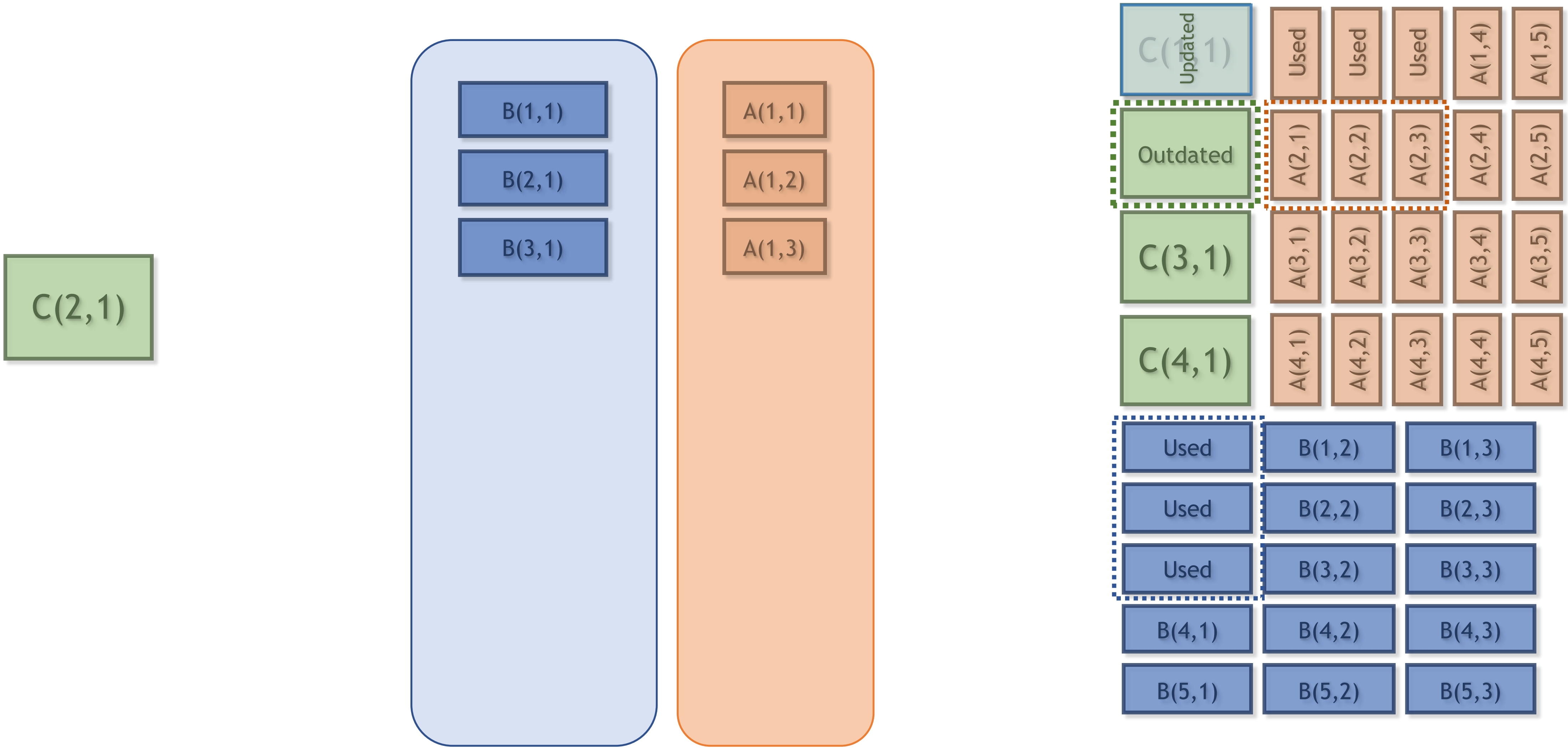
Unpacked Memory
usually not present in cache



Registers
x86 or ARMv8

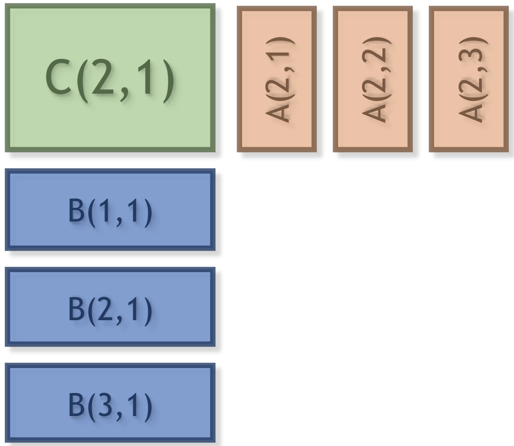
Packed Memory
usually present in cache

Unpacked Memory
usually not present in cache

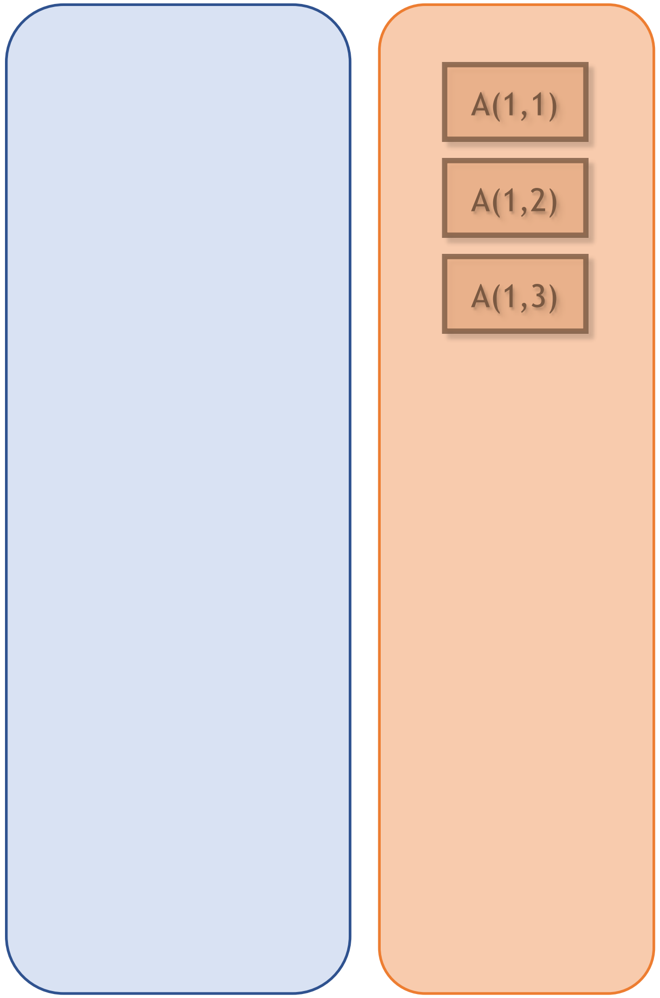


Registers
x86 or ARMv8

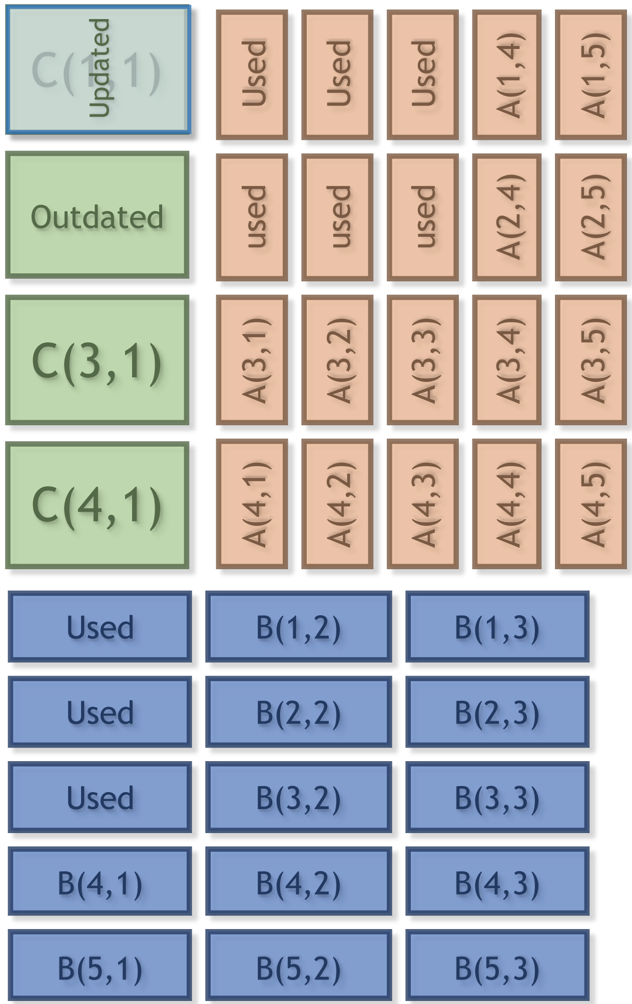
Now using *already-packed* memory for B *without* separate **packm** call!



Packed Memory
usually present in cache



Unpacked Memory
usually not present in cache

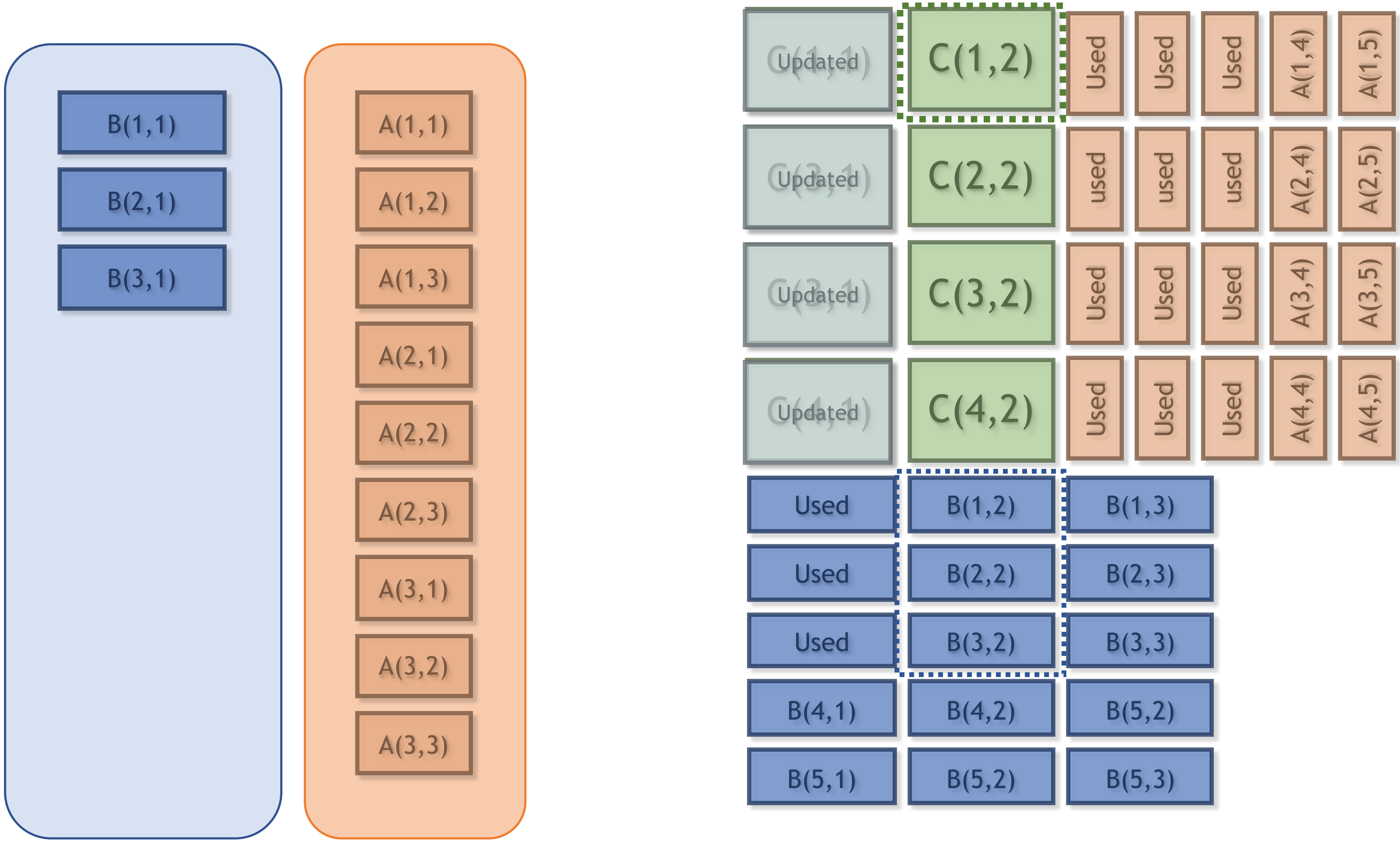


Registers
x86 or ARMv8

Packed Memory
usually present in cache

Unpacked Memory
usually not present in cache

All A-microtiles and one B-microtile is packed.



Registers

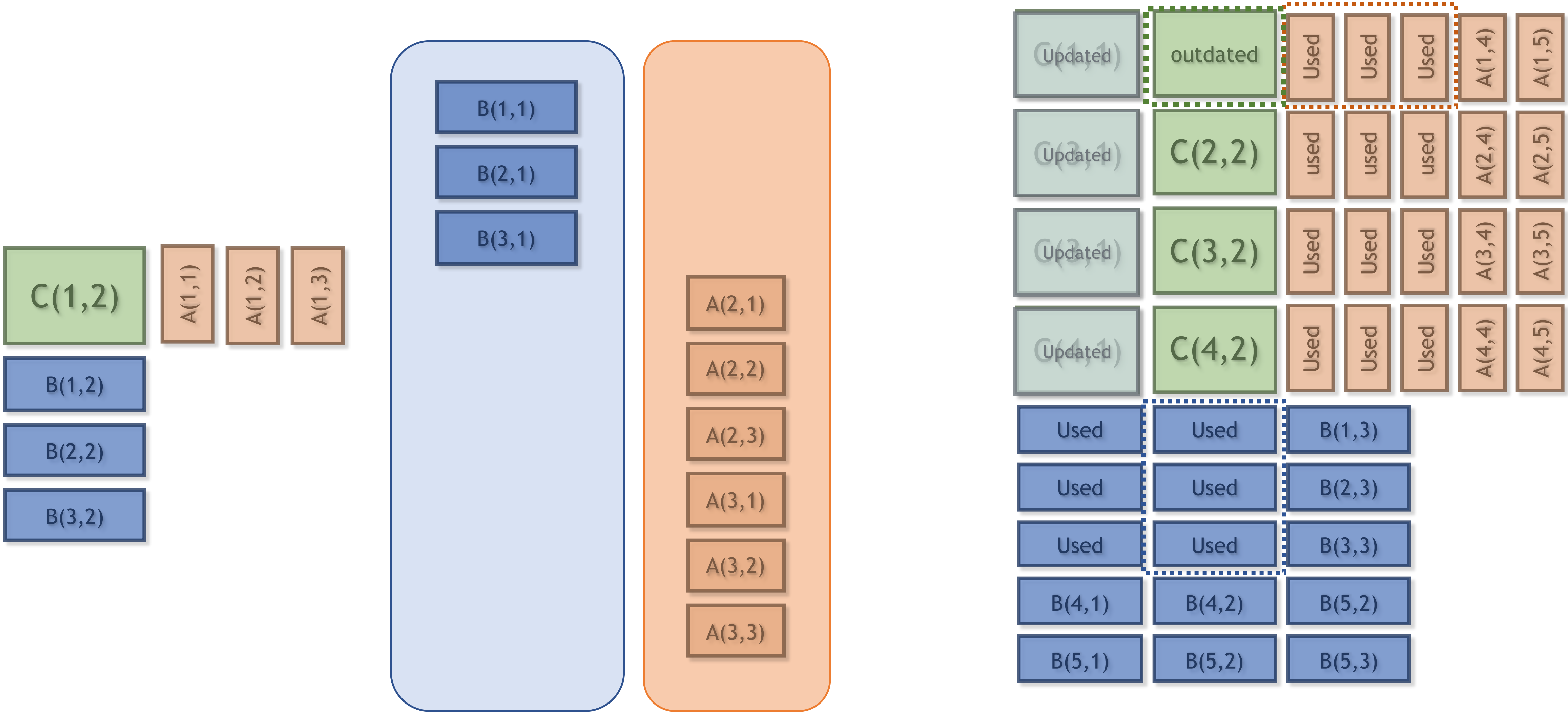
x86 or ARMv8

Packed Memory

usually present in cache

Unpacked Memory

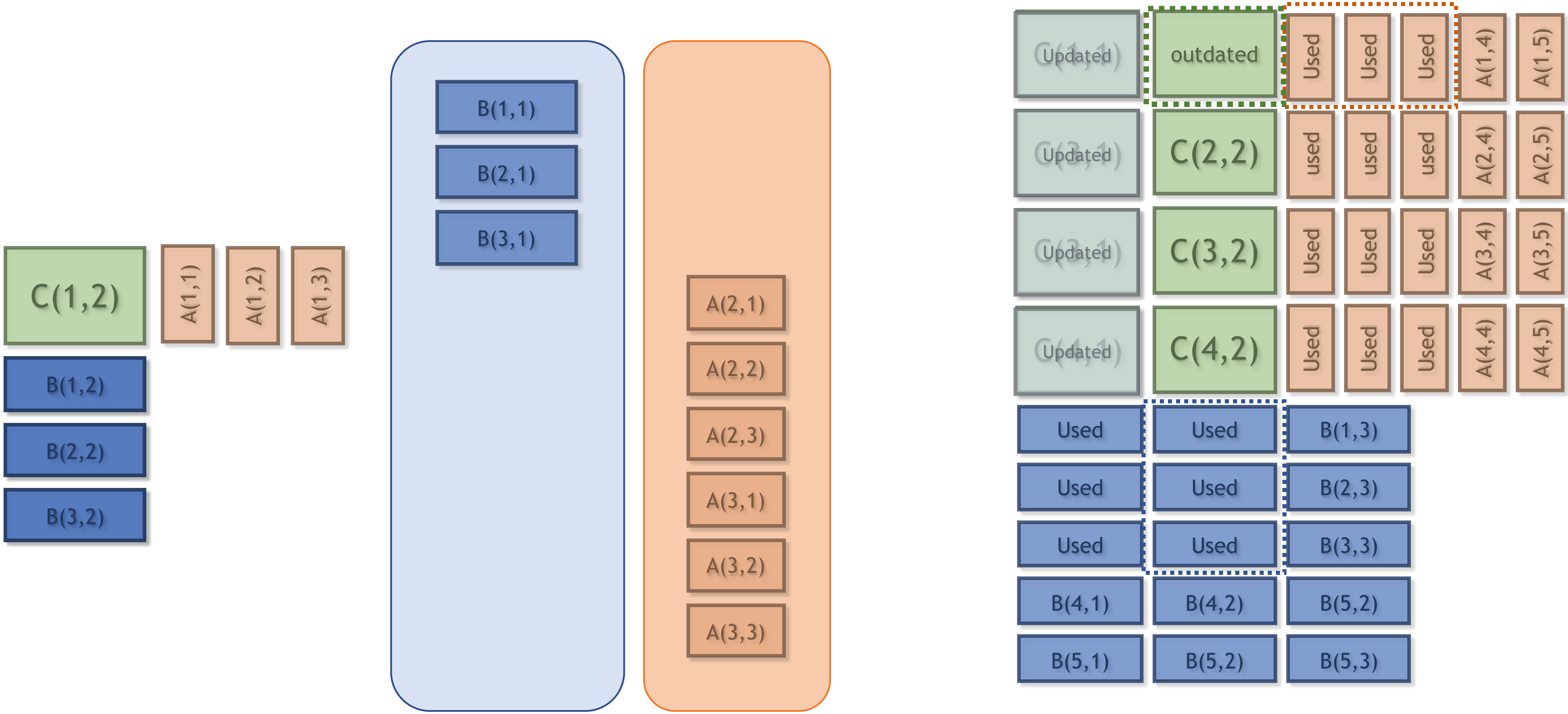
usually not present in cache



Registers
x86 or ARMv8

Packed Memory
usually present in cache

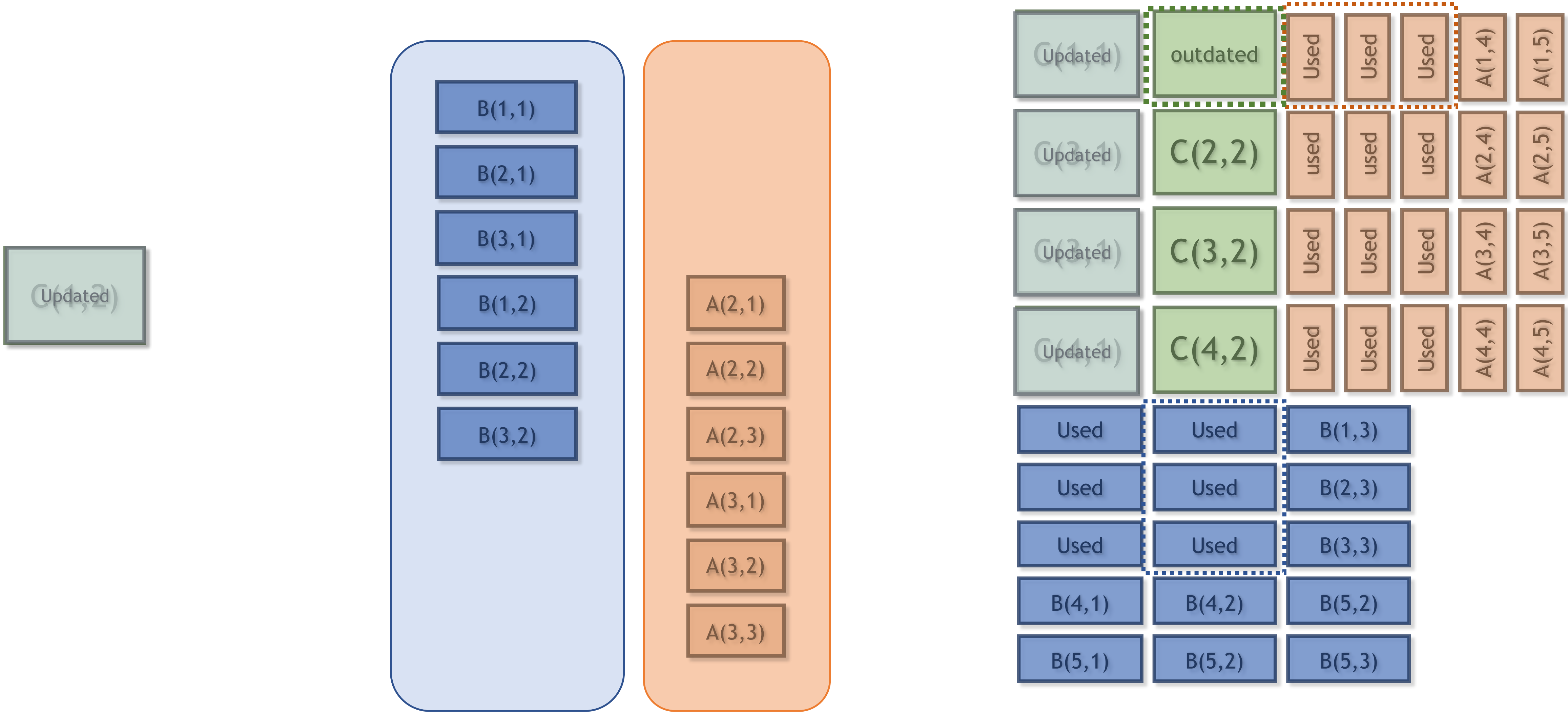
Unpacked Memory
usually not present in cache



Registers
x86 or ARMv8

Packed Memory
usually present in cache

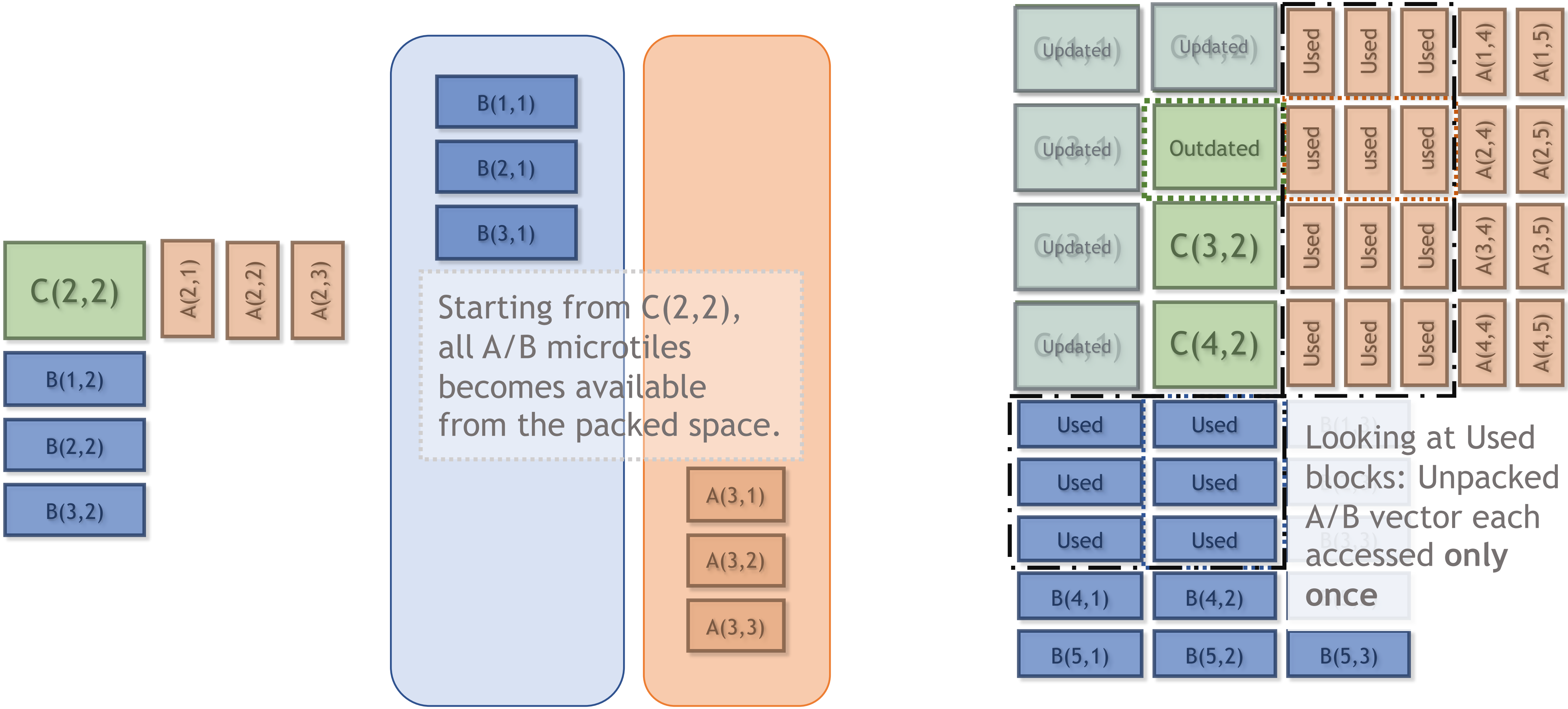
Unpacked Memory
usually not present in cache



Registers
x86 or ARMv8

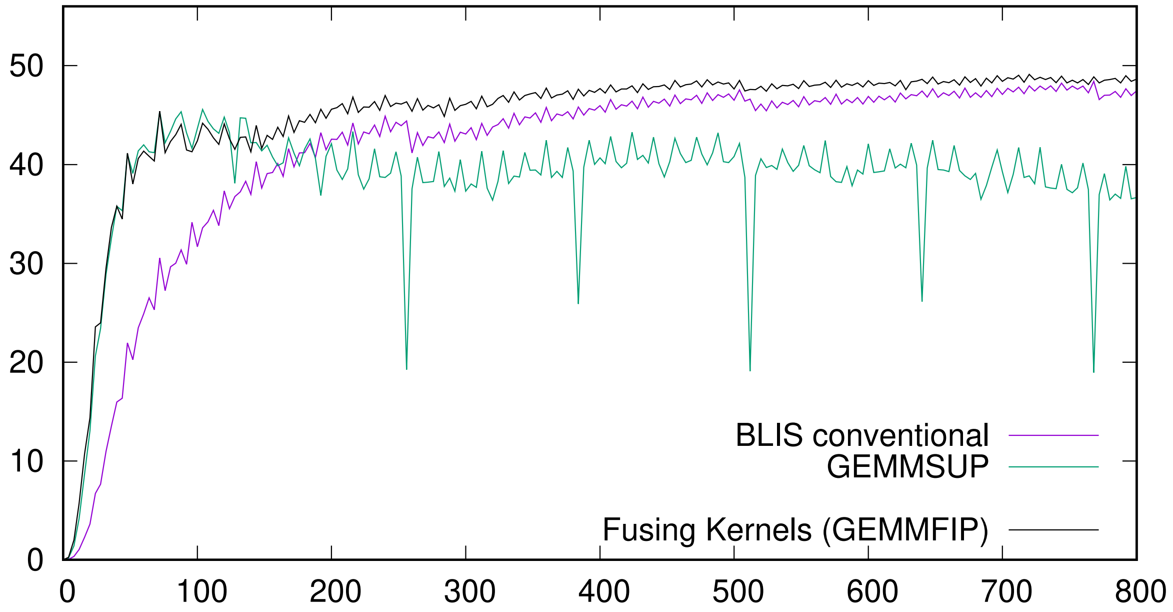
Packed Memory
usually present in cache

Unpacked Memory
usually not present in cache



Performance (single-threaded)

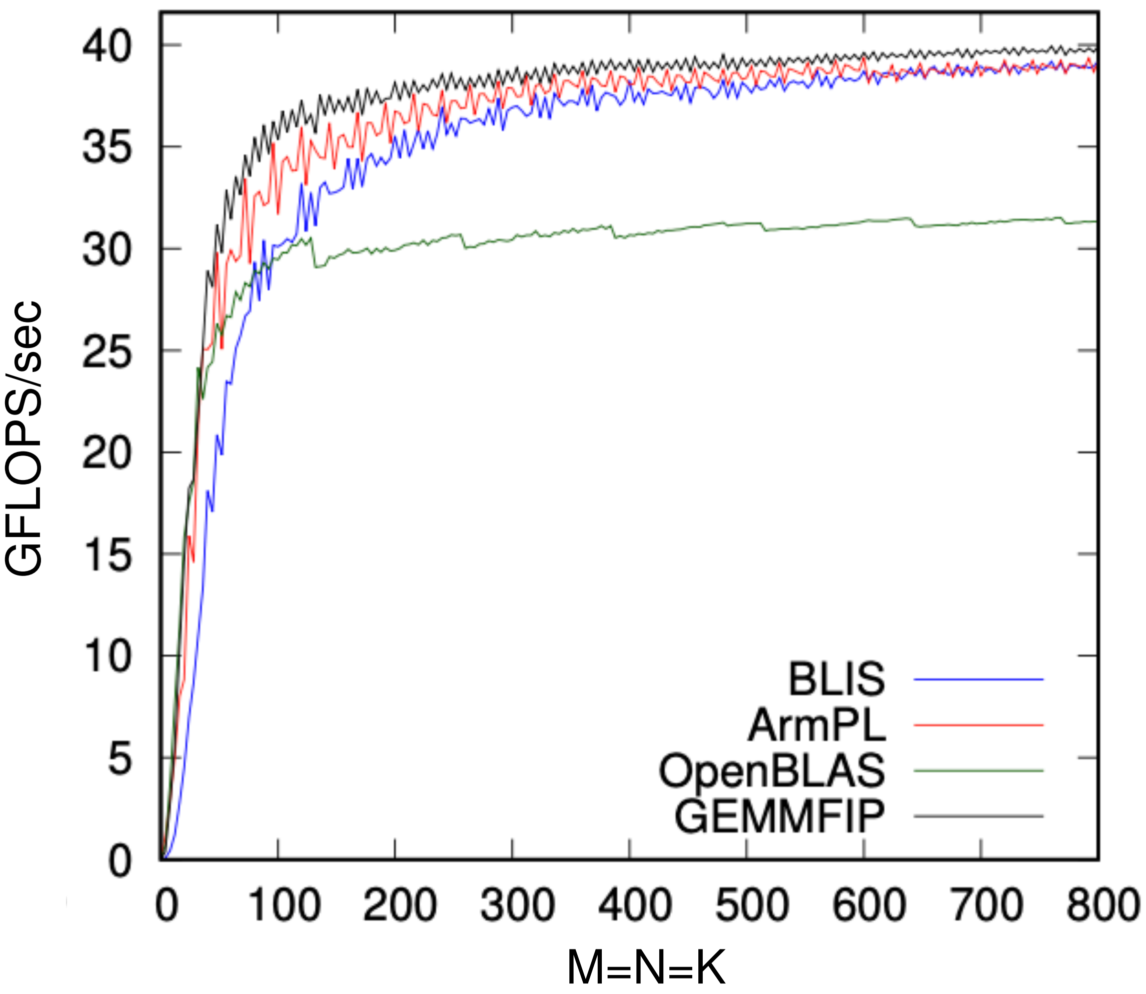
- Fused-in packing successfully bridged conventional GEMM with GEMMSUP where both ends are covered from above.
- Unifies conventional and unpacked BLIS kernels s. t. both code path and performance features better consistency.



Comparing w/ Other Impls. ARMv8-A + **SVE**

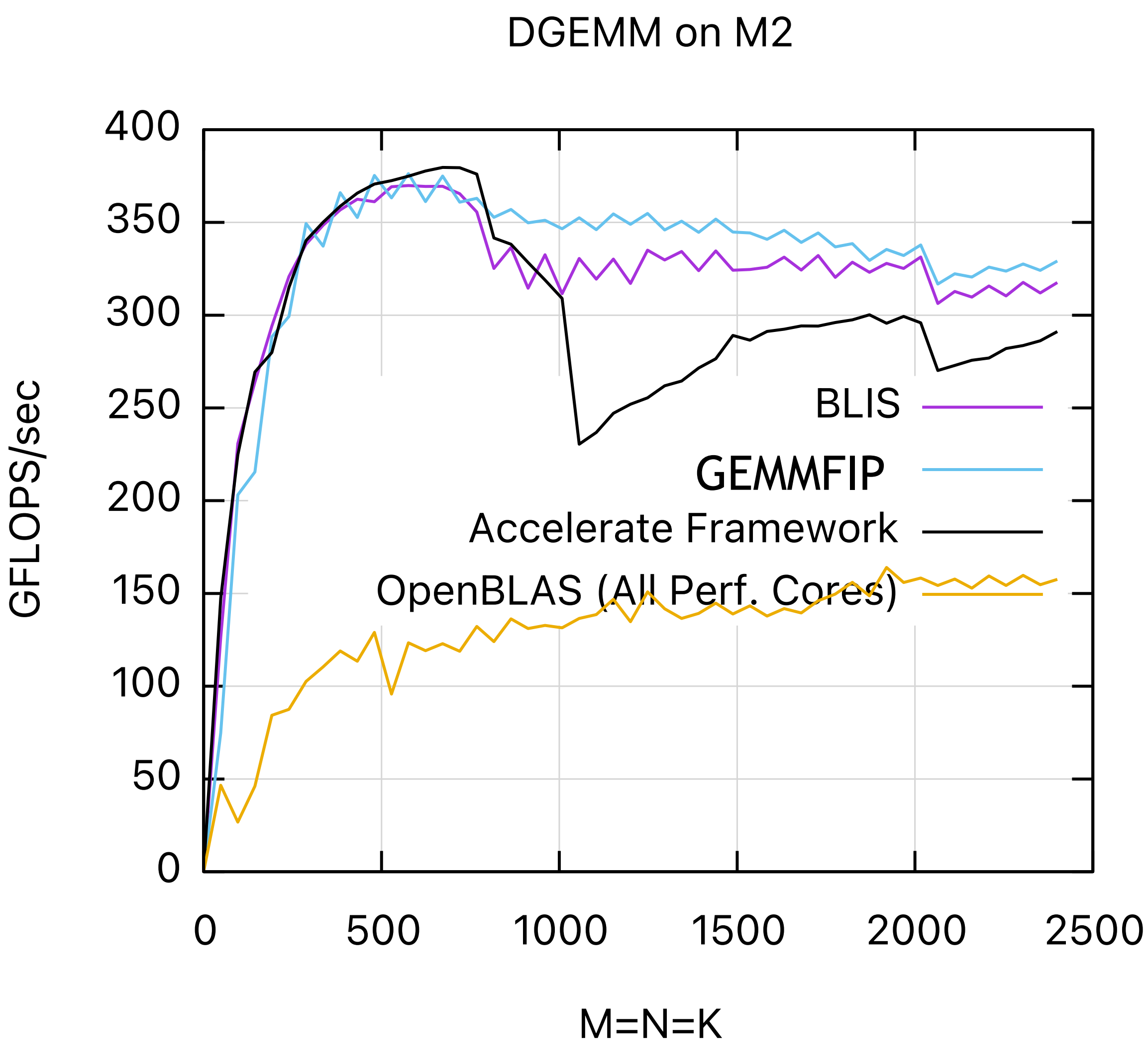
- OpenBLAS v0.3.21
  - v4 has supplied better kernels. We will try matching them in the future.
- Uniformly improvements over our previous BLIS-based implementation.

DGEMM on Amazon Graviton 3



Appendix: Perf. on Apple AMX2

- Special hardware that acts like a tensor core attached to CPUs.
  *On M2: **one** AMX per chip shared by all CPU cores.*
- On AMX2, instead of reducing *small-through middle-size* packing overhead, our new method improves that for large-size cases since AMX2's L2 cache can handle **very large** matrix tiles.



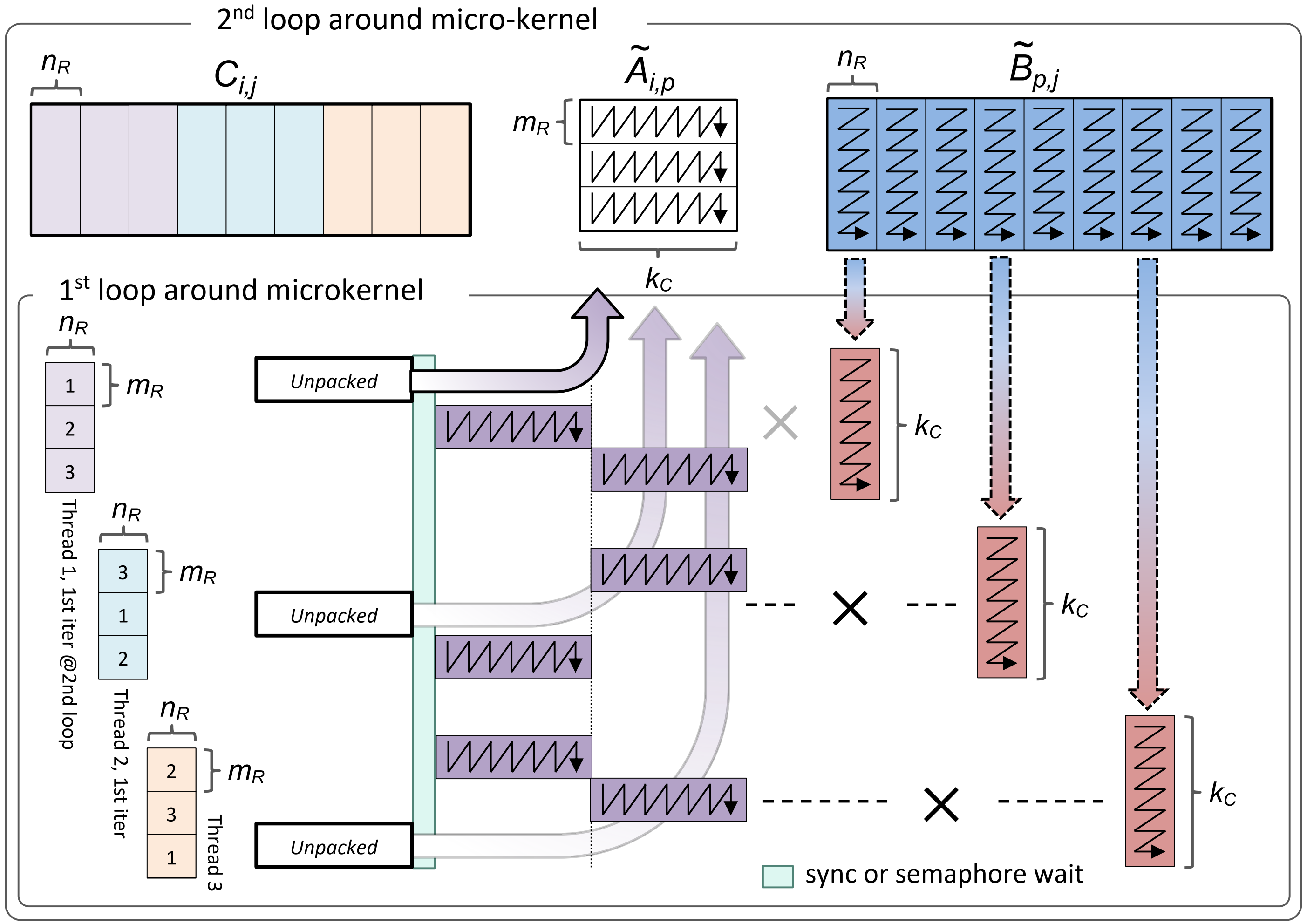
Threading

$\tilde{A}_{i,p}$:

- Threading involves multiple threads collaboratively updating the packing space $\tilde{A}_{i,p}$.
- Start using each other's packed tiles after finishing this collaborative packing:
  - Each unpacked tile only accessed **once**

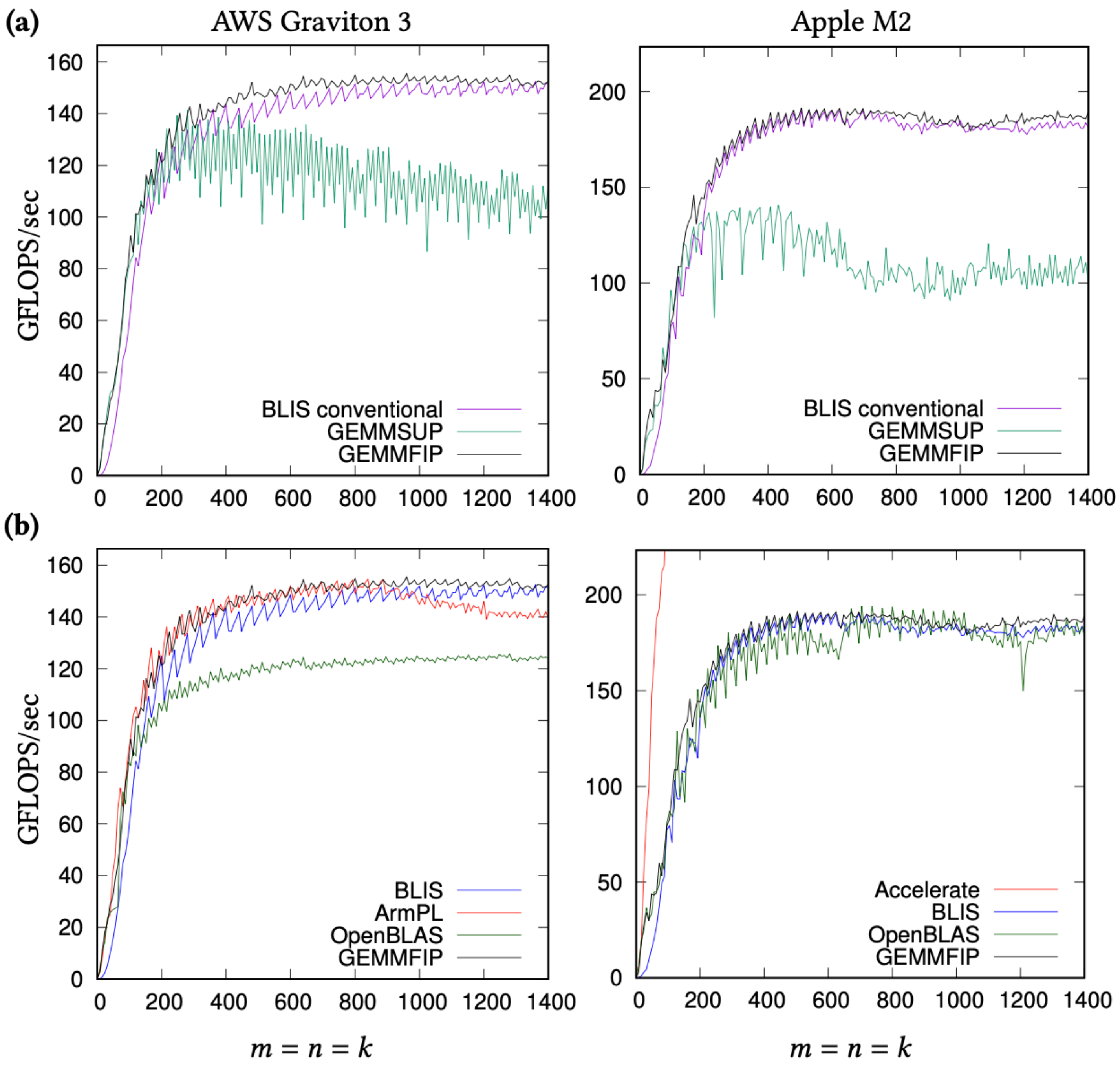
$B_{i,p}$:

- In thread-private L1. Each thread pack on its own.



Performance for Threading

- **Upper**: compares within BLIS
  Lower: compares against other impls.
- GEMMFIP's bridging effect extends into multithreaded regimes.



GEMMFIP

Pros

- **Method**: Fusing-in packing code into computational kernel.
- Identified and bridged the gap between packed (conventional) and unpacked (SUP) GEMM implementations.
- Produced a state-of-the-art level implementation.
- **To-dos**:
  - Cold-cache analysis.
- **Cons**:
  - Massive macros for writing fused-in packing!

Kernel Generating via Exo-lang:
A first attempt

What is Exo-lang?

- https://exo-lang.dev
- Domain-specific programming language for writing (for now,) CPU kernels.
- Loop transformation & register allocation is half-manually performed.
- Compiled down into C + intrinsics.
    - i.e. the compiler still holds the right to kill or spill those registers, but exo-lang transforms the code to hint the compiler to its best.

```
for k in seq(0, K):
    for i in seq(0, 6):
        for j in seq(0, 16):
            C[i, j] += A[i, k] * B[k, j]
```

exo.divide_loop
exo.stage_mem
exo.simplify

fn

```
...
    for i3 in seq(0, 8):
        C_reg[i0, i2, i3] = C[i0, i3 + 8 * i2]
...
    for jo in seq(0, 2):
        for ji in seq(0, 8):
            C_reg[i, jo, ji] +=
                A[i, k] * B[k, ji + 8 * jo]
...
    for i3 in seq(0, 8):
        C[i0, i3 + 8 * i2] = C_reg[i0, i2, i3]
```

Criteria of a *Nicely-gen'ed Kernel*

- A / B loading
- Cache carrying / evicting
- …
- K-loops unrolling

Std-C also handles

- During FMA loop, **no spill into the stack**
  - Stack spilling is, yet the hardest-to-manage behavior of a C compiler as it's an abstracted layer.

```
fmul    v4.2d, v4.2d, v30.d[0]
fmul    v3.2d, v3.2d, v30.d[0]
fadd    v11.2d, v11.2d, v28.2d
fmul    v30.2d, v2.2d, v7.d[0]
fmul    v28.2d, v2.2d, v5.d[0]
fmul    v2.2d, v2.2d, v10.d[0]
fadd    v12.2d, v12.2d, v8.2d
str     q11, [sp, 208]
ldr     q11, [sp, 224]
fadd    v23.2d, v23.2d, v30.2d
fadd    v22.2d, v22.2d, v29.2d
fadd    v11.2d, v11.2d, v27.2d
fmul    v27.2d, v1.2d, v7.d[0]
fmul    v7.2d, v0.2d, v7.d[0]
fadd    v21.2d, v21.2d, v28.2d
fadd    v24.2d, v24.2d, v2.2d
str     q11, [sp, 224]
ldr     q11, [sp, 288]
```

Compiler's LRU strategy thinks evicting v11/q11 into stack is only 1/32 regs, but it will stall the whole pipeline.

Managing Clang’s behavior

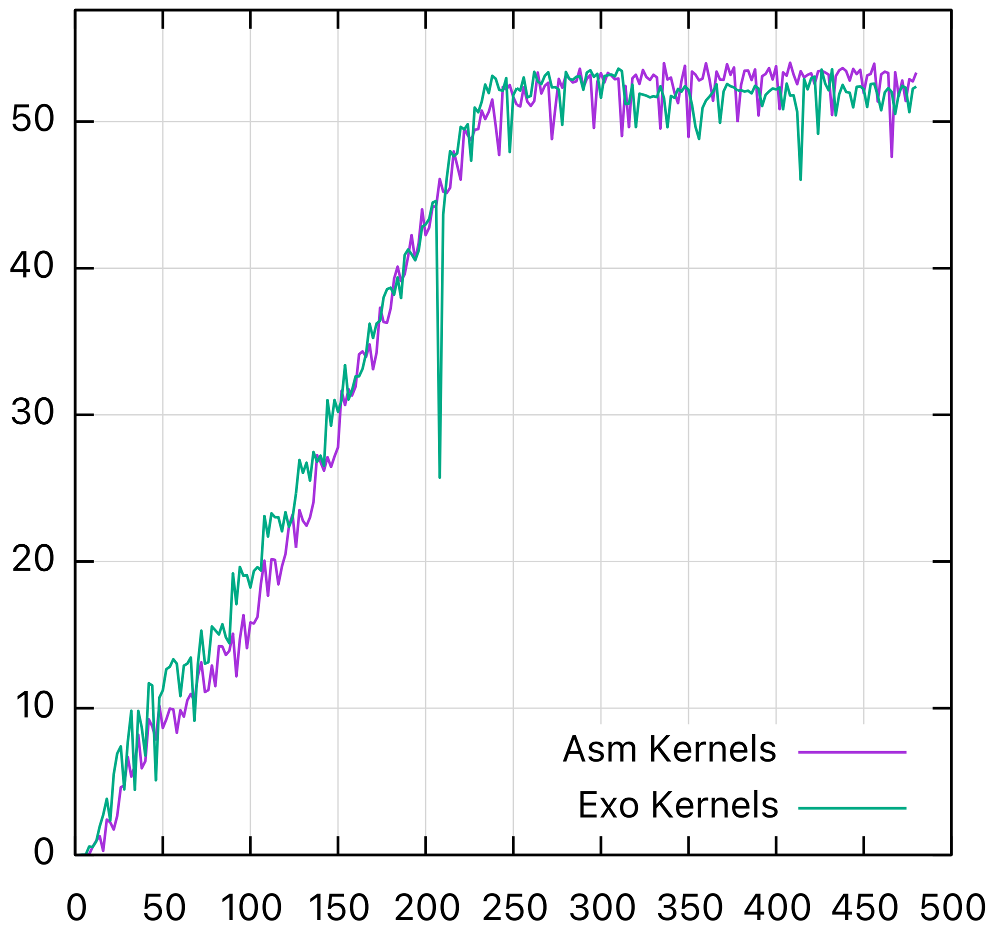
Before	Processing & reasons	After
float64x2_t C_reg[3][8]; for (i = ..) C_reg[i][j] += ..	“Array of float vectors” must be addressable. Const loop & const indices hardly got utilized. • exo.unroll_loop	float64x2_t C_reg[3][8]; C[0][0] += .. C[0][1] += ..
float64x2 C_reg[3][8];	Works when not k-unrolled. Spills at #unroll 4. • Use a new feature exo.unroll_buffer s.t. arrays with const indices get renamed into individual regs.	float64x2 C_reg_0_0; .. float64x2 C_reg_2_7;
Unroll k via exo.unroll-loop(tail=‘guard)	Spills. Clang failed to handle the flying register relations even it they are declared only once.	#pragma unroll 4 for (l=0; l<k; ++l) { .. }
…	…	…

A Glance in the Resulting Kernel

https://github.com/xrq-phys/blis/blob/gemmfip-exo/sandbox/gemmfip/exo_neon_d8x6/_8x6_nn_np.c

Comparing with Crafted Code

- On M2 + NEON, the generated kernel roughly matches crafted assembly.
- M2 contains many hardware-side optimizations. Could hide some latency.



Towards Full Integration

- SGEMMFIP / SHGEMMFIP reuses **exactly the same code**.

To-dos

- C’s load / store
- x86 kernels
  - AVX-family address alignment
  - Vector shuffling: Possible via current API, but requires some custom block defines.