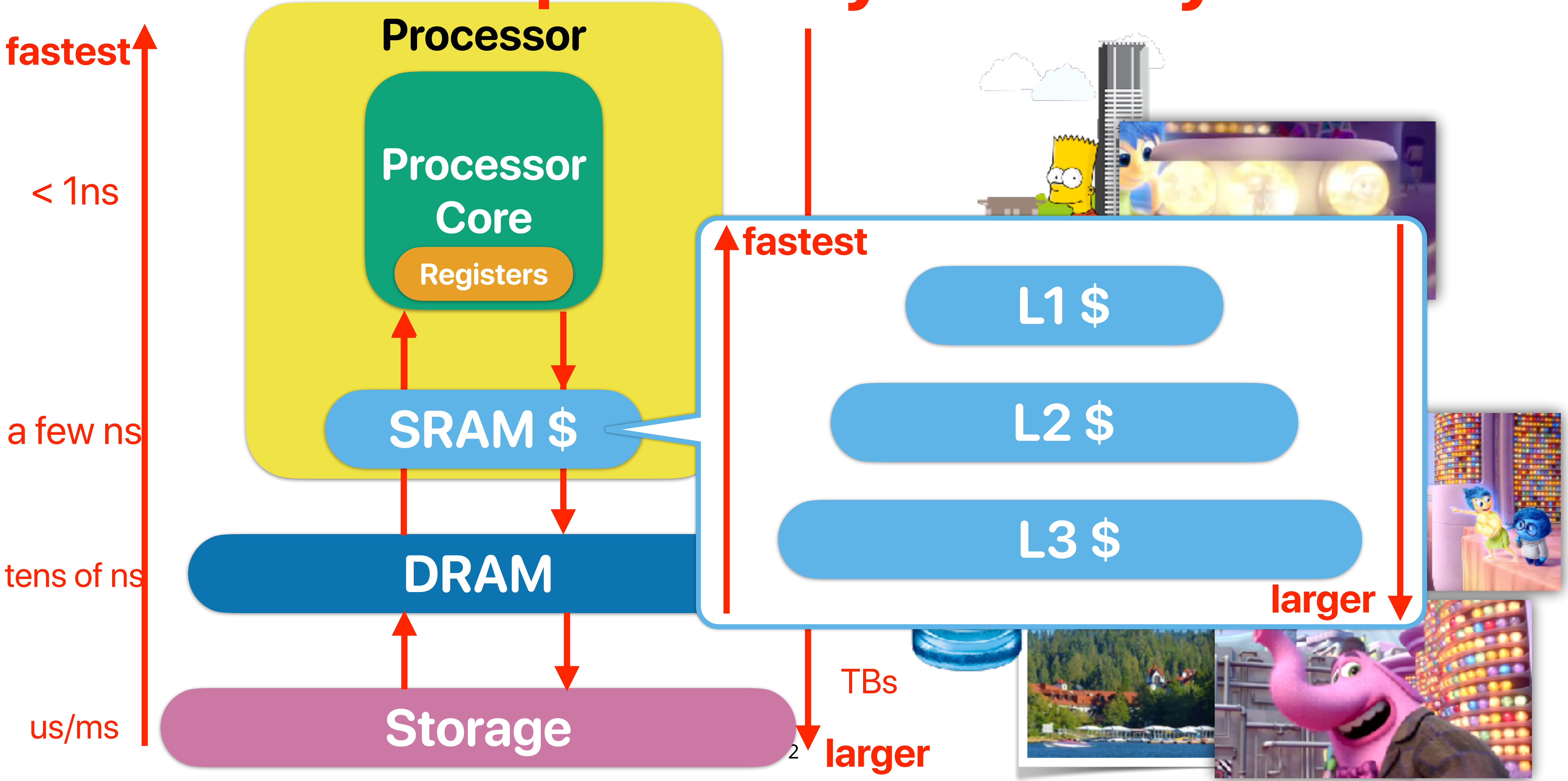
Lab 3: More Aspects on Memory (cont.) & Programming Assignment 2

Hung-Wei Tseng

Recap: Memory Hierarchy



Recap: Working set size

- The portion of memory that the program is currently using
- The total size of data blocks with "temporal locality"
- The total size of data blocks we have to visit before the next reuse of the current block

Recap: conflict misses v.s. capacity misses

- Conflict miss
  - If the miss occurs, and it's not the first time we access this block, and the working set size is smaller than the cache size
  - Solution — avoid the access pattern that creates a "stride" that creates conflicts among blocks
- Capacity miss
  - If the miss occurs, and it's not the first time we access this block, and the working set size is larger than the cache size
  - Solution — reduce the dataset size, reduce the working set size

Recap: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss
- Matrix transpose (a technique changes layout) — conflict misses
- Prefetching — compulsory misses

Q5: Loop renesting/loop interchange

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

Loop interchange

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

Q6: when is loop interchange/re-nesting useless?

What kind of miss does loop interchange eliminate?

Conflict misses

If conflict misses is not the main thing, loop interchange will become useless

Matrix Multiplication — let's consider "b"

```
for(i = 0; i < M; i++) {
  for(j = 0; j < K; j++) {
    for(k = 0; k < N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	0x0
b[5][0]	0x34000	0x34	0x0
b[6][0]	0x38000	0x38	0x0
b[7][0]	0x3C000	0x3C	0x0
b[8][0]	0x40000	0x40	0x0
b[9][0]	0x44000	0x44	0x0
b[10][0]	0x48000	0x48	0x0
b[11][0]	0x4C000	0x4C	0x0
b[12][0]	0x50000	0x50	0x0
b[13][0]	0x54000	0x54	0x0
b[14][0]	0x58000	0x58	0x0
b[15][0]	0x5C000	0x5C	0x0
b[16][0]	0x60000	0x60	0x0

- If the row dimension (N) of your matrix is 2048, each row element with the same column index is

  $$2048 \times 8 = 16384 = 0x4000$$

  away from each other

Each set can store only 12 blocks! So we will start to kick out b[0][0-7], b[1][0-7] ...

Now, when we work on c[0][1]

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	0x0
b[5][0]	0x34000	0x34	0x0
b[6][0]	0x38000	0x38	0x0
b[7][0]	0x3C000	0x3C	0x0
b[8][0]	0x40000	0x40	0x0
b[9][0]	0x44000	0x44	0x0
b[10][0]	0x48000	0x48	0x0
b[11][0]	0x4C000	0x4C	0x0
b[12][0]	0x50000	0x50	0x0
b[13][0]	0x54000	0x54	0x0
b[14][0]	0x58000	0x58	0x0
b[15][0]	0x5C000	0x5C	0x0

	Address	Tag	Index	
b[0][1]	0x20008	0x20	0x0	Conflict Miss

We're coming back to the block of b[0][0-7].

How big is the working set size?

$$16 \times 64 Bytes = 1024 Bytes < 48KB$$

Conflict misses! — not capacity misses

Each set can store only 12 blocks! So we will start to kick out b[0][0-7], b[1][0-7] ...

Now, when we work on c[0][1]

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	0x0
b[5][0]	0x34000	0x34	0x0
b[6][0]	0x38000	0x38	0x0
b[7][0]	0x3C000	0x3C	0x0
b[8][0]	0x40000	0x40	0x0
b[9][0]	0x44000	0x44	0x0
b[10][0]	0x48000	0x48	0x0
b[11][0]	0x4C000	0x4C	0x0
b[12][0]	0x50000	0x50	0x0
b[13][0]	0x54000	0x54	0x0
b[14][0]	0x58000	0x58	0x0
b[15][0]	0x5C000	0x5C	0x0

	Address	Tag	Index	
b[0][1]	0x20008	0x20	0x0	Conflict Miss
b[1][1]	0x24008	0x24	0x0	Conflict Miss
b[2][1]	0x28008	0x28	0x0	Conflict Miss
b[3][1]	0x2C008	0x2C	0x0	Conflict Miss
b[4][1]	0x30008	0x30	0x0	Conflict Miss
b[5][1]	0x34008	0x34	0x0	Conflict Miss
b[6][1]	0x38008	0x38	0x0	Conflict Miss
b[7][1]	0x3C008	0x3C	0x0	Conflict Miss
b[8][1]	0x40008	0x40	0x0	Conflict Miss
b[9][1]	0x44008	0x44	0x0	Conflict Miss
b[10][1]	0x48008	0x48	0x0	Conflict Miss
b[11][1]	0x4C008	0x4C	0x0	Conflict Miss
b[12][1]	0x50008	0x50	0x0	Conflict Miss
b[13][1]	0x54008	0x54	0x0	Conflict Miss
b[14][1]	0x58008	0x58	0x0	Conflict Miss
b[15][1]	0x5C008	0x5C	0x0	Conflict Miss

Each set can store only 12 blocks! So we will start to kick out b[0][0-7], b[1][0-7] ...

Now, when we work on c[0][1]

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	0x0
b[5][0]	0x34000	0x34	0x0
b[6][0]	0x38000	0x38	0x0
b[7][0]	0x3C000	0x3C	0x0
b[8][0]	0x40000	0x40	0x0
b[9][0]	0x44000	0x44	0x0
b[10][0]	0x48000	0x48	0x0
b[11][0]	0x4C000	0x4C	0x0

	Address	Tag	Index	
b[0][1]	0x20008	0x20	0x0	Hit!

If we can limit the access here —

Can you apply the same in Q6?

What value of `-arg1` should result in a very high (e.g., > 95%) hit rate, even with the `x` loop on the inside? Try to reason through the correct value before running any experiments.

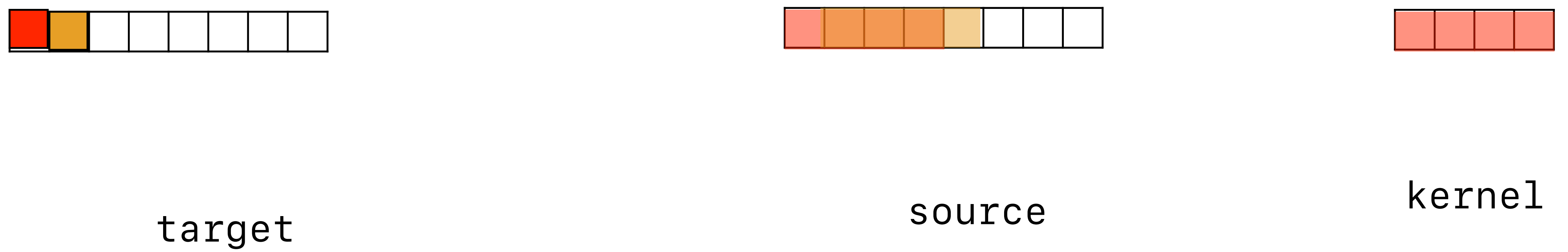
```
uint64_t* x_outside(uint64_t * data, uint64_t size, uint64_t
arg1) {
    uint64_t **matrix = to_2d_array(data,size,arg1);

    for(uint y = 0; y < arg1; y++) {
        for(uint x = 0; x < size/arg1; x++) {
            matrix[x][y] = x;
        }
    }

    return data;
}
```

Q7: 1-D convolution

```
void do_convolution(uint64_t *source,
                    uint64_t *kernel,
                    uint64_t *target,
                    uint64_t source_size,
                    uint64_t kernel_size, uint64_t tile_size) {
    uint64_t target_size = source_size - kernel_size;
    for(register uint64_t i = 0; i < target_size; i++) {
        for(register uint64_t j = 0; j < kernel_size; j++) {
            target[i] += source[i+j] * kernel[j];
        // target[i] = target[i] + source[i+j] * kernel[j];
     }
   }
}
```

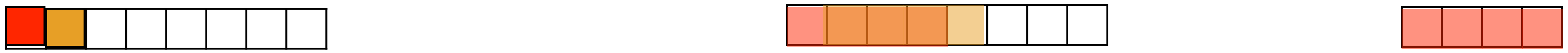


How many loads? How many stores?

Q8: working set of 1-D convolution

```
void do_convolution(uint64_t *source,
                    uint64_t *kernel,
                    uint64_t *target,
                    uint64_t source_size,
                    uint64_t kernel_size, uint64_t tile_size) {
    uint64_t target_size = source_size - kernel_size;
    for(register uint64_t i = 0; i < target_size; i++) {
        for(register uint64_t j = 0; j < kernel_size; j++) {
            target[i] += source[i+j] * kernel[j];
        }
    }
}
```

When `i` increments, we will come back to the same source block. How many blocks have visited before that?

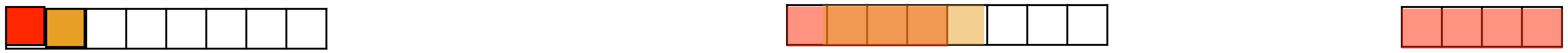


$$\frac{kernel\_size}{8} \text{ blocks} + \frac{kernel\_size}{8} \text{ blocks}$$

Q8: working set of 1-D convolution

```
void do_convolution(uint64_t *source,
                    uint64_t *kernel,
                    uint64_t *target,
                    uint64_t source_size,
                    uint64_t kernel_size, uint64_t tile_size) {
    uint64_t target_size = source_size - kernel_size;
    for(register uint64_t i = 0; i < target_size; i++) {
        for(register uint64_t j = 0; j < kernel_size; j++) {
            target[i] += source[i+j] * kernel[j];
        }
    }
}
```

The previously fetched source block may be kicked out!



What if I have a 24 KB kernel?

target source kernel

So we will consistently see misses in the first access of the block

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss
- Matrix transpose (a technique changes layout) — conflict misses
- Prefetching — compulsory misses
- Data alignments — capacity misses

The first idea

```
    for(int64_t i = 0; i < target_size; i++) {
        for(int64_t jj = 0; jj < kernel_size; jj += tile_size) { // We create a new loop variable jj that advanced one chunk at a time.
            for(int64_t j = jj; j < kernel_size && j < jj + tile_size; j++) { // We iterate over the chunk.  The more complicated termination
                // condition keeps ups from running off the end of the arry
                target[i] += source[i + j] * kernel[j];
            }
        }
    }
```

The tiling is not effective because...?

	size	arg1	le_si	function	IC	Cycles	CPI	CT	ET	L1_dcache_miss_rate	L1_dcache_accesses
1	131072	32768	64	convolution	22550921992	4099146202	0.181773	0.250845	1.028250	0.073631	9664503271
2	131072	32768	64	convolution_new_loop	29547281586	5336448357	0.180607	0.250840	1.338596	0.082891	9714951191

More ICs!

Slower!

Miss Rate is high!

Loop re-nesting/interchange

At least the tiling is now effective

```
    for(int32_t jj = 0; jj < kernel_size; jj += tile_size) {  // Move the jj chunk loop outside
        for(int32_t i = 0; i < target_size; i++) {
            for(int32_t j = jj; j < kernel_size && j < jj + tile_size; j++) {
                target[i] += source[i + j] * kernel[j];
            }
        }
    }
```

	size	arg1	le_si	function	IC	Cycles	CPI	CT	ET	L1_dcache_miss_rate	L1_dcache_accesses
1	131072	32768	64	convolution	22550921992	4099146202	0.181773	0.250845	1.028250	0.073631	9664503271
2	131072	32768	64	convolution_new_loop	29547281586	5336448357	0.180607	0.250840	1.338596	0.082891	9714951191
3	131072	32768	64	convolution_tiled	29395066175	4366858628	0.148558	0.250553	1.094130	0.001104	9714790824

More ICs still!

Slower!

Lower miss rate

Reduce the control overhead

apply "loop-unrolling" compiler optimization

```
extern "C"
uint64_t*  __attribute__((optimize("unroll-loops"))) convolution_tiled_unrolled(uint64_t * source,
uint64_t source_size,
                     uint64_t * kernel, uint64_t kernel_size,
                     uint64_t * target, uint64_t _target_size, int32_t tile_size) {

    uint64_t target_size = source_size - kernel_size;
    for(int32_t jj = 0; jj < kernel_size; jj += tile_size) {  // Move the jj chunk loop outside
        for(int32_t i = 0; i < target_size; i++) {
            for(int32_t j = jj; j < kernel_size && j < jj + tile_size; j++) {
                target[i] += source[i + j] * kernel[j];
            }
        }
    }
```

	size	arg1	le_si	function	IC	Cycles	CPI	CT	ET	L1_dcache_miss_rate	L1_dcache_accesses
1	131072	32768	64	convolution	22550921992	4099146202	0.181773	0.250845	1.028250	0.073631	9664503271
2	131072	32768	64	convolution_new_loop	29547281586	5336448357	0.180607	0.250840	1.338596	0.082891	9714951191
3	131072	32768	64	convolution_tiled	29395066175	4366858628	0.148558	0.250553	1.094130	0.001104	9714790824
4	131072	32768	64	convolution_tiled_unrolled	24915161674	3786971187	0.151995	0.251169	0.9488176	0.000942	9714659874

Faster now!

Can we do better?

```
extern "C"
uint64_t*  __attribute__((optimize("unroll-loops"))) convolution_tiled_unrolled(uint64_t * source,
uint64_t source_size,
                     uint64_t * kernel, uint64_t kernel_size,
                     uint64_t * target, uint64_t _target_size, int32_t tile_size) {

    uint64_t target_size = source_size - kernel_size;
    for(int32_t jj = 0; jj < kernel_size; jj += tile_size) {  // Move the jj chunk loop outside
        for(int32_t i = 0; i < target_size; i++) {
            for(int32_t j = jj; j < kernel_size && j < jj + tile_size; j++) {
                target[i] += source[i + j] * kernel[j];
            }
        }
    }
    return target;
}
```

We have to do two checks every time!

Only check `jj+tile_size` once

```
if (jj + tile_size > kernel_size) {
    for(int32_t j = jj; j < kernel_size; j++) {
        target[i] += source[i + j] * kernel[j];
    }
} else {
    for(int32_t j = jj; j < jj + tile_size; j++) {
        target[i] += source[i + j] * kernel[j];
    }
}
```

Further reduce the control/branch overhead

```
uint64_t* __attribute__((optimize("unroll-loops"))) convolution_tiled_split(uint64_t * source, uint64_t source_size,
                  uint64_t * kernel, uint64_t kernel_size,
                  uint64_t * target, uint64_t _target_size, int32_t tile_size) {
    uint64_t target_size = source_size - kernel_size;
    int32_t real_tile_size = tile_size/8 * 8; // this clears the low 3 bits.  Check the assembly!
    assert(tile_size>=8);

    for(int32_t jj = 0; jj < kernel_size; jj += real_tile_size) {  // Move the jj chunk loop outside
        for(int32_t i = 0; i < target_size; i++) {
            if (jj + real_tile_size > kernel_size) {
                for(int32_t j = jj; j < kernel_size; j++) {
                    target[i] += source[i + j] * kernel[j];
                }
            } else {
                for(int32_t j = jj; j < jj + real_tile_size; j++) {
                    target[i] += source[i + j] * kernel[j];
                }
            }
        }
    }
    return target;
}
```

	size	arg1	e_s	function	IC	Cycles	CPI	CT	ET	L1_dcache_miss_rate	L1_dcache_accesses
1	131072	32768	64	convolution	22550921992	4099146202	0.181773	0.250845	1.028250	0.073631	9664503271
2	131072	32768	64	convolution_new_loop	29547281586	5336448357	0.180607	0.250840	1.338596	0.082891	9714951191
3	131072	32768	64	convolution_tiled	29395066175	4366858628	0.148558	0.250553	1.094130	0.001104	9714790824
4	131072	32768	64	convolution_tiled_unrolled	24915161674	3786971187	0.151995	0.251169	0.9488176	0.000942	9714659874
5	131072	32768	64	convolution_tiled_split	16359042414	3252245650	0.198804	0.250875	0.815907	0.001299	9714595552

IC drops a lot!

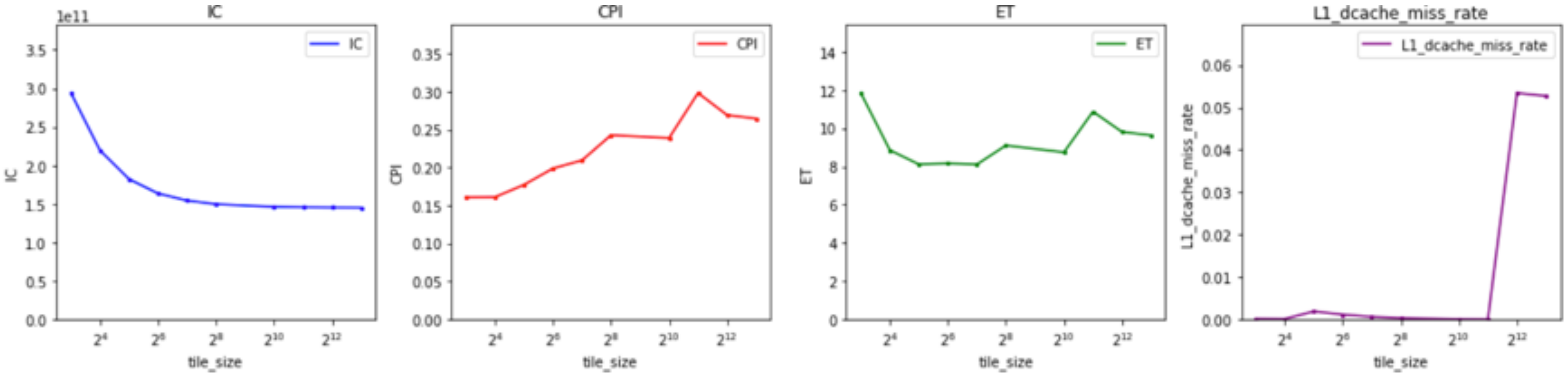
Significant improvement

Selecting the right tile size!

	size	rep	arg1	tile_size	function	IC	Cycles	CPI	CT	ET	_dcache_miss_ra
1	131072	1	32768	8	convolution_tiled_split	293954351884	47268351256	0.160802	0.250537	11.842483	0.000183
2	131072	1	32768	16	convolution_tiled_split	219458852083	35342730839	0.161045	0.250544	8.854903	0.000140
3	131072	1	32768	32	convolution_tiled_split	182212269246	32299455989	0.177263	0.251226	8.114475	0.001899
4	131072	1	32768	64	convolution_tiled_split	163589599579	32526336632	0.198829	0.250926	8.162029	0.001160
5	131072	1	32768	128	convolution_tiled_split	154278148344	32315039644	0.209460	0.250937	8.109041	0.000628
6	131072	1	32768	256	convolution_tiled_split	149623856171	36303704199	0.242633	0.250720	9.102061	0.000320
7	131072	1	32768	1024	convolution_tiled_split	146131592161	34898560301	0.238816	0.250537	8.743379	0.000084
8	131072	1	32768	2048	convolution_tiled_split	145552912507	43387111422	0.298085	0.250538	10.870136	0.000061
9	131072	1	32768	4096	convolution_tiled_split	145260546766	39083417418	0.269057	0.250995	9.809729	0.053440
10	131072	1	32768	8192	convolution_tiled_split	145114375483	38371619605	0.264423	0.251070	9.633955	0.052744

Sweet spot

Trade-offs between IC and miss rate!



Programming assignment

Markov Chain

Current State

A	B	C
1	0	0

×

Transition Matrix

	A	B	C
A	0.7	0.2	0.1
B	0.4	0.4	0.2
C	0.2	0.4	0.4

=

After the first transition

A	B	C
0.7	0.2	0.1

Capacity misses if states and matrix are large!

Conflict misses if we traverse by column

After 1st transition

A	B	C
0.7	0.2	0.1

×

Transition Matrix-1

	A	B	C
A	0.7	0.2	0.1
B	0.4	0.4	0.2
C	0.2	0.4	0.4

=

After the 2nd transition

A	B	C
0.59	0.26	0.13

Pretty much vector-matrix!

What kinds of misses are we supposed to see?

Remember what we've tried so far?

- Matrix multiplications
  - Tiling
  - Matrix transpose
- 1-D convolution
  - Tiling
  - Loop interchange
  - Unrolling
  - Reducing branches

```
for(uint64_t i = 0; i < days; i++) {
    std::memset(result, 0, scale*sizeof(double));
    for(uint64_t j = 0; j < scale; j++) {
        for(uint64_t k = 0; k < scale; k++) {
            result[k] += intermediate_result[j]*transition_matrix[k][j];
        }
    }
    std::memcpy(intermediate_result, result, scale*sizeof(double));
}
```

What do you think could be effective?

Leaderboard for now

Rank	Submission Name	markov_solution_c 8192 128 speedup
1	haley	20.55
2	ProfUsagi	**18.49**
3	Erick Fentress	17.51
4	Asher	17.32
5	Adi	16.3
6	NQ	16.21
7	😊	14.16
8	il	10.93
9	Wen	7.01
10	BATMANNN	5.88

Computer
Science &
Engineering

142L

つづく