

Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses

Changwan Hong, Wenlei Bao, Albert Cohen, Sriram
Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello,
J. Ramanujam, P. Sadayappan

David Miller
COP6622: Topics in Compiler Optimizations
July 13, 2018

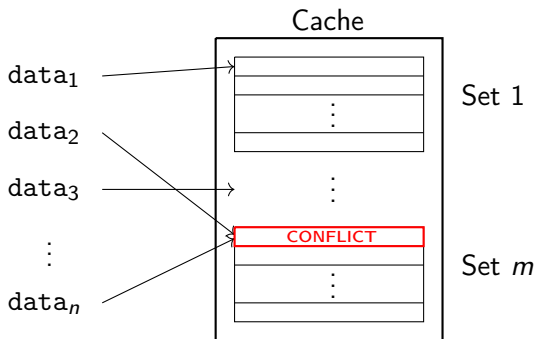
These slides were created by the presenter. Any inconsistencies or inaccuracies are due to the faults of the presenter and not the authors of the paper.

Outline

- Introduction
 - Motivation
 - Background
- Analytics: Divisible Tile Sizes
 - Direct-Mapped Cache
 - Set Associative Cache
- Numerics: Arbitrary Tile Sizes
 - 2D Data Space
 - 3D Data Space
 - Inter-Array Padding
- Experiments and Results
- References

Motivation

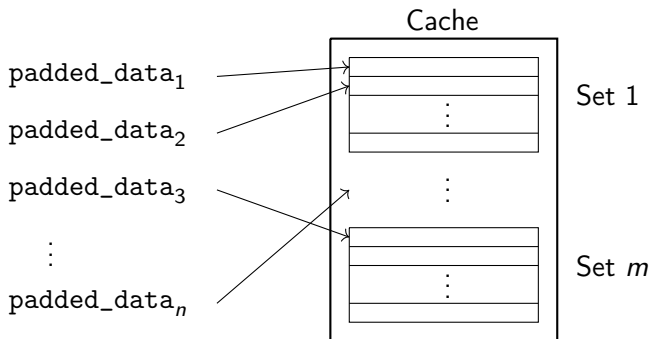
Caches are used to significantly improve performance. However, the number of accessed data elements mapping to the same set can easily exceed the degree of associativity.



Elements $data_2$ and $data_n$ map to the same entry in set m .

Motivation

Array padding (increasing the size of array dimensions) is used to avoid these cache conflicts

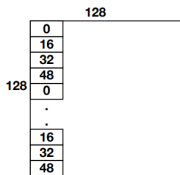


Mathematically, we try to make $f : \text{data} \rightarrow \text{cache}$ "surjective" via array padding to reduce cache conflicts

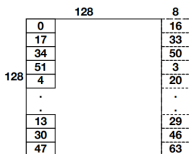
Motivation

Loop nest to symmetrize a square matrix of floating-point numbers using 8-way 32KB cache with line size of 64 bytes (64 sets)

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    B[i][j] = 0.5 *
      (A[i][j] + A[j][i]);
```



The above results in conflict misses for column-wise data access. Now pad by 8 dummy columns by declaring as A[128][136]

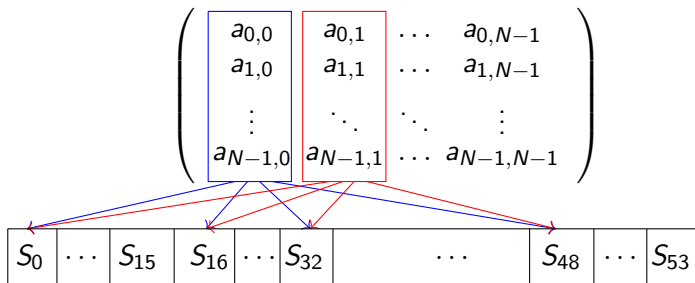


No conflict-misses

Padding results in 250% speedup and 70% less L3 cache misses

Motivation

The problem in the previous slide arises from the spacing of data



Elements $A[0][1], \dots, A[0][7]$ are brought into cache when computing $B[0][:]$. However, when $B[1][:]$ is being computed $A[0][1]$ is no longer in cache because some other data has evicted it due to same cache set mapping.

Problem Statement

From the example, it seems like the number of sets (and cache associativity) and cache conflicts are strongly related. Can we fix this? If so, how?

Given a set of multidimensional arrays and a multidimensional hyperrectangular data footprint for each array, can padding extents for arrays and inter-array spacing be found that completely eliminate conflict misses in a hierarchy of set-associative caches while minimizing the space overhead from the padding itself?

Background: Tiling

Loop nest for matrix vector multiplication

```
for(i=0; i<N; i++)
  c[i] = 0;
  for(j=0; j<N; j++)
    c[i] = c[i] + a[i][j] * b[j];
```

$$\Rightarrow \begin{pmatrix} \boxed{a_{0,0} \quad \dots \quad a_{0,N-1}} \\ \vdots \quad \ddots \quad \vdots \\ a_{N-1,0} \quad \dots \quad a_{N-1,N-1} \end{pmatrix}$$

where the blue box indicates array access for first iteration. After applying tiling using 2×2 blocks

```
for(i=0; i<N; i+=2)
  c[i] = 0;
  c[i+1] = 0;
  for(j=0; j<N; j+=2)
    for(x=i; x<min(i+2, N); x++)
      for(y=j; y<min(j+2, N), y++)
        c[x] = c[x] + a[x][y] * b[y];
```

$$\Rightarrow \begin{pmatrix} \boxed{a_{0,0} \quad a_{0,1} \quad \dots} \\ a_{1,0} \quad a_{1,1} \quad \dots \\ \vdots \quad \vdots \quad \ddots \end{pmatrix}$$

This gets rid of the cache conflicts when too much data is brought in from an iteration that has overlapping cache mapping

Notation

Given a d -dimensional array, we denote

- $N_i = M_i + P_i$ as the number of elements along dimension i ,
- M_i as the number of accessible elements at dimension i ,
- P_i as the amount of padding at dimension i

Given an ℓ -level cache, we denote

- $C_\ell = S_\ell A_\ell B$ as the cache capacity at level ℓ ,
- S_ℓ as the number of sets at level ℓ ,
- A_ℓ as the associativity at level ℓ ,
- B as the cache block size

Given a tile, we denote

- D_i as the size of the tile footprint at dimension i

Assumptions

Given a d -dimensional array, we assume

- dimension $n \rightarrow$ row n for row-major order and dimension $n \rightarrow$ column n for column-major order

Given an ℓ -level cache, we assume

- usual top-down ordering $C_\ell \leq C_{\ell+1}$,
- identical line/block size B at every level

Given a tile, we assume

- tile size D_1 is a multiple of the cache block size B

Direct-Mapped Cache

Theorem

Consider a direct-mapped cache of capacity $C = SB$. A loop nest whose tiles have a d -dimensional array footprint can fully utilize the cache and remain free of self-interference if and only if the following conditions are met:

- ❶ $\forall i, 1 \leq i \leq d, D_i \text{ divides } N_i$
- ❷ $\forall i, 1 \leq i \leq d - 1, \gcd(C / \prod_{1 \leq k \leq i} D_k, N_i / D_i) = 1$

In layman's terms, tiling on a direct-mapped cache will not incur cache conflicts if and only if:

- ❶ The amount of tiled data in each dimension is a factor of how much data there is in that dimension
- ❷ For each dimension $i < d$, the amount of chunks is relatively prime with how many can fit in that dimension

Direct-Mapped Cache

For dimension $d = 2$, the idea is as follows

- 1 Partition cache lines into chunks of size D_1/B
- 2 Since we have S lines, there are $SB/D_1 = C/D_1$ chunks
- 3 The function $f(\text{tile_rows}) \rightarrow \text{chunk}$ is conflict-free if and only if N_1/D_1 is a generator of the $\mathbb{Z}/(C/D_1)\mathbb{Z}$ group

Assume a direct-mapped cache with $S = 64$ and $B = 64$ bytes and matrix A is a 12×12 block matrix with floating-point entries, then

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & \dots \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & \dots \\ \vdots & \ddots & \ddots & \ddots & \ddots \end{pmatrix}$$

Bad Chunks

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & \dots \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & \dots \\ \vdots & \ddots & \ddots & \ddots & \ddots \end{pmatrix}$$

Good Chunks

Direct-Mapped Cache

Chunk mappings for the two tilings in the previous slide would be

0		5
6		11
12		17
⋮		⋮
28		1
2		7

Bad Chunks

0		2
3		5
6		8
⋮		⋮
8		10
11		13

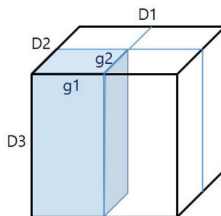
Good Chunks

The restriction that N_1/D_1 generates $\mathbb{Z}/(C/D_1)\mathbb{Z}$ is necessary to avoid cache conflicts.

Set-Associative Cache

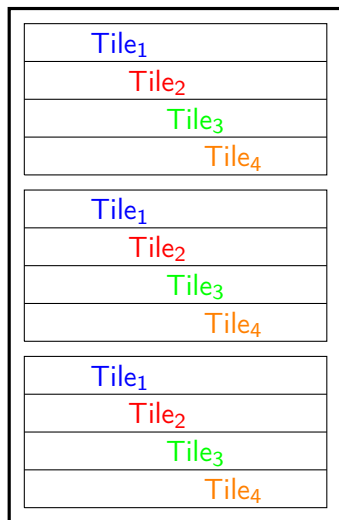
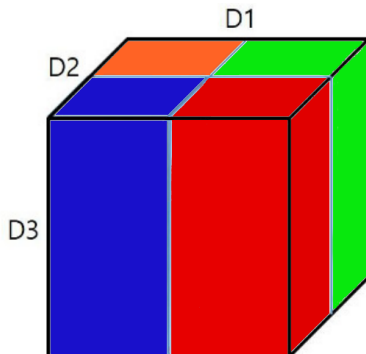
To extend the result to the set-associative case, we introduce the characteristic number g_i . As shown, we have that

- the smaller tile $g_1 \times \cdots \times g_{d-1} \times D_d$ is completely contained in $D_1 D_2 \dots D_d$,
- tile $D_1 D_2 \dots D_d$ is $A = 4$ times bigger than tile $g_1 \times \cdots \times g_{d-1} \times D_d$.



If the enclosed tile $g_1 \times \cdots \times g_{d-1} \times D_d$ is free of self-interference conflicts in a direct-mapped cache, then the A -times larger tile $D_1 D_2 \dots D_d$ is free of self-interference conflicts in an A -associative cache of the same capacity.

Set-Associative Cache



Set-Associative Cache

Theorem

Consider a set-associative cache of capacity $C = SAB$. For all $1 \leq i \leq d - 1$, let $g_i = \gcd(S / \prod_{1 \leq k \leq i-1} g_k, N_i)$. A loop nest whose tiles have a d -dimensional array footprint can fully utilize the cache and remain free of self-interference if and only if the following conditions are met:

- ① $\forall i, 1 \leq i \leq d - 1, \exists j, 1 \leq j \leq i, \prod_{1 \leq k \leq j} g_k$ divides $D_j \prod_{1 \leq i \leq j-1} g_i$
- ② $\exists i, 1 \leq i \leq d, S$ divides $D_i \prod_{1 \leq k \leq i-1} g_k$

In layman's terms, tiling on an A -associative cache will not incur cache conflicts if and only if:

- ① the tiling ratio up to the first i dimensions is a factor of some enclosing tile (this guarantees we cover the tiling perfectly)
- ② the associativity is evenly used

Set-Associative Cache

Previous Theorem has the following implications

- stride between chunks can be any integer dividing set size AB
- $g_i = D_i$ hits each set exactly once
- $g_i = D_i/A$ is analogous to direct-mapped case
- lower g_i means less conflicts to be tolerated at higher dimensions

Tile Hierarchies

Conflict-free padding under direct-mapped cache is not feasible for multiple levels of cache

L1 cache	L2 cache
$C_1 = 32$ KB	$C_2 = 256$ KB
$B = 64$ bytes	$B = 64$ bytes
$S_1 = 512$	$S_2 = 4096$
$D_1 = 8$	$D_2 = 32$
$M_1 = 1024$ doubles	

For conflict-free tiles in

- L1: N_1 must be of the form $1024 + 8(2k + 1)$ because $\gcd(1024 + 8(2k+1), 4096) = 8$,
- L2: N_1 must be of the form $1024 + 32(2k + 1)$ because $\gcd(1024 + 32(2k+1), 4096) = 32$

Since $\{1024 + 8(2k + 1) \mid k \geq 0\} \cap \{1024 + 32(2k + 1) \mid k \geq 0\} = \emptyset$,
L1 and L2 can not be simultaneously conflict-free

Tile Hierarchies

Theorem

Consider a high-level cache of capacity $C_\ell = S_\ell A_\ell B$ and a low-level cache of capacity $C_{\ell'} = S_{\ell'} A_{\ell'} B$. For all $1 \leq i \leq d-1$, let $g_i = \gcd(S_\ell / \prod_{1 \leq k \leq i-1} g_k, N_i)$ and $g'_i = \gcd(S_{\ell'} / \prod_{1 \leq k \leq i-1} g'_k, N_i)$. For both inner and enclosing tiles to fully utilize their respective caches levels and remain free of self-interference, it is sufficient that the following conditions are met:

- ① $\forall i, 1 \leq i \leq d-1, \exists j, 1 \leq j \leq i, \prod_{1 \leq k \leq i} g_k \text{ divides } D_j \prod_{1 \leq i \leq j-1} g_i$
- ② $\forall i, 1 \leq i \leq d-1, \exists j, 1 \leq j \leq i, \prod_{1 \leq k \leq i} g'_k \text{ divides } D'_j \prod_{1 \leq i \leq j-1} g'_i$
- ③ $\exists k, 1 \leq k \leq d, S_\ell \text{ divides } D_k \prod_{1 \leq i \leq k-1} g_i$
- ④ $\exists k, 1 \leq k \leq d, S_{\ell'} \text{ divides } D_k \prod_{1 \leq i \leq k-1} g'_i$
- ⑤ $\forall i, 1 \leq i \leq d-1, \prod_{1 \leq k \leq i} D'_k \text{ divides } A_{\ell'} \prod_{1 \leq k \leq i} D_k$

Padding for Arbitrary Tile Sizes

Previous slides address padding for divisible tiles (i.e. tile data footprint divides cache capacity). However, this is not always the case.

Is it always feasible to achieve conflict-free padding for any arbitrary tile size as long as the total tile data footprint is no greater than cache capacity?

Lemma

For an arbitrary data tile with footprint less than or equal to cache capacity, there always exists some padding that makes the tile conflict-free.

Notation

Given a 2/3-D array, we denote

- $M_2 M_1 / M_3 M_2 M_1$ as the size of the array,
- $D_2 D_1 / D_3 D_2 D_1$ as the data tile size,
- $P_1 / P_2, P_1$ as the minimal padding extents

Given elements x and y of an array, we denote

- $x \sim y$ as x conflicts with y

2D Data Space for Direct-Mapped Cache

Given an arbitrary 2D data tile of size $D_2 D_1$ and array A , we seek

$$\min_{P_1 \in \mathbb{Z}/S\mathbb{Z}} M_2(M_1 + P_1) \text{ s.t. } A \text{ is conflict-free}$$

Checking S padding values $(0, B, 2b, \dots, (S-1)B)$ is enough to find a conflict-free padding.

The main idea is as follows

- 1 consider all possible pairwise tile elements conflicts
- 2 find padding choices that create such conflicts
- 3 eliminate all such padding choices
- 4 choose from remaining padding choices that has smallest space overhead

2D Data Space for Direct-Mapped Cache

Given a $D_2 \times D_1$ tile, there are $\binom{D_2 D_1}{2}$ possible cases to consider. However, we can reduce the number of potentially conflicting data pairs to be considered.

Lemma

Consider a 2D tile space $A[i_2][i_1]$ such that $0 \leq i_2 \leq D_2$ and $0 \leq i_1 \leq D_1$. For all i_2, i_1 in the data space, there are no cache conflicts $A[0][0] \sim A[i_2][i_1]$ and $A[0][D-1] \sim A[i_2][i_1]$ if and only if the entire data space is conflict-free

A consequence of the lemma is that checking for $A[0][0] \sim A[i_2][i_1]$ and $A[0][D-1] \sim A[i_2][i_1]$ is sufficient to ensure the entire data space is conflict-free.

2D Data Space for Direct-Mapped Cache

Given tile $D_2 \times D_1$ and three different mappings, we can see which one will be the conflict-free in the tile and therefore over the entire data space

 $D_2 \times D_1$

0	3	6
1	4	7
2	5	0

 $A[0][0] \sim A[i_2][i_1]$
 $D_2 \times D_1$

0	3	6
1	4	7
2	5	6

 $A[0][D-1] \sim A[i_2][i_1]$
 $D_2 \times D_1$

0	3	6
1	4	7
2	5	8

Conflict-free

2D Data Space for Direct-Mapped Cache

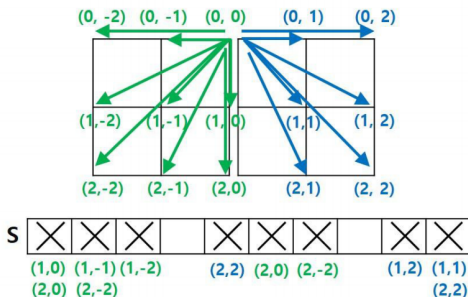
The previous lemma can be restated in terms of a test using a Diophantine equation

Lemma

Given a 2D array $A[\star][N_1]$ with padded size N_1 , the tile data space $A[i_2][i_1]$, $0 \leq i_2 < D_2$, $0 \leq i_1 < D_1$, is conflict-free if and only if $(N_1 i_2 + i_1)/B \bmod S \neq 0, \forall i_2, i_1$ such that $0 \leq i_2 < D_2, -D_1 < i_1 < D_1, i_1 \equiv 0 \bmod B$ and $(i_2, i_1) \neq (0, 0)$.

2D Data Space for Direct-Mapped Cache

Consider a direct-mapped cache with $S = 10$, $B = 1$, a 2D array of size 10×10 , and a 3×3 data tile. Using the previous lemma, we can see which values of N_1 are allowed. For example, $(i_2, i_1) = (1, 1)$ results in $(N_1 + 1) \equiv 0 \pmod{10}$. Therefore $N_1 = 9$ would result in conflicts. (S represents padded extent)



3D Data-Space for Direct-Mapped Cache

The 3D data space case naturally follows from the 2D case

Lemma

Let $A[i_3][i_2][i_1]$ be a 3D tile data space, with $0 \leq i_3 \leq D_3$, $0 \leq i_2 \leq D_2$, and $0 \leq i_1 \leq D_1$, with the additional constraint that $i_1 \equiv 0 \pmod{B}$. For all i_3, i_2, i_1 in the data space, there is no cache conflict $A[0][0][0] \sim A[i_3][i_2][i_1]$, $A[0][0][D_1 - 1] \sim A[i_3][i_2][i_1]$, $A[0][D_2 - 1][0] \sim A[i_3][i_2][i_1]$, and $A[0][D_2 - 1][D_1 - 1] \sim A[i_3][i_2][i_1]$ if and only if the data space is conflict-free.

A consequence of the lemma is checking the four conditions is sufficient to ensure the entire data space is conflict-free.

3D Data-Space for Direct-Mapped Cache

Similarly to the 2D case, we can derive the following central lemma

Lemma

For $\forall i_3, i_2, i_1$ such that $0 \leq i_3 < D_3, -D_2 < i_2 < D_2, -D_1 < i_1 < D_1, i_1 \equiv 0 \bmod B, (i_3, i_2, i_1) \neq (0, 0, 0)$ and given N_2, N_1 , the data space is conflict free if and only if $(N_2 N_1 i_3 + N_1 i_2 + i_1) / B \not\equiv 0 \bmod S$.

The algorithm is similar to that of the 2D case, exploring all necessary points (i_3, i_2, i_1) in the data space to eliminate unsuitable choices for conflict-free padding.

Algorithm

```

for(i2=0; i2<D2; i2++)
    c = gcd(i2,S);
    inv = (i2/c)^-1 % (S/c)
    for(i1=-(D1+B)/B; i1<=(D1-B)/B; i1++)
        if(i1 % c == 0)
            for(i0=0; i0<c; i0++)
                v = (-i1 * inv) % (S/c)
                PadOk[v+i0*(S/c)] = 0
for(i0=0; i0<S; i0++)
    if(PadOk[M1+i0*B%S] == 1)
        return i0*B
return 0

```

Complexity is $\mathcal{O}(S \log S)$ for 2D. Worst case is $\mathcal{O}(S^2)$ for 3D with an average of $\mathcal{O}(S \log S + SD_1/B)$.

2D Data-Space for Set Associative Cache

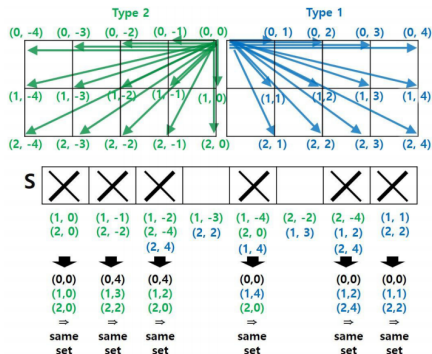
Lemma

For a cache with a associativity of A , $\forall k$ a data point $A[0][k]$ has less than A conflicts with other points in the data space if and only if the data space is conflict free.

In contrast to 2D direct-mapped case, we must count the number of conflicts. We must also check conflict counts w.r.t. all elements in the top row of the data tile; if any of these involve more conflicts than the cache associativity, the padding value is unsuitable.

2D Data-Space for Set Associative Cache

Consider a cache with $S = 8$, $A = 2$, $B = 1$, $M_1 = 80$, $D_1 = 5$, and $D_2 = 3$. Data tile has footprint $5 \times 3 = 15$ blocks and cache capacity is 16 blocks. (S represents padding value)



Looking for $(N_1 i_2 + i_1) / B \bmod S = ((80 + P_1) i_2 + i_1) \bmod 8 \neq 0$. This is all done with complexity $\mathcal{O}(SA \log S)$

3D Data-Space for Set Associative

The *PAdvisor* algorithm for 3D tiles and set-associative caches uses a combination of the approach previously described for the 3D direct-mapped case and the approach for handling associativity in 2D tiles. The worst case complexity is $\mathcal{O}(A^2S^2)$ and average case complexity is $\Theta(S \log S + SD_1/B + AS)$

Inter-Array Padding

Assume a 4-way cache with $S = 8$. Also, for some padding N_i , assume interference counts of 1,3,1,2,2,2,0,2 for two arrays.

	1	3	1	2	2	2	0	2	
	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	
+	2	1	3	1	2	2	2	0	2
	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	
<hr/>									
	3	4	4	3	4	4	2	2	
	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	

Without shifting, interference counts would be 2,6,2,4,4,4,0,4. This is bad since we only have 4-way associativity.

Benchmarks

The work was evaluated over six benchmarks

- ➊ **ADI** - Alternating Direction Implicit solver used for PDEs
- ➋ **MKL-FFT** - Fast Fourier Transform
- ➌ **HPGMG** - High Performance Geometric Multigrid is an adaptive mesh refinement solver
- ➍ **DGEMM** - basic linear algebra solver
- ➎ **Stencil-2D** - stencil method to compute Jacobi, PDE, or function smoothing using orthogonal direction neighbors
- ➏ **Stencil-3D** - stencil method to compute Jacobi, PDE, or function smoothing using orthogonal direction neighbors

Experimental Setup

The tests were run on two machines

- Intel Core i7-2600K @ 3.4 GHz / Intel Core i7-4770 @ 3.5 GHz
- Each machine ran Linux and had
 - L1: 8-way, $S = 64$, 32 KB
 - L2: 8-way, $S = 512$, 256 KB
 - L3: 16-way, $S = 8192$, 8192 KB
 - $B = 64$ bytes using double-precision floating point, therefore each cache line had 8 elements
- Manually computed padding for smallest cache level fully enclosing data space (e.g. a column of data for MKL-FFT and ADI)

Results

N	1 core			4 cores		
	no pad	pad	Imp.	no pad	pad	Imp.
512	8.94	9.14	2.3%	18.72	21.25	13.5 %
640	7.80	7.95	1.9%	21.21	21.94	3.4%
768	7.78	7.92	1.7%	22.43	23.27	3.7%
896	7.49	7.56	0.9%	21.69	21.87	0.8%
1024	8.46	9.08	7.3%	18.02	22.26	23.6%
1280	6.90	7.40	7.1%	16.78	18.85	12.4%
1536	6.27	6.91	10.3%	16.42	17.56	7.0%
1792	6.55	7.24	10.5%	16.46	17.94	9.0%
2048	6.20	8.14	31.1%	13.65	19.37	41.9%
2560	5.87	6.86	16.7%	12.92	19.07	47.6%
3072	5.78	7.04	21.6%	11.54	16.99	47.3%
3584	5.54	6.53	17.8%	12.18	17.58	44.3%
4096	6.40	7.98	24.6%	14.55	19.61	34.7%

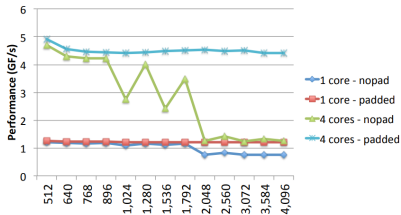
MKL-FFT on Sandy Bridge

N	1 core			4 cores		
	no pad	pad	Imp.	no pad	pad	Imp.
512	10.04	11.13	11.0%	24.73	24.76	0.1%
640	8.47	9.23	9.0%	23.53	24.55	4.3%
768	8.49	9.31	9.6%	17.88	26.07	45.8%
896	8.67	9.04	4.3%	22.05	26.84	21.7%
1024	9.90	11.33	14.5%	24.86	28.62	15.1%
1280	8.50	8.62	1.5%	21.18	23.33	10.1%
1536	7.88	8.10	2.8%	20.19	22.37	10.8%
1792	8.46	8.49	0.4%	22.00	23.66	7.5%
2048	7.40	10.38	40.3%	15.62	25.70	64.6%
2560	7.36	8.10	10.1%	18.80	22.68	20.6%
3072	7.35	8.09	10.0%	18.57	22.96	23.6%
3584	7.05	7.77	10.2%	18.46	22.69	22.9%
4096	8.41	9.63	14.4%	21.07	25.23	19.7%

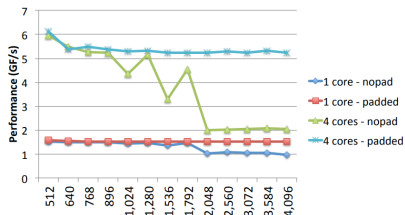
MKL-FFT on Haswell

Tables show GF/s when performing MKL-FFT over 2D problem sizes using single and multi cores

Results



ADI on Sandy Bridge



ADI on Haswell

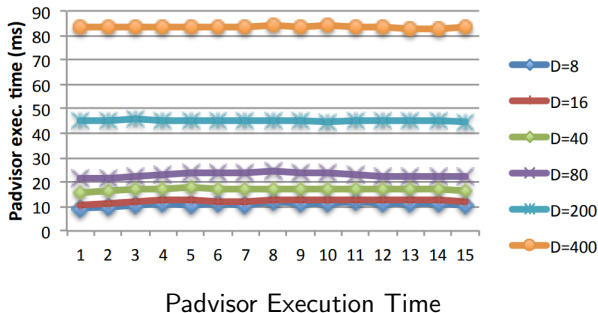
Impact of padding on SB (left) and HSW (right)

Results

N	SB			HSW		
	no pad	intra	intra+inter	no pad	intra	intra+inter
1024	10.22	21.54	25.22	20.23	32.16	32.30
1536	13.03	27.44	33.52	26.23	39.23	39.89
2048	13.06	27.66	32.02	26.19	37.97	38.54
256	13.74	22.42	24.76	20.19	27.94	30.45
384	18.63	21.86	22.11	24.06	27.00	27.29
512	17.74	20.29	20.31	22.28	27.08	27.08

Stencil-2D (top three entries) and Stencil-3D (bottom three entries), in GFlops/s using 4 cores

Results






The figure shows the execution time in ms for a variety of tile sizes and fixed inner-most size. The computations were over a 3D data space using non-power-of-two data tiles with L3 16-way cache.

Conclusions

- *PAdvisor* works for arbitrary multidimensional tile data footprints, and nested hierarchical tiles
- *PAdvisor* provides conflict-free padding solutions with minimal padding space overhead
- *PAdvisor* is very fast and can effectively be used for the tuning of padded data

References

-  Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, P. Sadayappan. *Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses*. PLDI 2016.
-  Z. Li and Y. Song. *Automatic tiling of iterative stencil loops*. ACM Trans. Program. Lang. Syst., 26(6):975-1028, Nov. 2004.
-  G. Rivera and C.-W. Tseng. *Tiling optimizations for 3D scientific computations*. In SC'00, page 32. IEEE, 2000.