Enabling temporal blocking for stencil computations by multicore-aware wavefront parallelization

G. Wellein, G. Hager, T. Zeiser, H. Fehske
M. Wittmann, J. Habich, J. Treibig

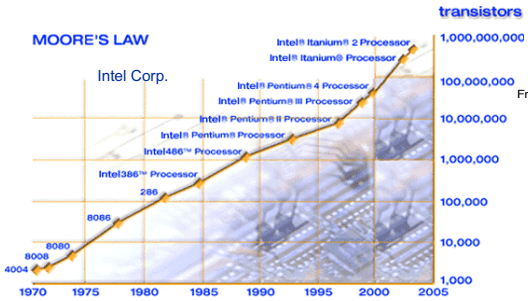
G. Wellein, G. Hager, T. Zeiser, H. Fehske, accepted for publication in Proceedings of the 33rd IEEE Computer Software and Applications Conference (COMPSAC 2009), IEEE Computer Society Press.

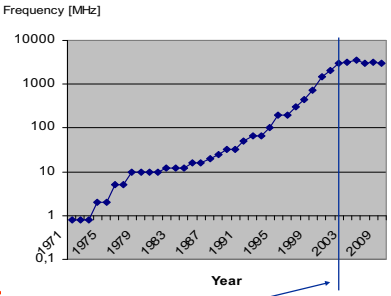
Enabling temporal blocking for stencil computations by multi-aware wavefront parallelization

Welcome to the multi-/manycore era

Moore's law – still the driving force

- **1965 G. Moore claimed**
  #transistors on processor chip doubles every 12-24 months



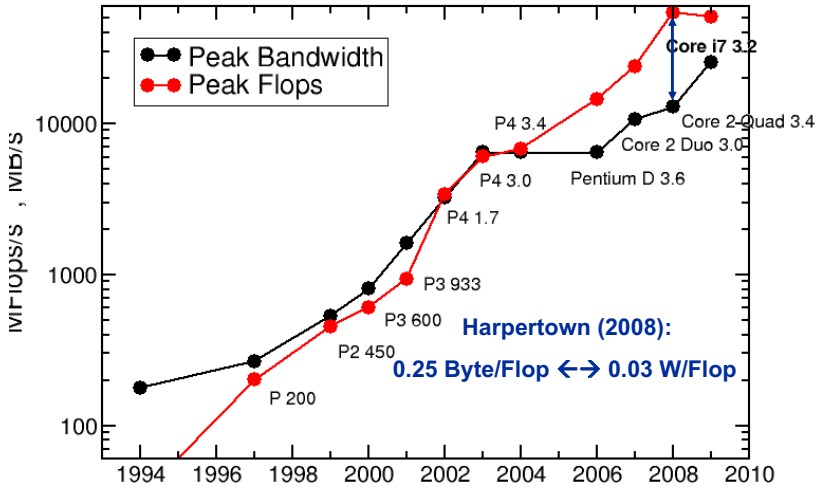


- **Processor speed grew roughly at the s**
  My computer: 350 MHz (1998) – 3,000 MHz (2004)
- **Problem since 2004: Power dissipation (supply voltage)**

The x86 multicore evolution….

… even more trouble ahead?

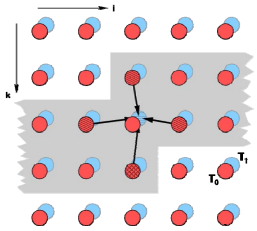
Single socket Intel chips: Peak Performance & Memory Bandwidth over time



- **Jacobi iteration: Basics and baseline implementation**
- **Conventional temporal blocking**
- **Wavefront approach**

Jacobi Solver
Basics: 2 arrays; naïve version

```
do k = 1 , Nk
  do j = 1 , Nj
       do i = 1 , Ni
          y(i,j,k) = a*x(i,j,k) + b*
                     (x(i-1,j,k)+x(i+1,j,k)+
                      x(i,j-1,k)+x(i,j+1,k)+
                      x(i,j,k-1)+x(i,j,k+1))
enddo; enddo; enddo
```



- Performance Measure: Million Lattice Site Updates per second (MLUPs)

  Equivalent MFLOPs: 8 FLOP/LUP * MLUPs

- Bandwidth requirements: **24 Byte / Lattice Site Update** (LUP) if:

  N*N*8Byte*2 < Cache size → Cache size = 2 MB → N ~ 350

- Performance estimate: **B_M / (24 Byte/LUP)**
  (B_M : attainable memory bandwidth as measured with `stream`)

Jacobi Solver
Basics: 2 arrays; `nontemporal` stores

```
do k = 1 , Nk
  do j = 1 , Nj
       do i = 1 , Ni
          y(i,j,k) = a*x(i,j,k) + b*
                     (x(i-1,j,k)+ x(i+1,j,k) + x(i,j-1,k)
                     +x(i,j+1,k)+ x(i,j,k-1) + x(i,j,k+1) )
enddo; enddo; enddo
```

- ReadForOwnership (RFO) is executed for y on cache based systems
  - →Suppress RFO on x86 with `nontemporal` stores
  - →Bandwidth requirements: **16 Byte / LUP**
  - →Performance estimate: **$P_0 = B_M$ / (16 Byte/LUP)**
  - →Time per LUP: **T_0 = 16 Byte / B_M**

  Baseline for further discussions

  - →E.g: B_M = 8 GByte → P_0=500 MLUPs (T_0=2 ns)

Jacobi solver
Baseline implementation

- Implement spatial blocking (Inner loops: b_k,b_j,b_i)

  $b_k = b_j = 10$; **$b_i = N_i$**
- At source code level: `!$DEC VECTOR NONTEMPORAL`
  - Requires stores aligned to 16 Byte boundaries
  - And for some compiler versions the load streams as well!
- Implement nontemporal stores with intrinsics

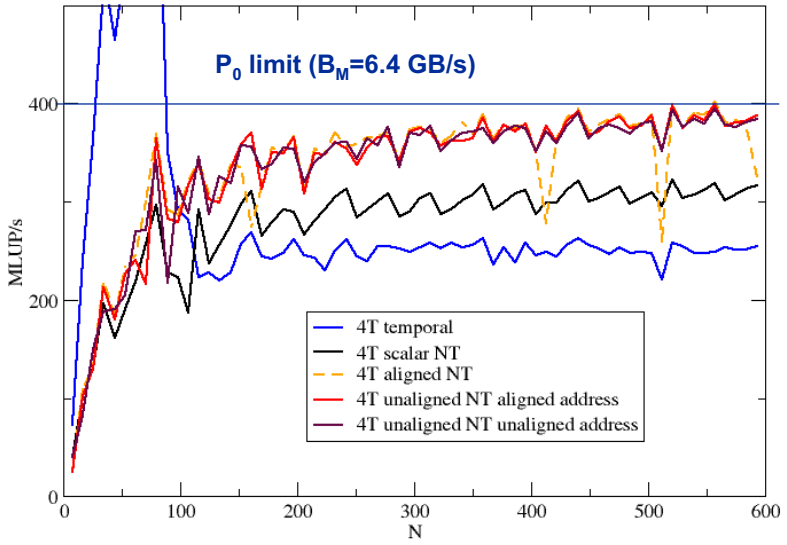
```
// unaligned store stream ?
// peel off unaligned 1st iteration
for(i=istart; i<((ie-istart) & (-2)); i+=2) {
    xmm1 = _mm_loadu_pd(s_line+i-1); xmm8 = _mm_loadu_pd(s_line+i);
    xmm2 = _mm_loadu_pd(s_line+i+1); xmm3 = _mm_loadu_pd(s_zp+i);
    xmm4 = _mm_loadu_pd(s_zm+i);     xmm5 = _mm_loadu_pd(s_yp+i);
    xmm6 = _mm_loadu_pd(s_ym+i);     xmm8 = _mm_mul_pd(xmm8,xmmnul);
    xmm7 = _mm_add_pd(xmm1,xmm2);    xmm7 = _mm_add_pd(xmm7,xmm3);
    xmm7 = _mm_add_pd(xmm7,xmm4);    xmm7 = _mm_add_pd(xmm7,xmm5);
    xmm7 = _mm_add_pd(xmm7,xmm6);    xmm7 = _mm_mul_pd(xmm7,xmmoos);
    xmm7 = _mm_add_pd(xmm7,xmm8);
    // aligned NT store
    _mm_stream_pd(d_line+i,xmm7);}
  if((ie-istart) & 1) { // remainder}
```


Jacobi solver
Aligned NT store

- **Loads (`s_*`) are always done unaligned (no control over mutual alignment of rows; could be fixed)**
- **Standard NT store (`movntpd`) must use 16-byte aligned address**
  - Could use `maskmovdqu` for unaligned (even scalar) NT store, but this is slightly more inefficient in most cases
- **If store stream (`d_line`) is not 16-byte aligned, peel off the first iteration and do the rest with packed SSE**
- **Peel off the last iteration if required (remainder)**

- **Next slide: performance comparison (4 threads, 2x Xeon 5160) between**
  - Temporal (standard) store
  - Scalar NT store
  - Aligned NT store (movntpd)
  - Unaligned NT store on aligned address
  - Unaligned NT store on unaligned address

The different ways of storing…

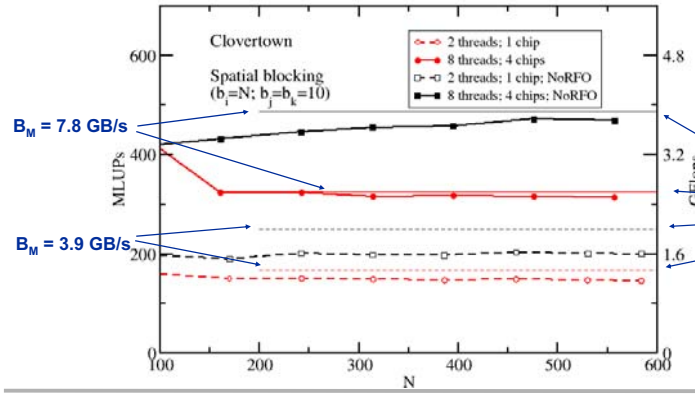


Jacobi solver
Baseline

[2] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick: *Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures.* In: ACM/IEEE (Ed.): Proceedings of the ACM/IEEE SC 2008 Conference (Supercomputing Conference '08, Austin, TX, Nov 15–21, 2008).

→ **Clovertown: 2.5 GFlop/s**

(including opt. spatial blocking and NoRFO)



Performance limits (P_0) based on STREAM bandwidth with RFO and without RFO (NoRFO)

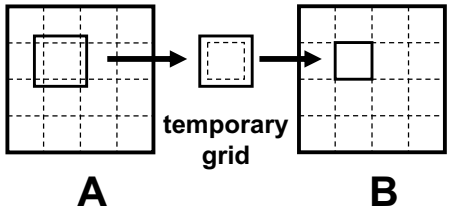
Jacobi solver
Benchmark architectures: Intel Quad-/Hexa-Cores

		Clovertown	Nehalem	Dunnington
Type		Xeon 5345 @2.33 GHz	"Core i7" @2.66 GHz	Xeon 7460 @2.66 GHz
L1 group	size [kB]	32	32	32
	TRIAD GB/s	3.9	11.6	3.0
L2 group	L2 size [MB]	4	0.25	3
	# cores	2	1	2
	TRIAD GB/s	4.0	see L1	3.5
L3 group / socket	L3 size [MB]	-	8	16
	# cores	4	4	6
	TRIAD GB/s	4.0	16.6	3.5
System	# sockets	2	2	4
	raw bw [GB/s]	21.3	51.2	34.0
	TRIAD GB/s	7.8	32.7	13.2

Stream TRIAD (array size: 20,000,000; `nontemporal stores` via compiler)

Nehalem: Early access; pre-production; SMT disabled

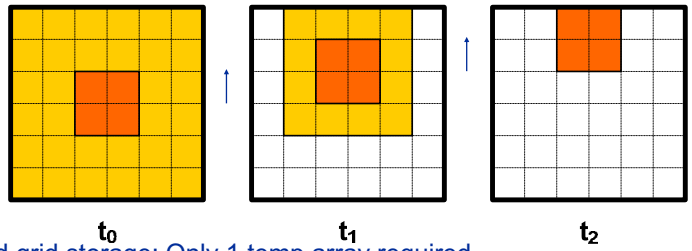
Conventional temporal blocking approaches



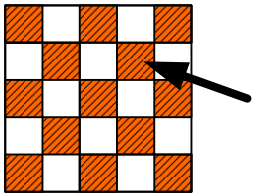
Multiple sweeps / timespteps on a small spatial block (in-cache)

Conventional temporal blocking
Halo implementation

- Load (N_b*t_b)x(N_b*t_b) block & perform t_b time steps on N_b X N_b block
- N_b=2; t_b=2:

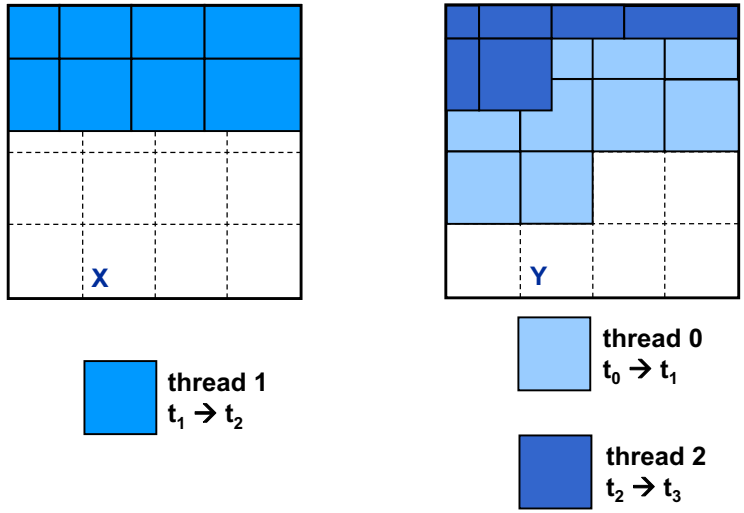


- Compressed grid storage: Only 1 temp array required
- Non-diagonal shift ensures alignement of load and stores!

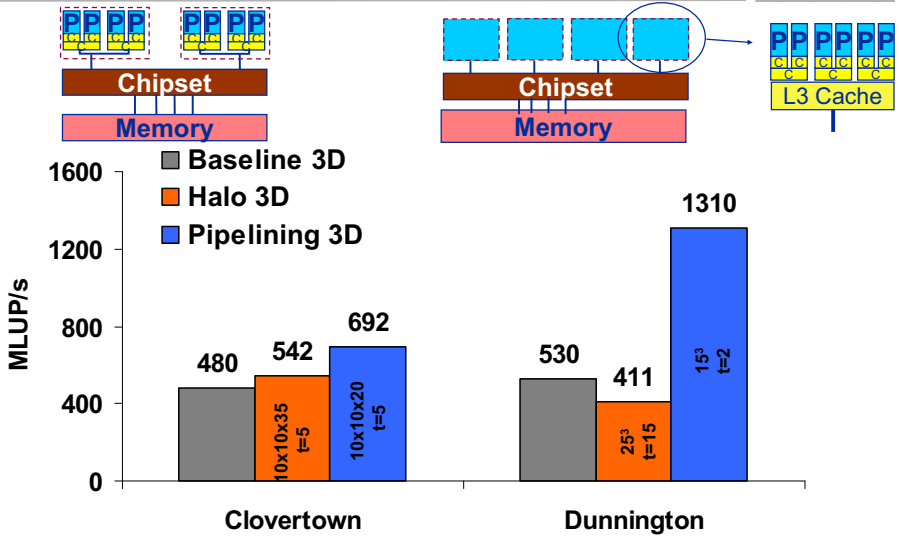


- destination cell for compressed grid
- stencil updated auto vectorizable

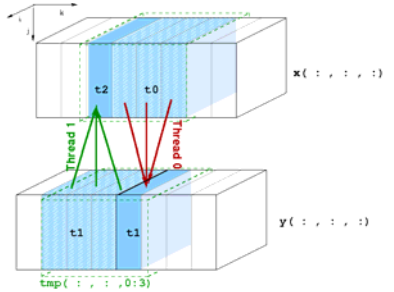
Conventional temporal blocking
Pipelining approach



Conventional temporal blocking
Performance



Enabling temporal blocking through wavefront parallelization



Multiple sweeps / timespteps on successive planes by different cores/threads

Jacobi solver
Wavefront parallelization: Temporal blocking

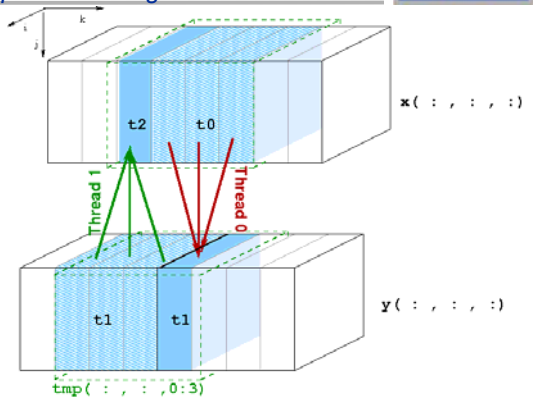
Run second update with second thread

- appropriately shifted
- using a shared caches

at the same time:

Thread 0:
$x(k-1:k+1)_t$ → $y(k)_{t+1}$

Thread 1:
$y(k-3:k-1)_{t+1}$ → $x(k-2)_{t+2}$



Mem. transfers for a sinlge LUP (assuming cache_size > 8 (i-j) planes):

Thread0: LD x_t; LD y_t (RFO)

Thread1: ST x_{t+2}; ST y_{t+2} (Cache line evict)

→ 32 Byte / 2 LUP

Same as baseline !?!?

Jacobi solver
Wavefront parallelization: Temporal blocking

`y(:,:,:)` is obsolete!

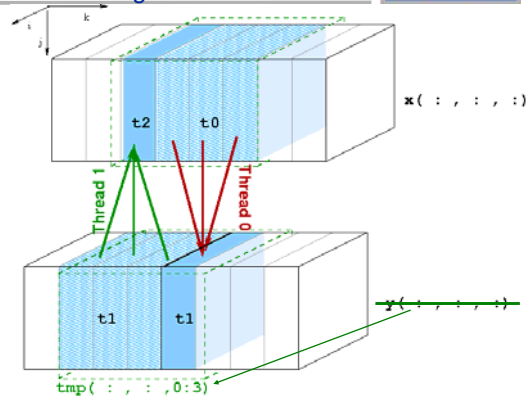
Use small buffer
`tmp(:,:,0:3)`
which fits into the cache

↓

Save main memory data transfers for `y(:,:,:)` !

↓

16 Byte / 2 LUP !



Compare with baseline: Maximum speed-up of 2 can be expected

(assuming infinitely fast cache and
no overhead for OMP BARRIER after each k-iteration)

Jacobi solver
Wavefront parallelization: Temporal blocking

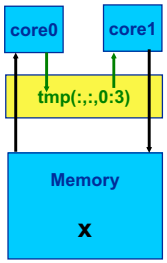
Thread 0: $x(:,:,k-1:k+1)_t$ → $tmp(:,:,mod(k,4))$

Thread 1: $tmp(:,:,mod(k-3,4):mod(k-1,4))$ → $x(:,:,k-2)_{t+2}$

Performance model including finite cache bandwidth (B_C)

Time for 2 LUP:

$$T_{2LUP} = 16\ \text{Byte}/B_M + x * 8\ \text{Byte} / B_C = T_0 (1 + x/2 * B_M/B_C)$$



Minimum value: x =2

Speed-Up vs. baseline:

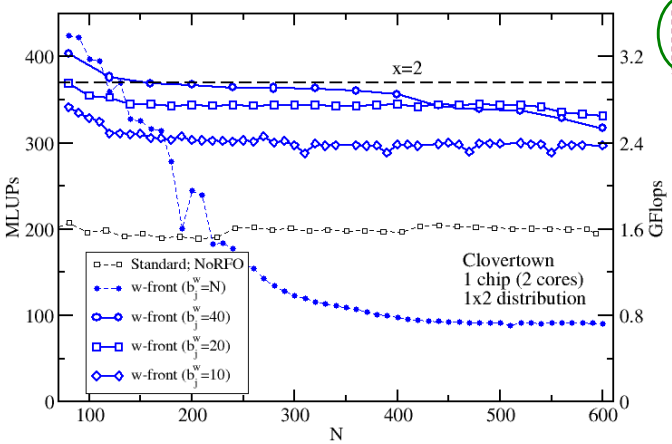
$$S_W = 2*T_0/T_{2LUP} = 2\ (1 + B_M/B_C)$$

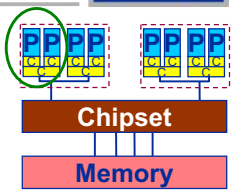
BC and BM are measured in saturation runs:

Clovertown: B_M/B_C = 1/12 → S_W = 1.85

Nehalem1 : B_M/B_C = 1/4 → S_W = 1.6

Jacobi solver
Wavefront parallelization: L2 group Clovertown





Implement blocking in j-direction (b_j^w) to ensure that tmp(:,:,0:3) & 4 planes of x stay in cache!

→ Speed-Up:~1.7-1.8x! (S_W = 1.85)

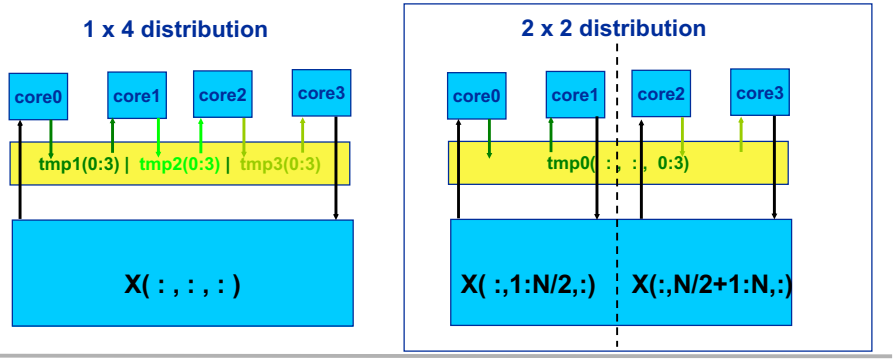
Jacobi solver
Wavefront parallelization: New choices on native quad-cores

Thread 0: `x(:,:,k-1:k+1)`$_t$ → *tmp1(mod(k,4))*

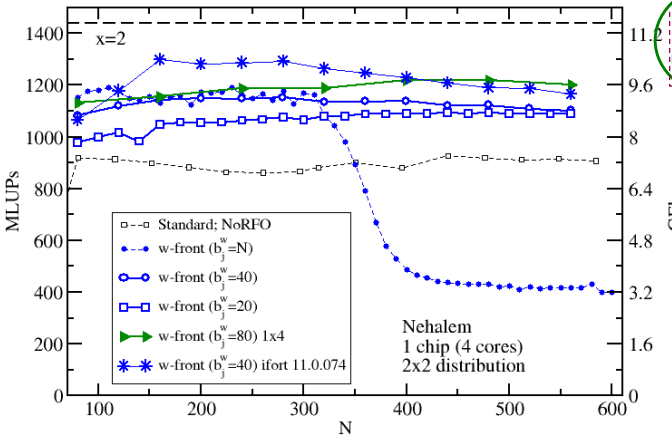
Thread 1: *tmp1(mod(k-3,4):mod(k-1,4))* → *tmp2(mod(k-2,4))*

Thread 2: *tmp2(mod(k-5,4:mod(k-3,4))* → *tmp3(mod(k-4,4))*

Thread 3: *tmp3(mod(k-7,4):mod(k-5,4))* → `x(:,:,k-6)`$_{t+4}$



Jacobi solver
Wavefront parallelization: L3 group Nehalem1

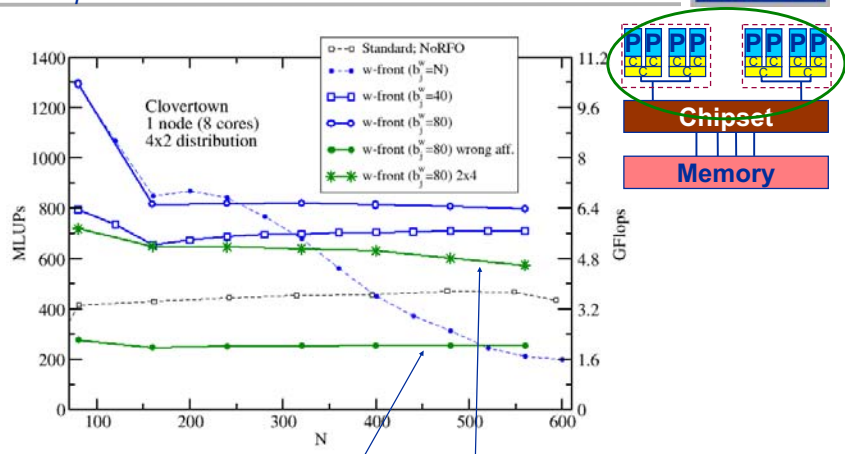


400^3 bj=40	MLUPs
1 x 2	786
2 x 2	1230
1 x 4	1254

Performance model indicates some potential gain → new compiler tested.

Only marginal benefit when using 4 wavefronts → A single copy stream does not achieve full bandwidth

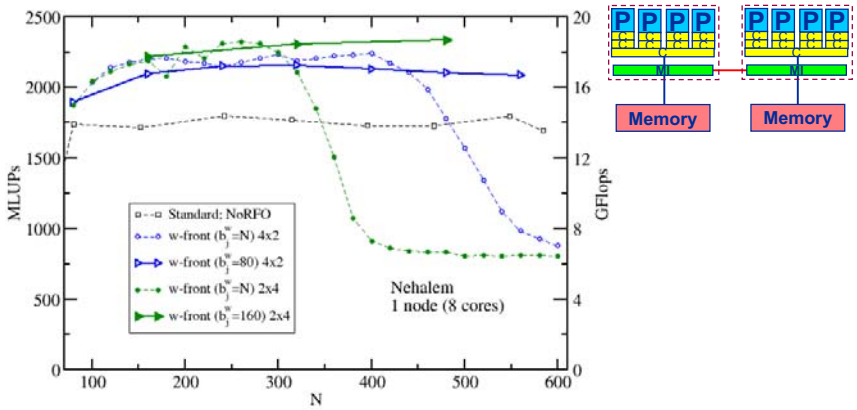
Jacobi solver
Wavefront parallelization: full Clovertown node



→ Thread affinity matters!

→2x4: Run 4 wavefronts in two domains at the same time (no shared cache for 4 threads/cores)

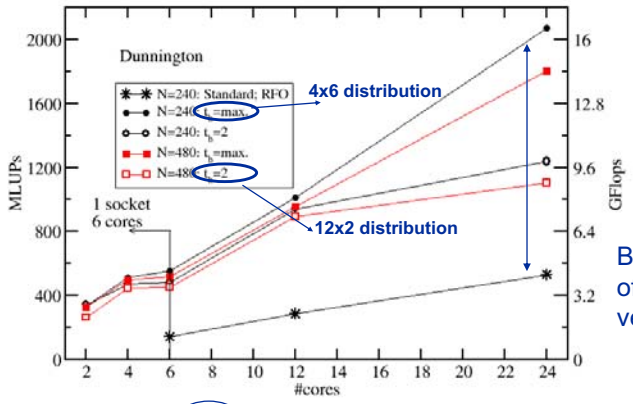
Jacobi solver
Wavefront parallelization: full Nehalem node



→Only limited benefit on Nehalem

→2x4: Run 4 wavefronts in two domains at the same time performs best! (4 wavefronts require 3 tmp arrays for data exchange)

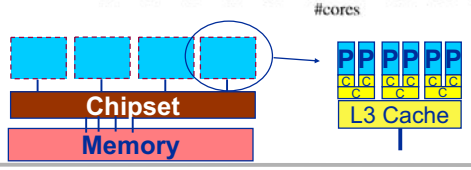
Jacobi solver
Wavefront parallelization: full Dunnington node



Speed-Up of WF:

3,…,4 x !

Bad performance of baseline NoRFO version!



4 socket HexaCore system

Maximum number of useful wavefronts: t_b=6

Jacobi solver
Wavefront parallelization: status

- No in-cache optimizations implemented / explored so far
- Wavefront parallelization for Jacobi beneficial if
  - **Multi-Core Chip is bandwidth starved (one core can sustain main memory bandwidth)**
  - **Large shared (on-chip) cache is available**
- Easy to implement and parallelize but hybrid approach is required if used in a larger application, e.g. as a smoother in MG
- Can easily be implemented for other stencil based methods:
  - **3D LBM → Proc. of ParCFD2009**
  - **Gauß-Seidel → first tests**

- **Financial support through**
  - BMBF Project SKALB (01 IH08003A)

Bundesministerium für Bildung und Forschung

  - KONWIHR-II project OMI4papps

KONWIHR II