

Index of /markusp/teaching/263-2300-ETH-spring11/slides

	Name	Last modified	Size	Description
	Parent Directory	-		
	arch.pdf	04-Mar-2011 16:08	380K	
	blasqr.pdf	21-Mar-2011 09:01	95K	
	class01.pdf	21-Feb-2011 15:24	1.9M	
	class02.pdf	25-Feb-2011 11:40	585K	
	class03.pdf	27-Feb-2011 16:06	586K	
	class04.pdf	08-Mar-2011 10:06	698K	
	class05.pdf	09-Mar-2011 11:10	559K	
	class06.pdf	19-Mar-2011 15:11	555K	
	class07.pdf	25-Mar-2011 11:20	666K	
	class08.pdf	21-Mar-2011 09:44	570K	
	class10.pdf	27-Mar-2011 10:57	588K	
	class11.pdf	30-Mar-2011 11:07	566K	
	class12.pdf	04-Apr-2011 09:26	436K	
	class13.pdf	06-Apr-2011 12:06	367K	
	class15.pdf	19-Apr-2011 11:28	1.3M	
	class16.pdf	20-Apr-2011 14:14	710K	
	class17.pdf	17-Jan-2012 15:08	940K	
	class19.pdf	08-May-2011 14:02	744K	
	class20.pdf	11-May-2011 10:26	443K	
	class21.pdf	15-May-2011 12:41	757K	
	class22.pdf	18-May-2011 09:59	903K	
	class23-spiral.pdf	23-May-2011 09:44	2.6M	
	m1_32.c	06-Jul-2003 03:31	18K	
	n1_2.c	06-Jul-2003 03:29	2.0K	
	n1_3.c	06-Jul-2003 03:29	2.5K	
	notes02.PDF	22-Feb-2012 16:04	30K	
	notes03.PDF	27-Feb-2011 16:31	29K	
	notes07.pdf	18-Mar-2011 17:17	108K	
	notes08.PDF	27-Mar-2011 11:12	109K	
	notes10.PDF	27-Mar-2011 12:57	61K	

 notes12.PDF	08-Apr-2013 09:32	83K
 notes13.pdf	13-Apr-2011 09:26	174K
 notes16.PDF	18-Apr-2011 13:32	24K
 notes18.PDF	03-May-2011 14:46	51K
 notes19.pdf	11-May-2011 09:21	84K
 notes20.PDF	15-May-2011 13:00	56K
 notes21.PDF	13-May-2013 09:40	84K
 t1_3.c	06-Jul-2003 03:29	2.9K

Apache/2.2.15 (Red Hat) Server at people.inf.ethz.ch Port 443

Index of /markusp/teaching/263-2300-ETH-spring11/slides

	Name	Last modified	Size	Description
	Parent Directory	-		
	t1_3.c	06-Jul-2003 03:29	2.9K	
	notes21.PDF	13-May-2013 09:40	84K	
	notes20.PDF	15-May-2011 13:00	56K	
	notes19.pdf	11-May-2011 09:21	84K	
	notes18.PDF	03-May-2011 14:46	51K	
	notes16.PDF	18-Apr-2011 13:32	24K	
	notes13.pdf	13-Apr-2011 09:26	174K	
	notes12.PDF	08-Apr-2013 09:32	83K	
	notes10.PDF	27-Mar-2011 12:57	61K	
	notes08.PDF	27-Mar-2011 11:12	109K	
	notes07.pdf	18-Mar-2011 17:17	108K	
	notes03.PDF	27-Feb-2011 16:31	29K	
	notes02.PDF	22-Feb-2012 16:04	30K	
	n1_3.c	06-Jul-2003 03:29	2.5K	
	n1_2.c	06-Jul-2003 03:29	2.0K	
	m1_32.c	06-Jul-2003 03:31	18K	
	class23-spiral.pdf	23-May-2011 09:44	2.6M	
	class22.pdf	18-May-2011 09:59	903K	
	class21.pdf	15-May-2011 12:41	757K	
	class20.pdf	11-May-2011 10:26	443K	
	class19.pdf	08-May-2011 14:02	744K	
	class17.pdf	17-Jan-2012 15:08	940K	
	class16.pdf	20-Apr-2011 14:14	710K	
	class15.pdf	19-Apr-2011 11:28	1.3M	
	class13.pdf	06-Apr-2011 12:06	367K	
	class12.pdf	04-Apr-2011 09:26	436K	
	class11.pdf	30-Mar-2011 11:07	566K	
	class10.pdf	27-Mar-2011 10:57	588K	
	class08.pdf	21-Mar-2011 09:44	570K	
	class07.pdf	25-Mar-2011 11:20	666K	

 class06.pdf	19-Mar-2011 15:11 555K
 class05.pdf	09-Mar-2011 11:10 559K
 class04.pdf	08-Mar-2011 10:06 698K
 class03.pdf	27-Feb-2011 16:06 586K
 class02.pdf	25-Feb-2011 11:40 585K
 class01.pdf	21-Feb-2011 15:24 1.9M
 blasqr.pdf	21-Mar-2011 09:01 95K
 arch.pdf	04-Mar-2011 16:08 380K

Apache/2.2.15 (Red Hat) Server at people.inf.ethz.ch Port 443

Index of /markusp/teaching/263-2300-ETH-spring11/slides

	Name	Last modified	Size	Description
	Parent Directory	-		
	n1_2.c	06-Jul-2003 03:29	2.0K	
	n1_3.c	06-Jul-2003 03:29	2.5K	
	t1_3.c	06-Jul-2003 03:29	2.9K	
	m1_32.c	06-Jul-2003 03:31	18K	
	class01.pdf	21-Feb-2011 15:24	1.9M	
	class02.pdf	25-Feb-2011 11:40	585K	
	class03.pdf	27-Feb-2011 16:06	586K	
	notes03.PDF	27-Feb-2011 16:31	29K	
	arch.pdf	04-Mar-2011 16:08	380K	
	class04.pdf	08-Mar-2011 10:06	698K	
	class05.pdf	09-Mar-2011 11:10	559K	
	notes07.pdf	18-Mar-2011 17:17	108K	
	class06.pdf	19-Mar-2011 15:11	555K	
	blasqr.pdf	21-Mar-2011 09:01	95K	
	class08.pdf	21-Mar-2011 09:44	570K	
	class07.pdf	25-Mar-2011 11:20	666K	
	class10.pdf	27-Mar-2011 10:57	588K	
	notes08.PDF	27-Mar-2011 11:12	109K	
	notes10.PDF	27-Mar-2011 12:57	61K	
	class11.pdf	30-Mar-2011 11:07	566K	
	class12.pdf	04-Apr-2011 09:26	436K	
	class13.pdf	06-Apr-2011 12:06	367K	
	notes13.pdf	13-Apr-2011 09:26	174K	
	notes16.PDF	18-Apr-2011 13:32	24K	
	class15.pdf	19-Apr-2011 11:28	1.3M	
	class16.pdf	20-Apr-2011 14:14	710K	
	notes18.PDF	03-May-2011 14:46	51K	
	class19.pdf	08-May-2011 14:02	744K	
	notes19.pdf	11-May-2011 09:21	84K	
	class20.pdf	11-May-2011 10:26	443K	

 class21.pdf	15-May-2011 12:41 757K
 notes20.PDF	15-May-2011 13:00 56K
 class22.pdf	18-May-2011 09:59 903K
 class23-spiral.pdf	23-May-2011 09:44 2.6M
 class17.pdf	17-Jan-2012 15:08 940K
 notes02.PDF	22-Feb-2012 16:04 30K
 notes12.PDF	08-Apr-2013 09:32 83K
 notes21.PDF	13-May-2013 09:40 84K

Apache/2.2.15 (Red Hat) Server at people.inf.ethz.ch Port 443

Index of /markusp/teaching/263-2300-ETH-spring11/slides

	Name	Last modified	Size	Description
	Parent Directory	-		
	n1_2.c	06-Jul-2003 03:29	2.0K	
	n1_3.c	06-Jul-2003 03:29	2.5K	
	t1_3.c	06-Jul-2003 03:29	2.9K	
	m1_32.c	06-Jul-2003 03:31	18K	
	notes16.PDF	18-Apr-2011 13:32	24K	
	notes03.PDF	27-Feb-2011 16:31	29K	
	notes02.PDF	22-Feb-2012 16:04	30K	
	notes18.PDF	03-May-2011 14:46	51K	
	notes20.PDF	15-May-2011 13:00	56K	
	notes10.PDF	27-Mar-2011 12:57	61K	
	notes12.PDF	08-Apr-2013 09:32	83K	
	notes21.PDF	13-May-2013 09:40	84K	
	notes19.pdf	11-May-2011 09:21	84K	
	blasqr.pdf	21-Mar-2011 09:01	95K	
	notes07.pdf	18-Mar-2011 17:17	108K	
	notes08.PDF	27-Mar-2011 11:12	109K	
	notes13.pdf	13-Apr-2011 09:26	174K	
	class13.pdf	06-Apr-2011 12:06	367K	
	arch.pdf	04-Mar-2011 16:08	380K	
	class12.pdf	04-Apr-2011 09:26	436K	
	class20.pdf	11-May-2011 10:26	443K	
	class06.pdf	19-Mar-2011 15:11	555K	
	class05.pdf	09-Mar-2011 11:10	559K	
	class11.pdf	30-Mar-2011 11:07	566K	
	class08.pdf	21-Mar-2011 09:44	570K	
	class02.pdf	25-Feb-2011 11:40	585K	
	class03.pdf	27-Feb-2011 16:06	586K	
	class10.pdf	27-Mar-2011 10:57	588K	
	class07.pdf	25-Mar-2011 11:20	666K	
	class04.pdf	08-Mar-2011 10:06	698K	

 class16.pdf	20-Apr-2011 14:14 710K
 class19.pdf	08-May-2011 14:02 744K
 class21.pdf	15-May-2011 12:41 757K
 class22.pdf	18-May-2011 09:59 903K
 class17.pdf	17-Jan-2012 15:08 940K
 class15.pdf	19-Apr-2011 11:28 1.3M
 class01.pdf	21-Feb-2011 15:24 1.9M
 class23-spiral.pdf	23-May-2011 09:44 2.6M

Apache/2.2.15 (Red Hat) Server at people.inf.ethz.ch Port 443

Index of /markusp/teaching/263-2300-ETH-spring11/slides

	Name	Last modified	Size	Description
	Parent Directory	-		
	arch.pdf	04-Mar-2011 16:08	380K	
	blasqr.pdf	21-Mar-2011 09:01	95K	
	class01.pdf	21-Feb-2011 15:24	1.9M	
	class02.pdf	25-Feb-2011 11:40	585K	
	class03.pdf	27-Feb-2011 16:06	586K	
	class04.pdf	08-Mar-2011 10:06	698K	
	class05.pdf	09-Mar-2011 11:10	559K	
	class06.pdf	19-Mar-2011 15:11	555K	
	class07.pdf	25-Mar-2011 11:20	666K	
	class08.pdf	21-Mar-2011 09:44	570K	
	class10.pdf	27-Mar-2011 10:57	588K	
	class11.pdf	30-Mar-2011 11:07	566K	
	class12.pdf	04-Apr-2011 09:26	436K	
	class13.pdf	06-Apr-2011 12:06	367K	
	class15.pdf	19-Apr-2011 11:28	1.3M	
	class16.pdf	20-Apr-2011 14:14	710K	
	class17.pdf	17-Jan-2012 15:08	940K	
	class19.pdf	08-May-2011 14:02	744K	
	class20.pdf	11-May-2011 10:26	443K	
	class21.pdf	15-May-2011 12:41	757K	
	class22.pdf	18-May-2011 09:59	903K	
	class23-spiral.pdf	23-May-2011 09:44	2.6M	
	m1_32.c	06-Jul-2003 03:31	18K	
	n1_2.c	06-Jul-2003 03:29	2.0K	
	n1_3.c	06-Jul-2003 03:29	2.5K	
	notes02.PDF	22-Feb-2012 16:04	30K	
	notes03.PDF	27-Feb-2011 16:31	29K	
	notes07.pdf	18-Mar-2011 17:17	108K	
	notes08.PDF	27-Mar-2011 11:12	109K	
	notes10.PDF	27-Mar-2011 12:57	61K	

 notes12.PDF	08-Apr-2013 09:32	83K
 notes13.pdf	13-Apr-2011 09:26	174K
 notes16.PDF	18-Apr-2011 13:32	24K
 notes18.PDF	03-May-2011 14:46	51K
 notes19.pdf	11-May-2011 09:21	84K
 notes20.PDF	15-May-2011 13:00	56K
 notes21.PDF	13-May-2013 09:40	84K
 t1_3.c	06-Jul-2003 03:29	2.9K

Apache/2.2.15 (Red Hat) Server at people.inf.ethz.ch Port 443

Index of /markusp/teaching/263-2300-ETH-spring11

	Name	Last modified	Size	Description
	Parent Directory		-	
	course.html	12-Oct-2016 17:42	30K	
	homeworks/	05-May-2011 16:04	-	
	midterm/	14-Apr-2016 14:56	-	
	project/	18-May-2011 11:04	-	
	slides/	23-May-2011 09:45	-	

Apache/2.2.15 (Red Hat) Server at people.inf.ethz.ch Port 443

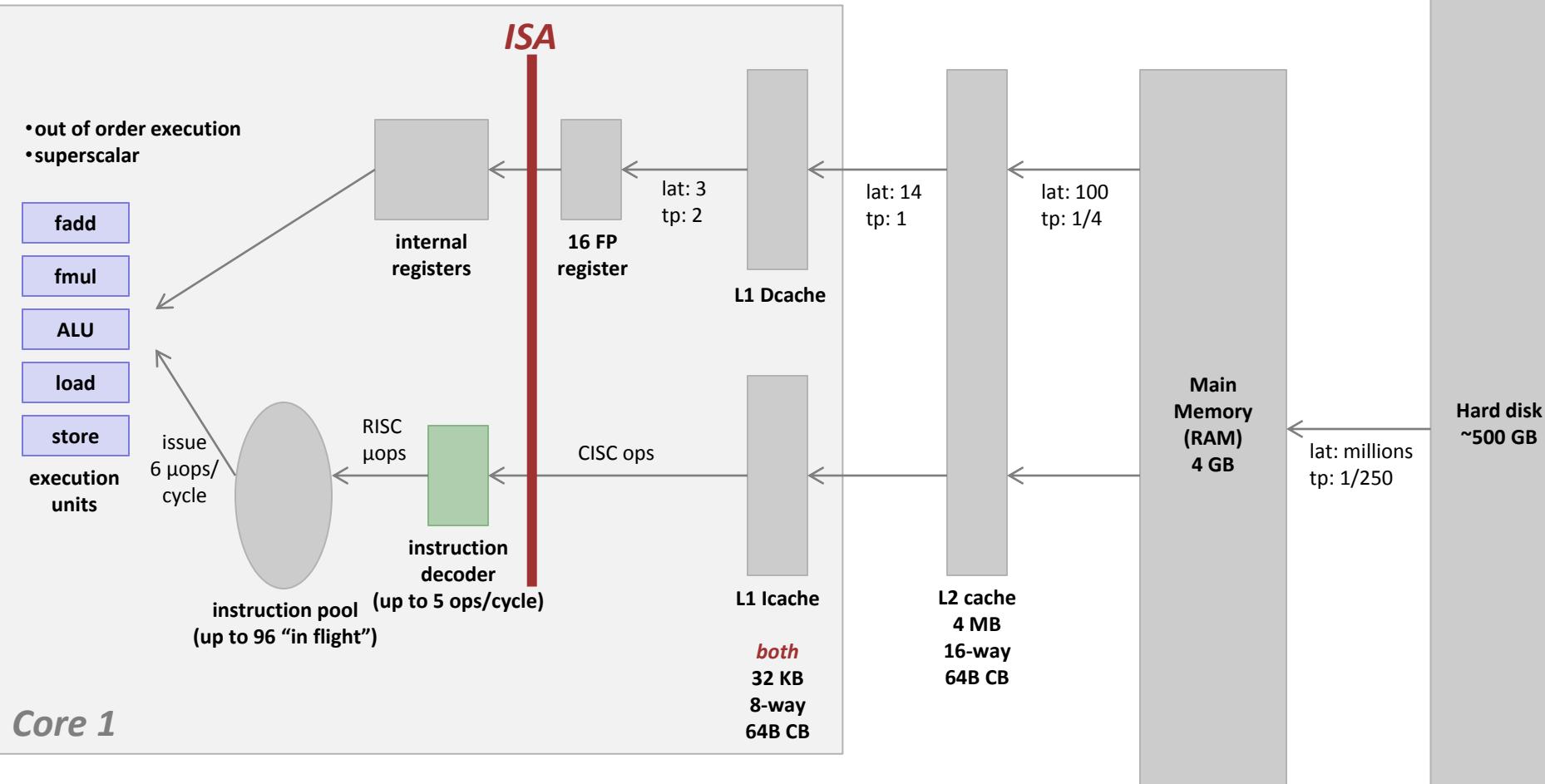
Abstracted Microarchitecture: Example Core (2008)

Throughput is measured in doubles/cycle

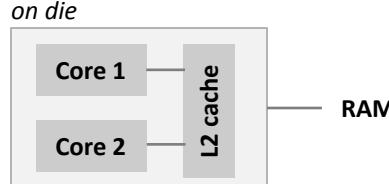
Latency in cycles for one double

1 double = 8 bytes

Rectangles not to scale



Core 2 Duo:



Memory hierarchy:

- Registers
- L1 cache
- L2 cache
- Main memory
- Hard disk

Level 1 BLAS

```

dim scalar vector    vector    scalars      5-element array
SUBROUTINE xROTG (          A, B, C, S )
SUBROUTINE xROTMG(          D1, D2, A, B,   PARAM )
SUBROUTINE xROT ( N,         X, INCX, Y, INCY,   C, S )
SUBROUTINE xROTM ( N,         X, INCX, Y, INCY,   PARAM )
SUBROUTINE xSWAP ( N,         X, INCX, Y, INCY )
SUBROUTINE xSCAL ( N, ALPHA, X, INCX )
SUBROUTINE xCOPY ( N,         X, INCX, Y, INCY )
SUBROUTINE xAXPY ( N, ALPHA, X, INCX, Y, INCY )
FUNCTION  xDOT ( N,         X, INCX, Y, INCY )
FUNCTION  xDOTU ( N,        X, INCX, Y, INCY )
FUNCTION  xDOTC ( N,        X, INCX, Y, INCY )
FUNCTION  xxDOT ( N,        X, INCX, Y, INCY )
FUNCTION  xNRM2 ( N,        X, INCX )
FUNCTION  xASUM ( N,        X, INCX )
FUNCTION  IxAMAX( N,        X, INCX )

```

Level 2 BLAS

options	dim	b-width	scalar	matrix	vector	scalar	vector
xGEMV (TRANS,	M, N,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xGBMV (TRANS,	M, N, KL, KU,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xHEMV (UPLO,	N,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xHBMV (UPLO,	N, K,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xHPMV (UPLO,	N,		ALPHA, AP, X, INCX, BETA, Y, INCY)				
xSYMV (UPLO,	N,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xSMBMV (UPLO,	N, K,		ALPHA, A, LDA, X, INCX, BETA, Y, INCY)				
xSPMV (UPLO,	N,		ALPHA, AP, X, INCX, BETA, Y, INCY)				
xTRMV (UPLO, TRANS, DIAG,	N,		A, LDA, X, INCX)				
xTBMV (UPLO, TRANS, DIAG,	N, K,		A, LDA, X, INCX)				
xTPMV (UPLO, TRANS, DIAG,	N,		AP, X, INCX)				
xTRSV (UPLO, TRANS, DIAG,	N,		A, LDA, X, INCX)				
xTBSV (UPLO, TRANS, DIAG,	N, K,		A, LDA, X, INCX)				
xTPSV (UPLO, TRANS, DIAG,	N,		AP, X, INCX)				
options	dim	scalar	vector	vector	matrix		
xGER (M, N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xGERU (M, N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xGERC (M, N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xHER (UPLO,	N,	ALPHA, X, INCX,			A, LDA)		
xHPR (UPLO,	N,	ALPHA, X, INCX,			AP)		
xHER2 (UPLO,	N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xHPR2 (UPLO,	N,	ALPHA, X, INCX, Y, INCY, AP)					
xSYR (UPLO,	N,	ALPHA, X, INCX,			A, LDA)		
xSPR (UPLO,	N,	ALPHA, X, INCX,			AP)		
xSYR2 (UPLO,	N,	ALPHA, X, INCX, Y, INCY, A, LDA)					
xSPR2 (UPLO,	N,	ALPHA, X, INCX, Y, INCY, AP)					

Level 3 BLAS

options		dim	scalar	matrix	matrix	scalar	matrix
xGEMM (TRANSA, TRANSB,	M, N, K,	ALPHA, A,	LDA, B,	LDB, BETA,	C,	LDC)
xSYMM (SIDE, UPLO,	M, N,	ALPHA, A,	LDA, B,	LDB, BETA,	C,	LDC)
xHEMM (SIDE, UPLO,	M, N,	ALPHA, A,	LDA, B,	LDB, BETA,	C,	LDC)
xSYRK (UPLO, TRANS,	N, K,	ALPHA, A,	LDA,	BETA,	C,	LDC)
xHERK (UPLO, TRANS,	N, K,	ALPHA, A,	LDA,	BETA,	C,	LDC)
xSYR2K(UPLO, TRANS,	N, K,	ALPHA, A,	LDA, B,	LDB, BETA,	C,	LDC)
xHER2K(UPLO, TRANS,	N, K,	ALPHA, A,	LDA, B,	LDB, BETA,	C,	LDC)
xTRMM (SIDE, UPLO, TRANS,	DIAG, M, N,	ALPHA, A,	LDA, B,	LDB)		
xTRSM (SIDE, UPLO, TRANS,	DIAG, M, N,	ALPHA, A,	LDA, B,	LDB)		

```

Generate plane rotation
Generate modified plane rotation
Apply plane rotation
Apply modified plane rotation
 $x \leftrightarrow y$ 
 $x \leftarrow \alpha x$ 
 $y \leftarrow x$ 
 $y \leftarrow \alpha x + y$ 
 $dot \leftarrow x^T y$ 
 $dot \leftarrow x^T y$ 
 $dot \leftarrow x^H y$ 
 $dot \leftarrow \alpha + x^T y$ 
 $nrm2 \leftarrow \|x\|_2$ 
 $asum \leftarrow |re(x)|_1 + |im(x)|_1$ 
 $amax \leftarrow 1^{st} k \ni |re(x_k)| + |im(x_k)|$ 
 $= max(|re(x_i)| + |im(x_i)|)$ 

```

```

 $y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$ 
 $y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$ 
 $y \leftarrow \alpha Ax + \beta y$ 
 $x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$ 
 $x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$ 
 $x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$ 
 $x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$ 
 $x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$ 
 $x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$ 

```

$$\begin{aligned}
A &\leftarrow \alpha x y^T + A, A - m \times n \\
A &\leftarrow \alpha x y^T + A, A - m \times n \\
A &\leftarrow \alpha x y^H + A, A - m \times n \\
A &\leftarrow \alpha x x^H + A \\
A &\leftarrow \alpha x x^H + A \\
A &\leftarrow \alpha x y^H + y(\alpha x)^H + A \\
A &\leftarrow \alpha x y^H + y(\alpha x)^H + A \\
A &\leftarrow \alpha x x^T + A \\
A &\leftarrow \alpha x x^T + A \\
A &\leftarrow \alpha x y^T + \alpha y x^T + A \\
A &\leftarrow \alpha x y^T + \alpha y x^T + A
\end{aligned}$$

$$\begin{aligned}
C &\leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n \\
C &\leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T \\
C &\leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H \\
C &\leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n \\
C &\leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n \\
C &\leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha AT^B + \bar{\alpha} BT^A + \beta C, C - n \times n \\
C &\leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha AH^B + \bar{\alpha} BH^A + \beta C, C - n \times n \\
B &\leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n \\
B &\leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B - m \times n
\end{aligned}$$

prefixes
S, D
S, D
S, D
S, D
S, D, C, Z
S, D, C, Z, CS, ZD
S, D, C, Z
S, D, C, Z
S, D, DS
C, Z
C, Z
SDS
S, D, SC, DZ
S, D, SC, DZ
S, D, C, Z

S, D, C, Z
S, D, C, Z
C, Z
C, Z
C, Z
S, D
S, D
S, D
S, D, C, Z
S, D, C, Z

S, D
C, Z
C, Z
C, Z
C, Z
C, Z
C, Z
S, D
S, D
S, D
S, D

Meaning of prefixes

S - REAL	C - COMPLEX
D - DOUBLE PRECISION	Z - COMPLEX*16 (this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS

In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS

Matrix types:

GE - GEneral	GB - General Band	
SY - SYmmetric	SB - Sym. Band	SP - Sum. Packed
HE - HERmitian	HB - Herm. Band	HP - Herm. Packed
TR - TRiangular	TB - Triang. Band	TP - Triang. Packed

Level 2 and Level 3 BLAS Options

Dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.

TRANx	= 'No transpose', 'Transpose', 'Conjugate transpose' (X, X^T, X^H)
UPLO	= 'Upper triangular', 'Lower triangular'
DIAG	= 'Non-unit triangular', 'Unit triangular'
SIDE	= 'Left', 'Right' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANSx = 'T' and TRANSx = 'C' have the same meaning.

For Hermitian matrices, TRANSx = 'T' is not allowed.

For complex symmetric matrices, TRANSx = 'H' is not allowed.

References

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. on Math. Soft.* 5 (1979) 308-325

J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* 14,1 (1988) 1-32

J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* (1989)

Obtaining the Software via netlib@ornl.gov

To receive a copy of the single-precision software,
type in a mail message:

```
send sblas from blas
send sblas2 from blas
send sblas3 from blas
```

To receive a copy of the double-precision software,
type in a mail message:

```
send dblas from blas
send dblas2 from blas
send dblas3 from blas
```

To receive a copy of the complex single-precision software,
type in a mail message:

```
send cblas from blas
send cblas2 from blas
send cblas3 from blas
```

To receive a copy of the complex double-precision software,
type in a mail message:

```
send zblas from blas
send zblas2 from blas
send zblas3 from blas
```

Send comments and questions to lapack@cs.utk.edu.

Basic

Linear

Algebra

Subprograms

A Quick Reference Guide

University of Tennessee
Oak Ridge National Laboratory
Numerical Algorithms Group Ltd.

May 11, 1997

How to Write Fast Numerical Code

Spring 2011, Lecture 1



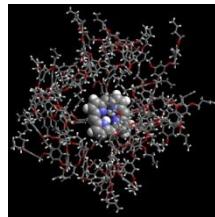
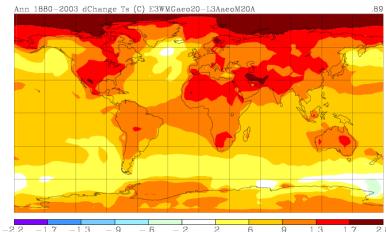
Instructor: *Markus Püschel*

TA: *Georg Ofenbeck*

Today

- **Motivation for this course**
- **Organization of this course**

Scientific Computing



Physics/biology simulations

Computing

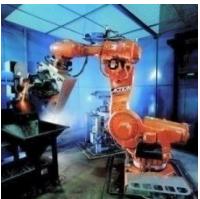
- Unlimited need for performance
- Large set of applications, but ...
- Relatively small set of critical components (100s to 1000s)
 - Matrix multiplication
 - Discrete Fourier transform (DFT)
 - Viterbi decoder
 - Shortest path computation
 - Stencils
 - Solving linear system
 -

Consumer Computing



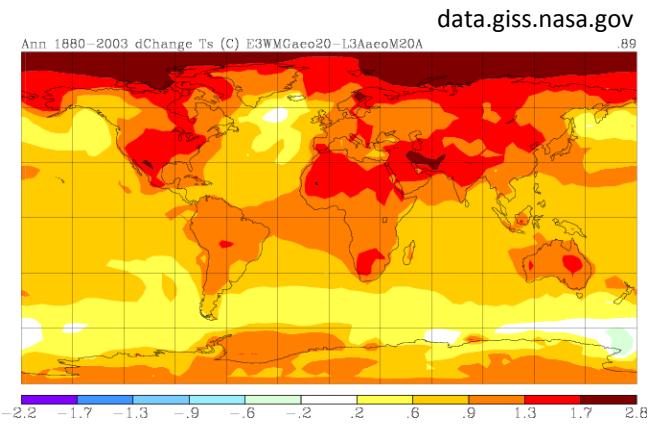
Audio/image/video processing

Embedded Computing



Signal processing, communication, control

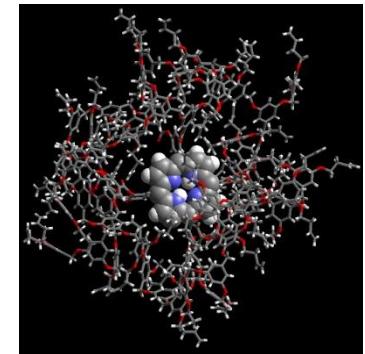
Scientific Computing (Clusters/Supercomputers)



Climate modelling



Finance simulations



Molecular dynamics

Other application areas:

- Fluid dynamics
- Chemistry
- Biology
- Medicine
- Geophysics

Methods:

- Mostly linear algebra
- PDE solving
- Linear system solving
- Finite element methods

Consumer Computing (Desktop, ...)



Photo/video processing



Audio coding



Security



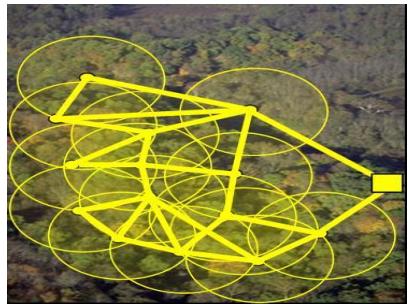
Image compression

Methods:

- Linear algebra
- Transforms
- Filters
- Others

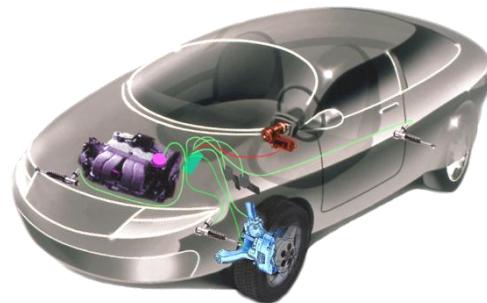
Embedded Computing (Low-power processors)

www.dei.unipd.it



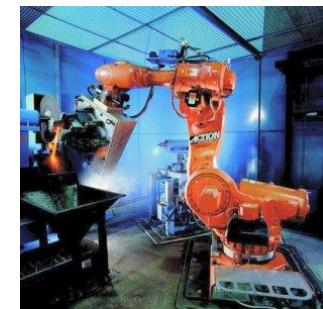
Sensor networks

www.ece.drexel.edu



Cars

www.microway.com.au



Robotics

Computation needed:

- Signal processing
- Control
- Communication

Methods:

- Linear algebra
- Transforms, Filters
- Coding

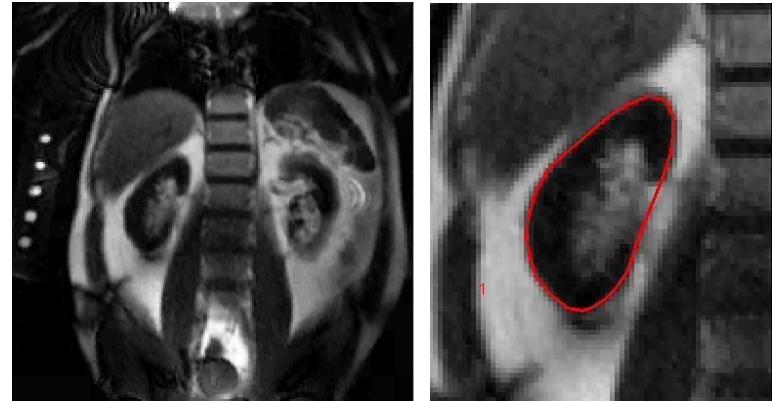
Research (Examples from Carnegie Mellon)

Bhagavatula/Savvides



Biometrics

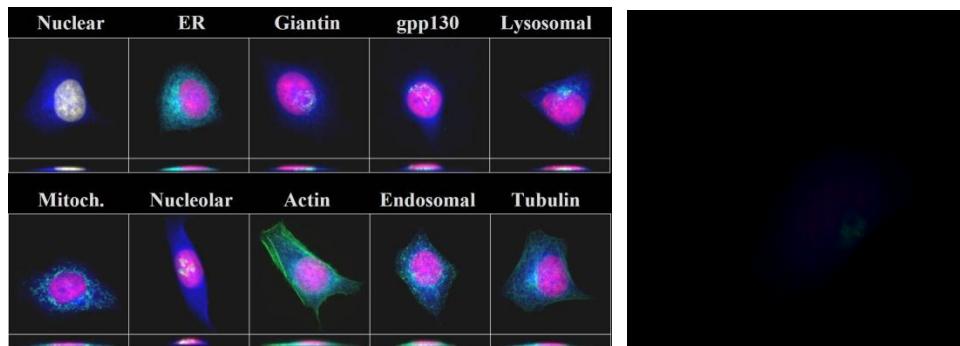
Moura



Medical Imaging

Kanade

Kovacevic



Bioimaging



Computer vision

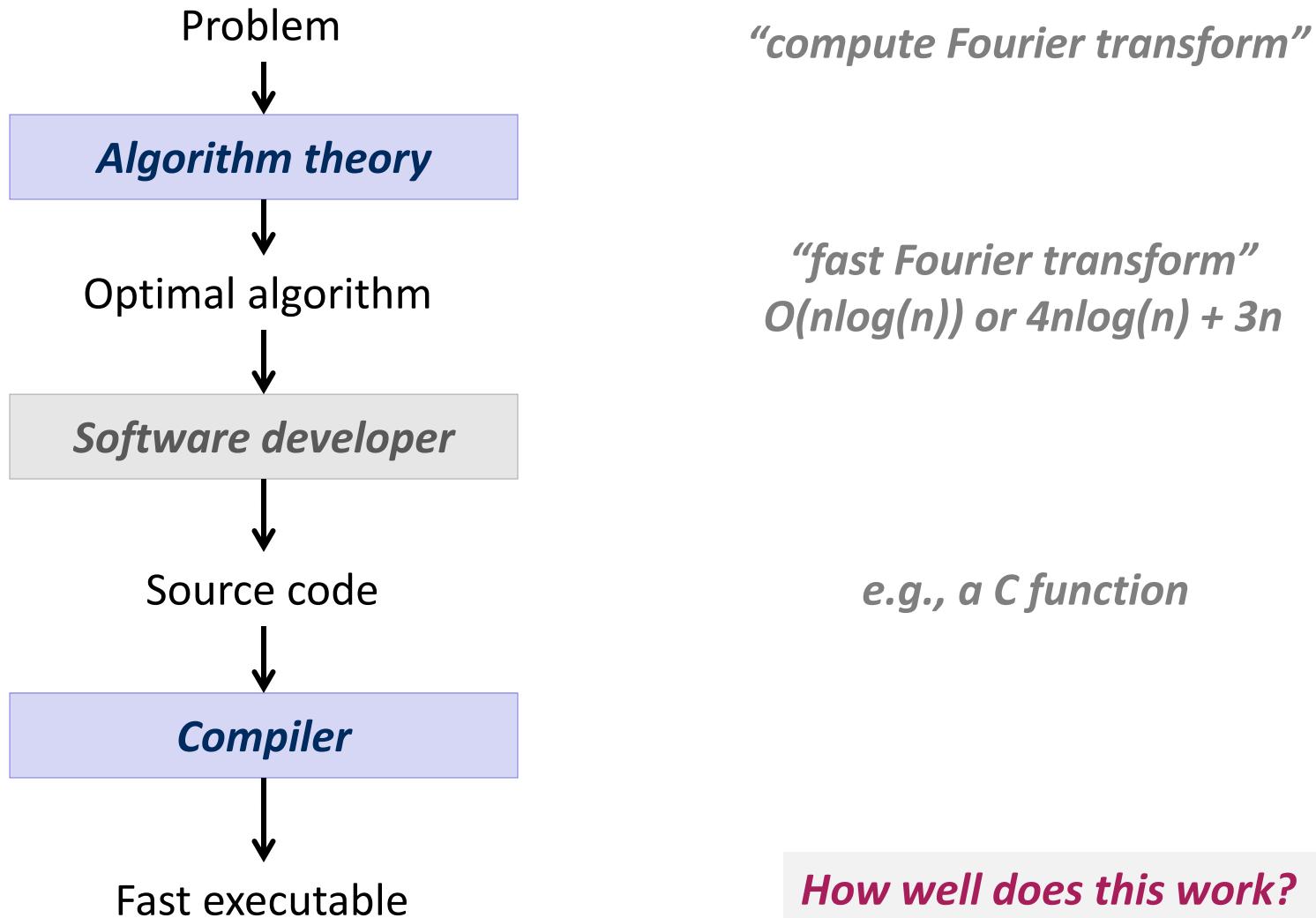
Classes of Performance-Critical Functions

- Transforms
- Filters/correlation/convolution/stencils/interpolators
- Dense linear algebra functions
- Sparse linear algebra functions
- Coder/decoders
- ... *several others*

See also the 13 dwarfs/motifs in

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>

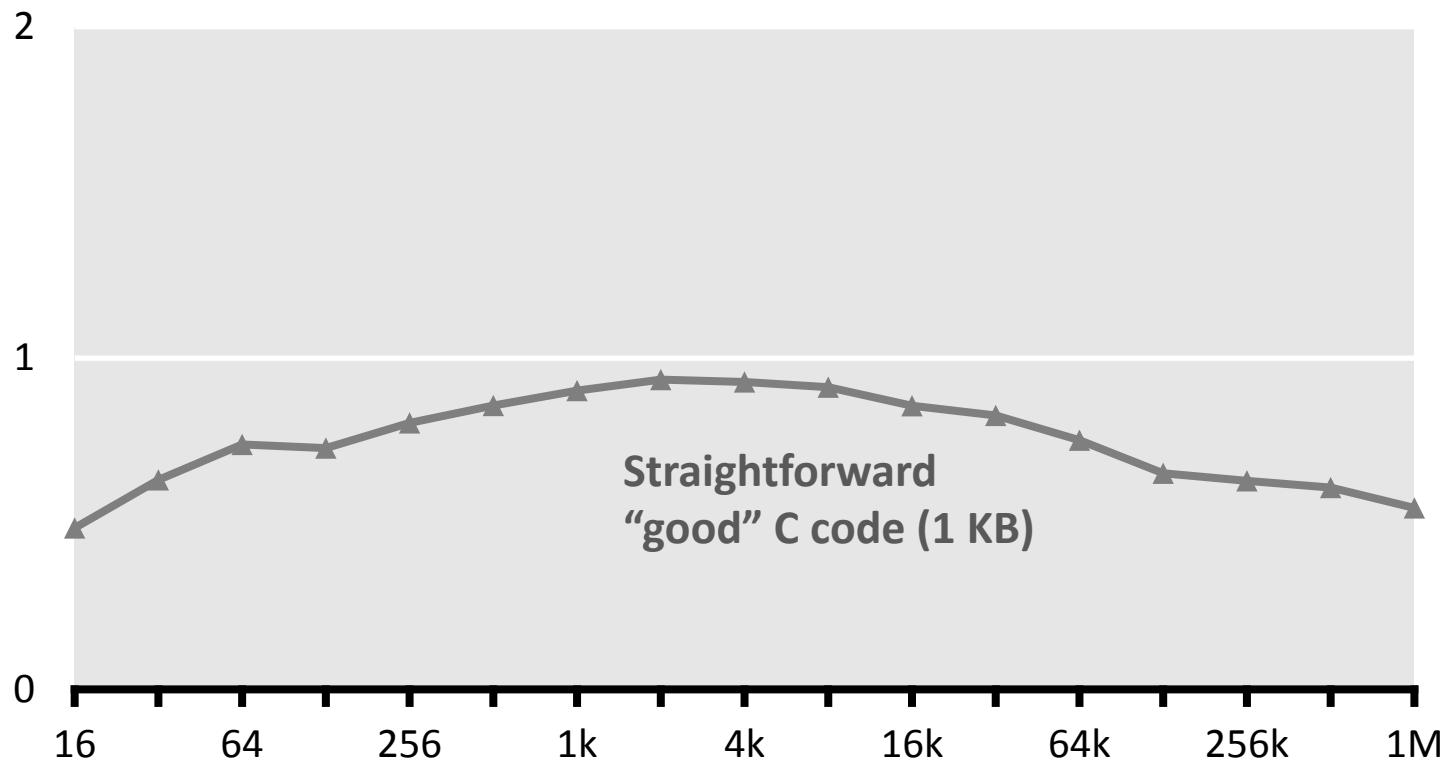
How Hard Is It to Get Fast Code?



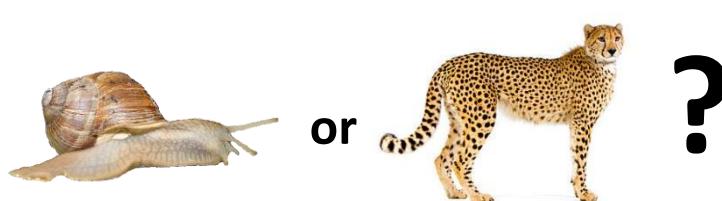
The Problem: Example 1

DFT (single precision) on Intel Core i7 (4 cores, 2.66 GHz)

Performance [Gflop/s]



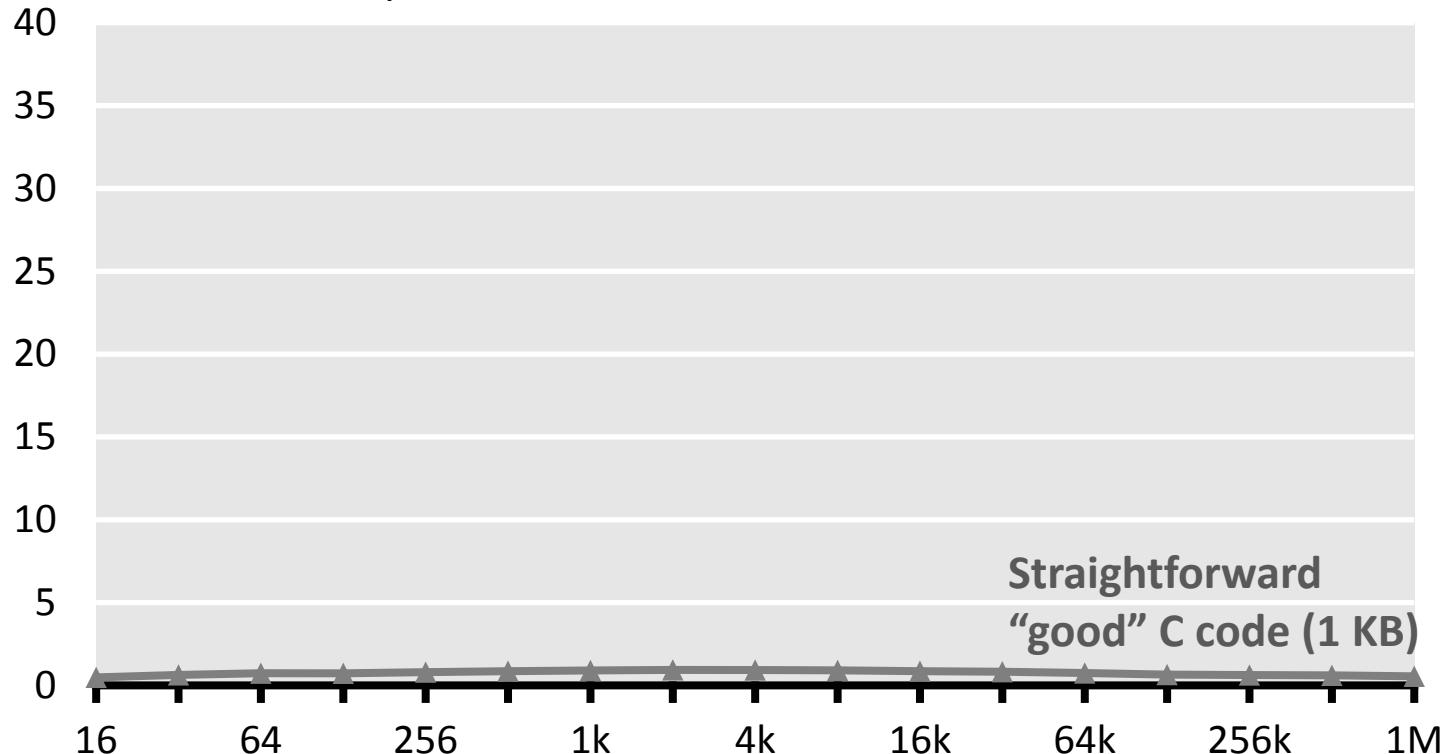
Straightforward
“good” C code (1 KB)



The Problem: Example 1

DFT (single precision) on Intel Core i7 (4 cores, 2.66 GHz)

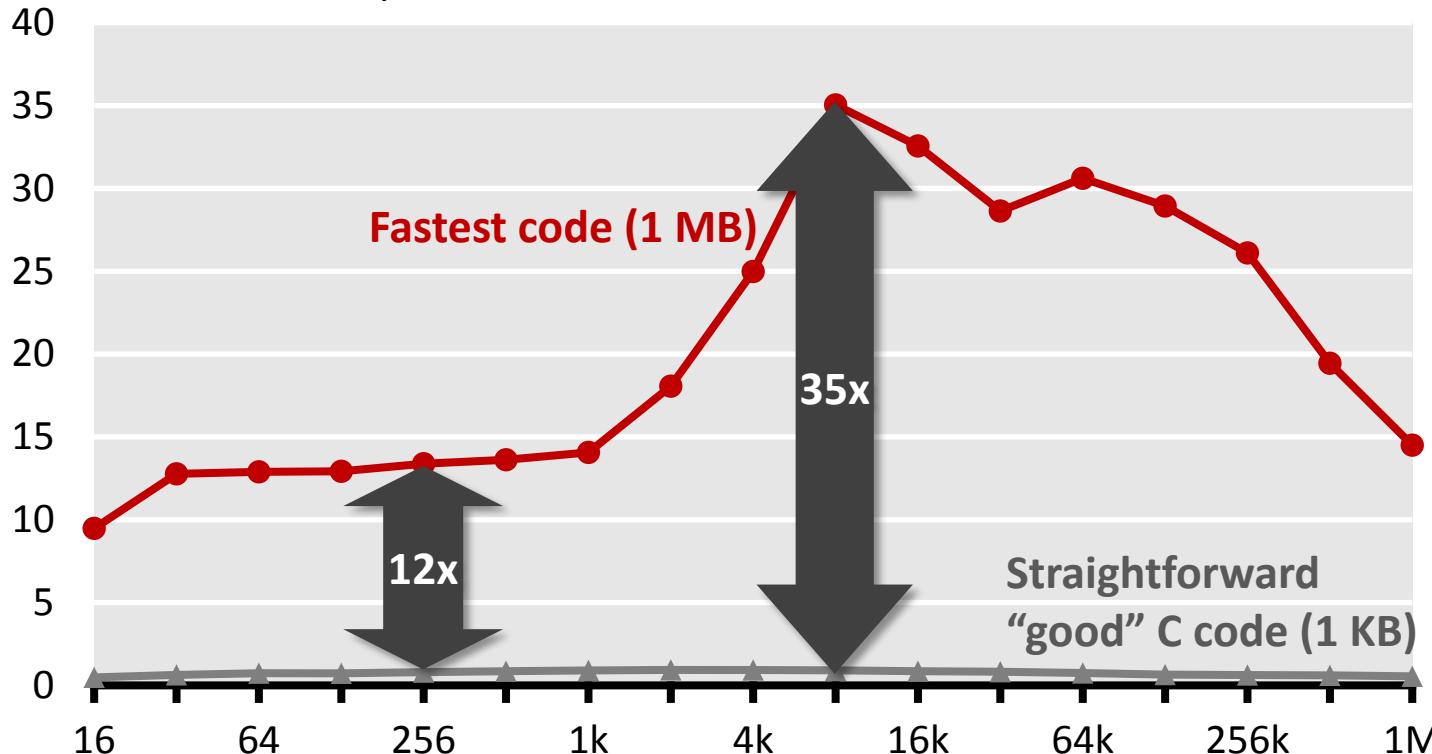
Performance [Gflop/s]



The Problem: Example 1

DFT (single precision) on Intel Core i7 (4 cores, 2.66 GHz)

Performance [Gflop/s]



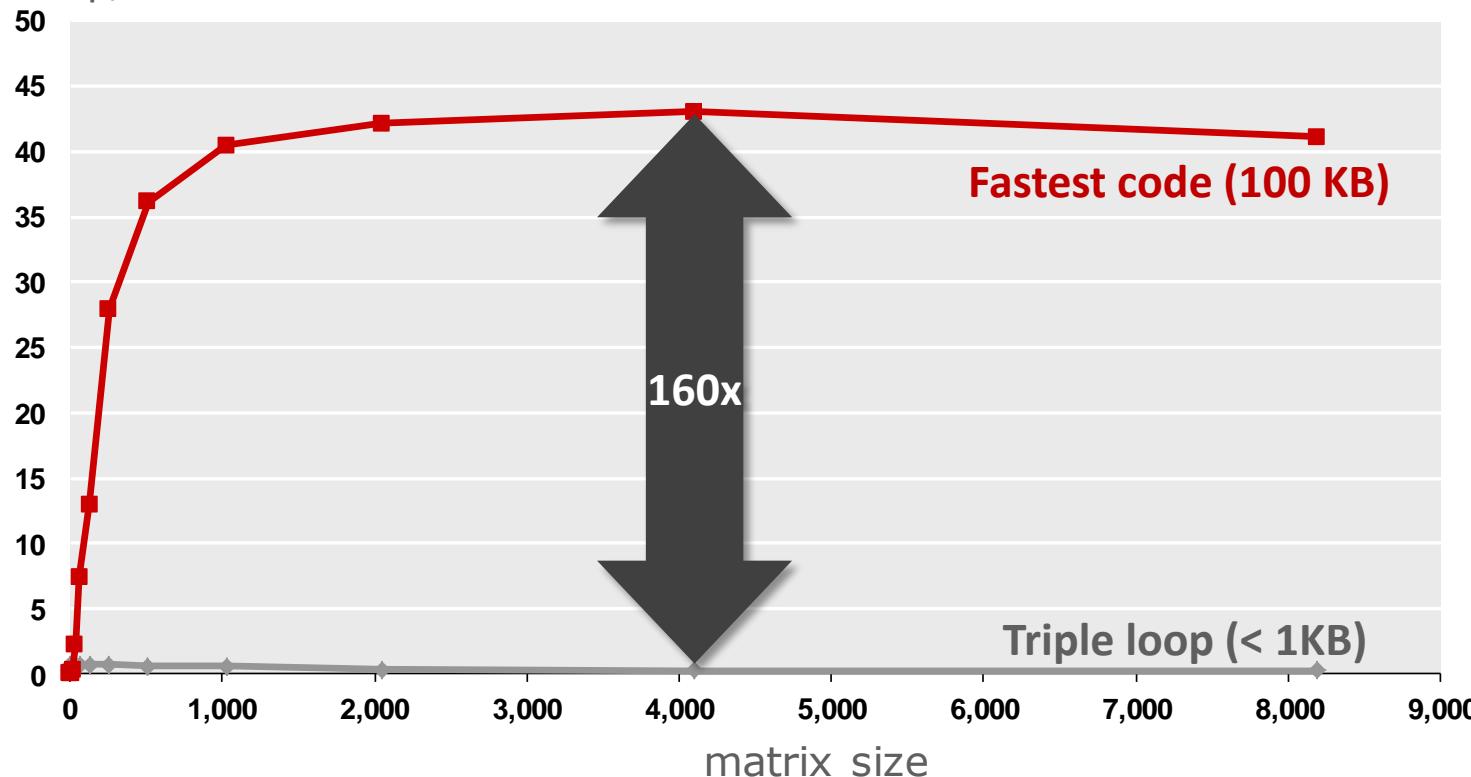
- Vendor compiler, best flags
- Roughly same operations count



The Problem: Example 2

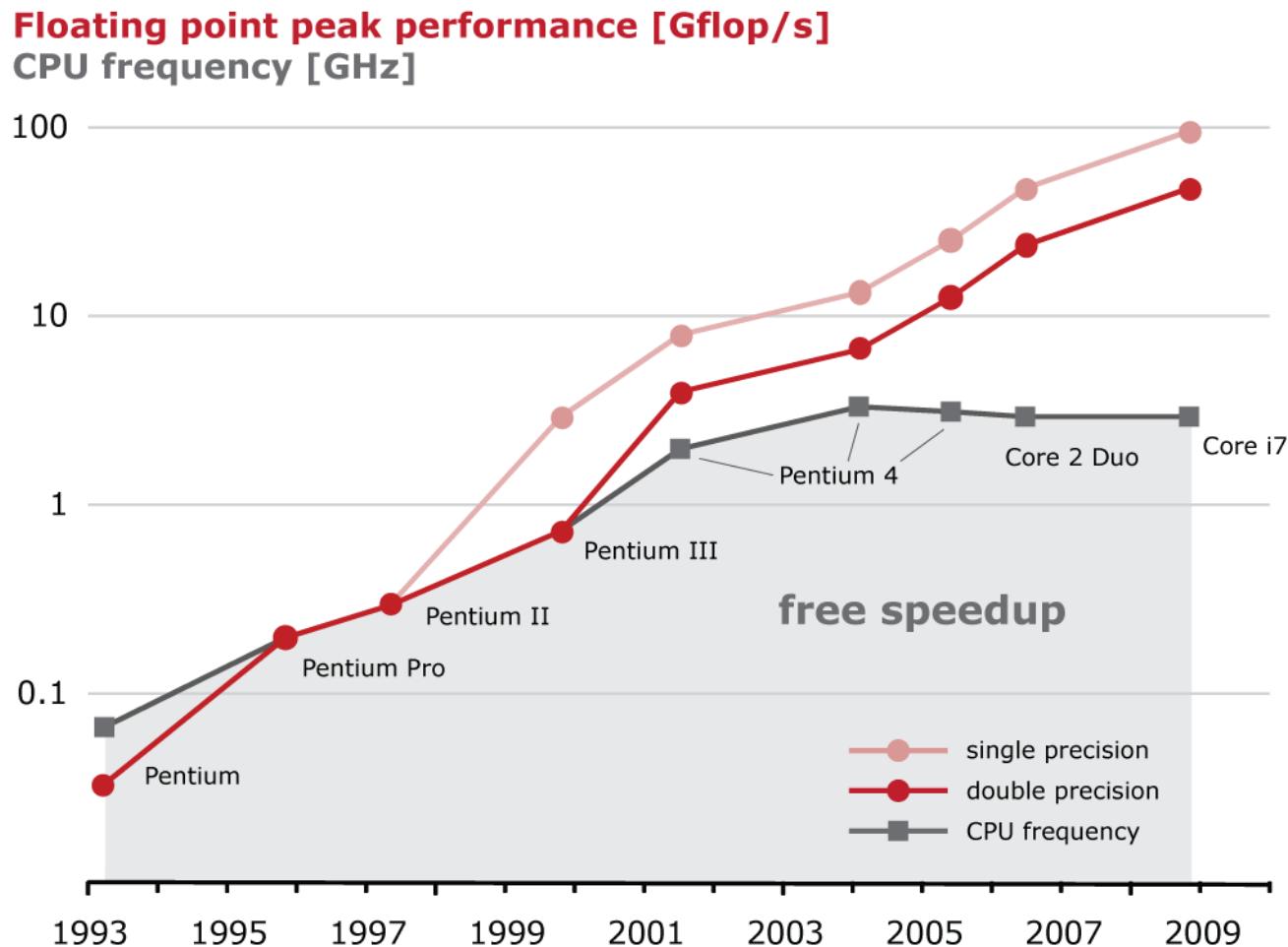
Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

Gflop/s

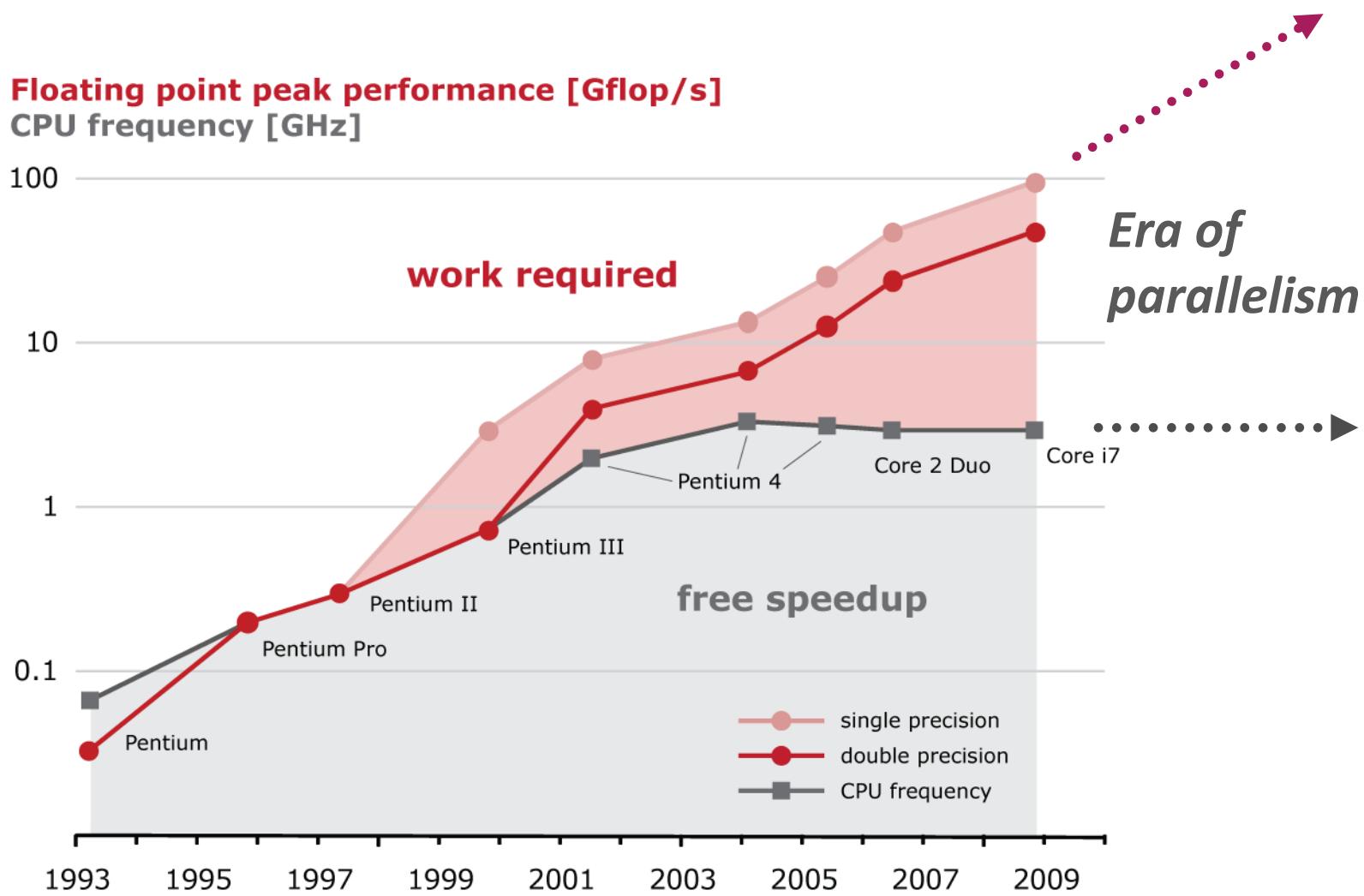


- Vendor compiler, best flags
- Exact same operations count ($2n^3$)
- *What is going on?*

Evolution of Processors (Intel)

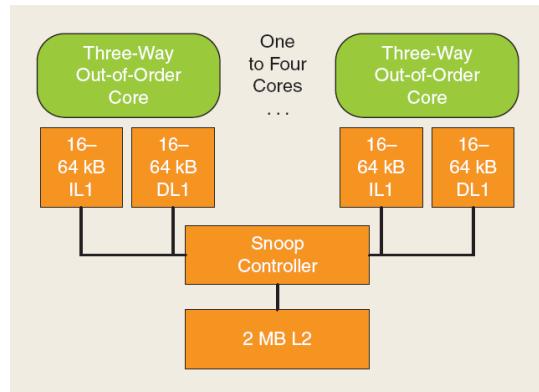


Evolution of Processors (Intel)

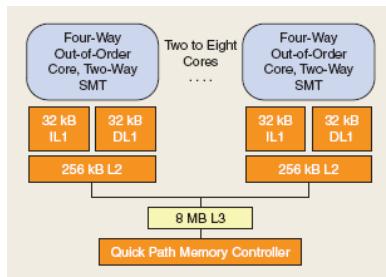


And There Will Be Variety ...

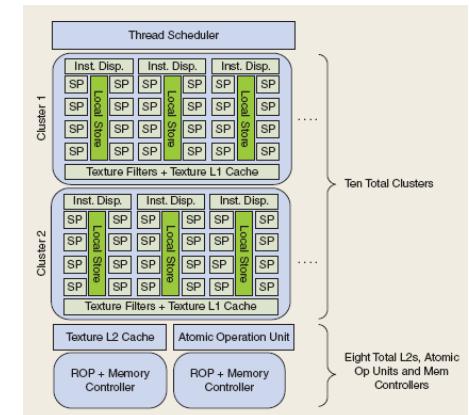
Arm Cortex A9



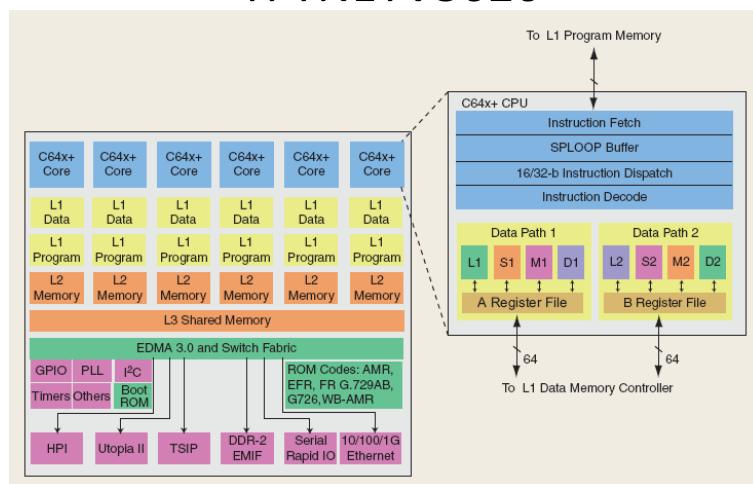
Core i7



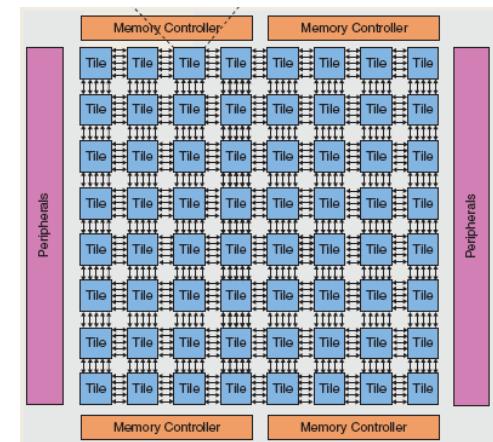
Nvidia G200



TI TNETV3020

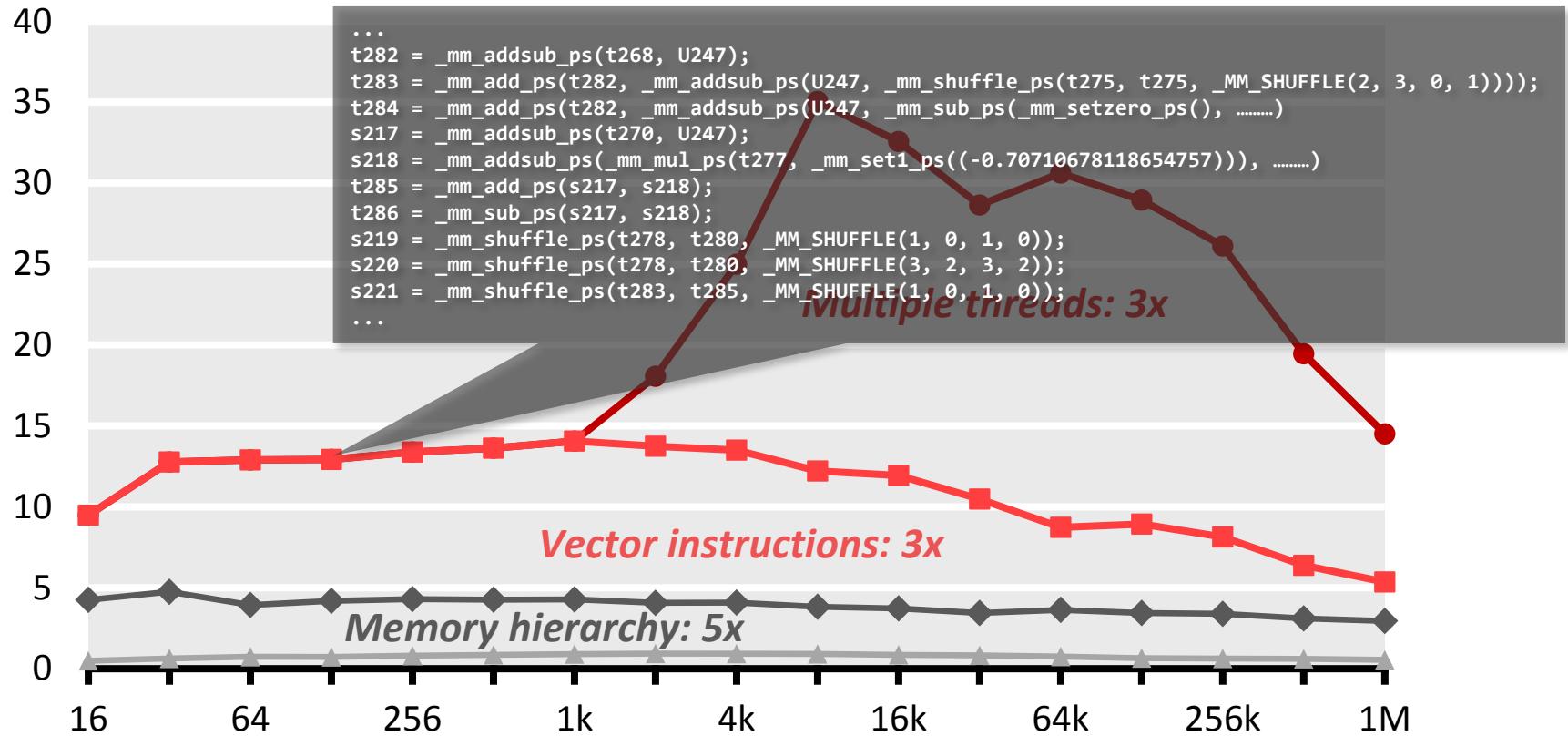


Tilera Tile64



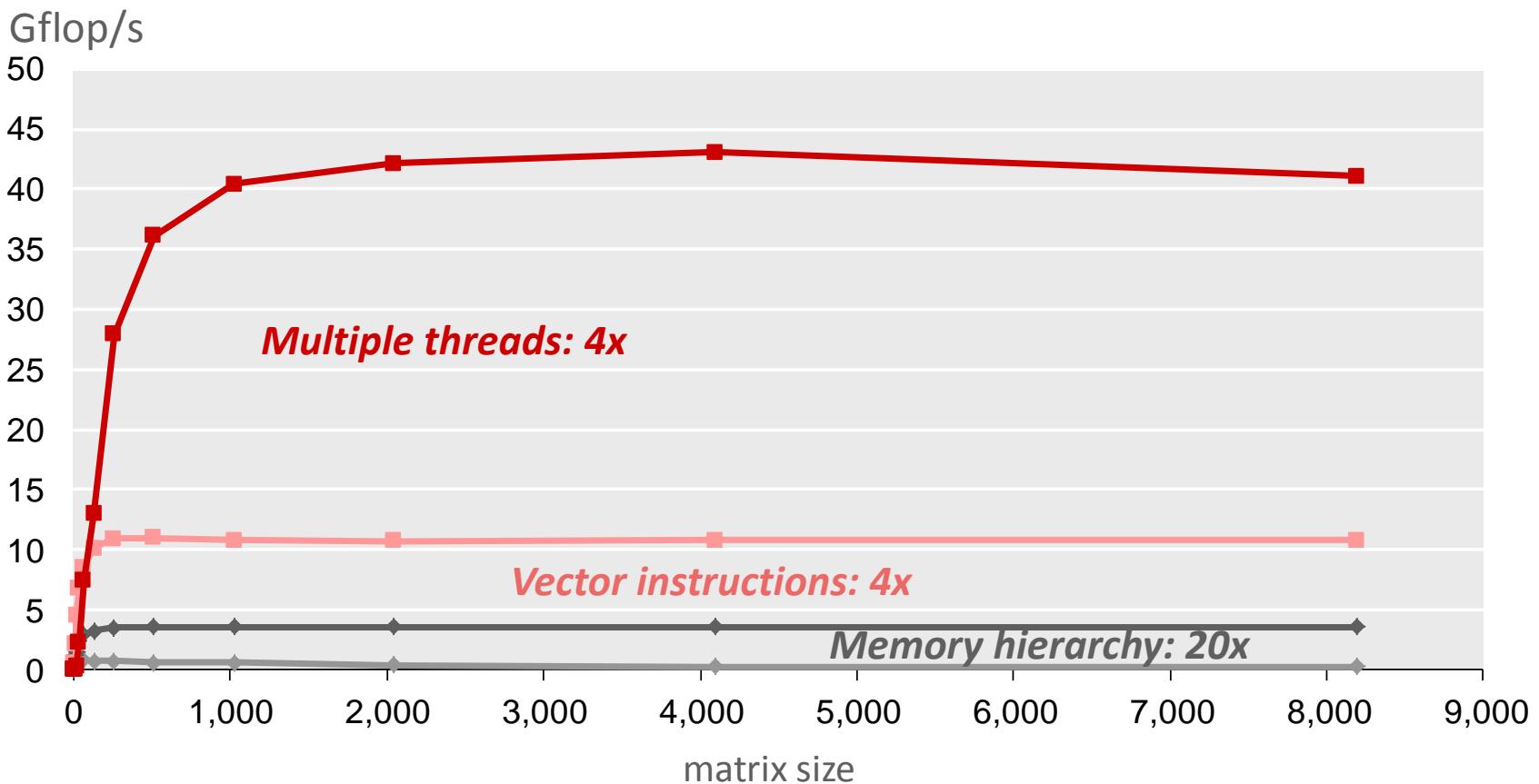
DFT (single precision) on Intel Core i7 (4 cores, 2.66 GHz)

Performance [Gflop/s]



- Compiler doesn't do the job
- Doing by hand: *nightmare*

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz



- Compiler doesn't do the job
- Doing by hand: *nightmare*

Summary and Facts I

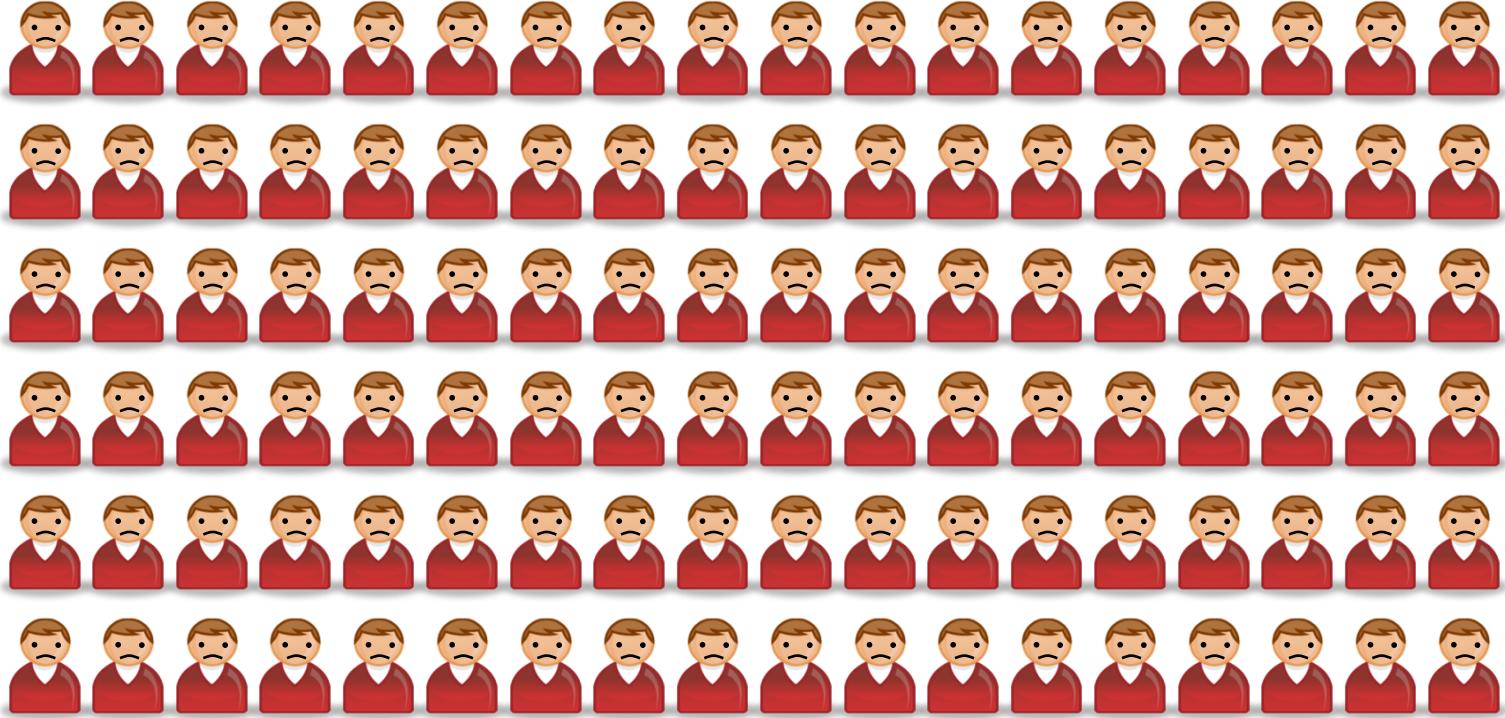
- **Implementations with same operations count can have vastly different performance (up to 100x and more)**
 - A cache miss can be 100x more expensive than an operation
 - Vector instructions
 - Multiple cores = processors on one die
- **Minimizing operations count ≠ maximizing performance**
- **End of free speed-up for legacy code**
 - Future performance gains through increasing parallelism

Summary and Facts II

- **It is very difficult to write the fastest code**
 - Tuning for memory hierarchy
 - Vector instructions
 - Efficient parallelization (multiple threads)
 - Requires expert knowledge in algorithms, coding, and architecture
- **Fast code can be large**
 - Can violate “good” software engineering practices
- **Compilers often can't do the job**
 - Often intricate changes in the algorithm required
 - Parallelization/vectorization still unsolved
- **Highest performance is in general non-portable**

Performance/Productivity Challenge

Current Solution



- *Legions* of programmers implement and optimize the *same* functionality for *every* platform and *whenever* a new platform comes out.

Better Solution: Autotuning

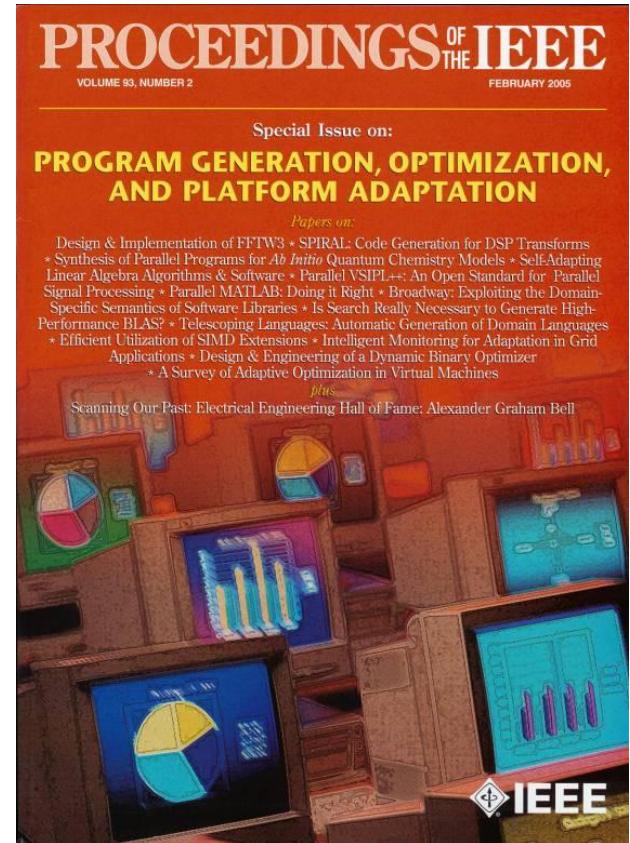
- Automate (parts of) the implementation or optimization



- Research efforts

- Linear algebra: *Phipac/ATLAS*, LAPACK, *Sparsity/Bebop/OSKI*, Flame
- Tensor computations
- PDE/finite elements: Fenics
- Adaptive sorting
- *Fourier transform: FFTW*
- *Linear transforms: Spiral*
- ...others
- New compiler techniques

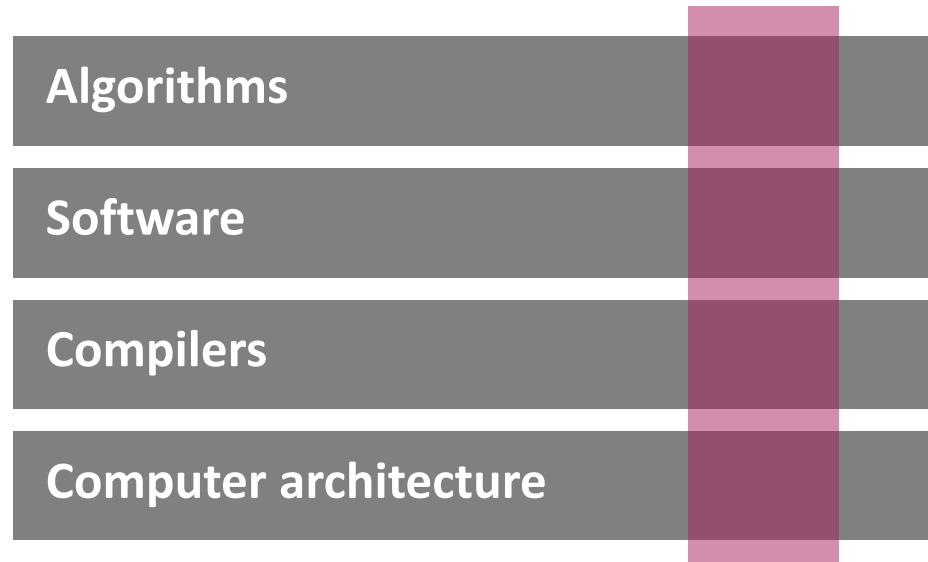
*Promising new area but
much more work needed ...*



Proceedings of the IEEE special issue, Feb. 2005

This Course

*Fast implementations of
numerical problems*



- Obtain an understanding of performance (runtime)
- Learn how to write *fast code* for numerical problems
 - Focus: Memory hierarchy and vector instructions
 - Principles studied using important examples
 - *Applied in homeworks and a semester-long research project*
- Learn about autotuning

Today

- Motivation for this course
- Organization of this course

About this Course

- Team
 - Me
 - TA: Georg Ofenbeck
- Office hours: to be determined
- Email address for any questions: fastcode@lists.inf.ethz.ch
- Course website has *ALL* information



About this Course (cont'd)

■ Requirements

- solid C programming skills
- matrix algebra
- Master student or above

■ Grading

- 40% research project
- 20% midterm exam
- 40% homework

■ Friday slot

- Gives you scheduled time to work together
- Occasionally I will move lecture there

Research Project

- Team up in pairs
- ***Topic:*** Very fast implementation of a numerical problem
- ***Until March 9th:*** suggest to me a problem or I give you a problem
Tip: pick something from your research or that you are interested in
- Show “milestones” during semester
- Write 4 page standard conference paper (template will be provided)
- Give short presentation end of semester
- Submit final code (early semester break)

Midterm Exam

- Some algorithm analysis
 - Memory hierarchy
 - Other
-
- *There is no final exam*

Homework

- **Exercises on algorithm/performance analysis (Math)**
- **Implementation exercises**
 - Concrete numerical problems
 - Study the effect of program optimizations, use of compilers, use of special instructions, etc. (Writing C code + creating runtime/performance plots)
 - Some templates will be provided
 - *Does everybody have access to an Intel processor?*
- **Homework scheduled to leave time for research project**
- **Small part of homework grade for neatness**
- **Late homework policy:**
 - *No deadline extensions*, but
 - 3 late days for the entire semester
 - You can use at most 2 for a homework

Academic Integrity

- Zero tolerance cheating policy (cheat = fail + being reported)
- Homeworks
 - All single-student
 - Don't look at other students code
 - Don't copy code from anywhere
 - Ok to discuss things – but then you have to do it alone
- Code may be checked with tools

Background Material

- Course website
- *Chapter 5 in:*
Computer Systems: A Programmer's Perspective, 2nd edition
Randal E. Bryant and David R. O'Hallaron
(several ones are in the library)
web: <http://csapp.cs.cmu.edu/>
- Prior version of this course:
[spring 2008 at ECE/CMU](#)

Class Participation

- **I'll start on time**
- **It is important to attend**
 - Many things I'll teach are not in books
 - I'll use part slides part blackboard
- **Ask questions**
- **I will provide some anonymous feedback mechanism
(maybe every 3–4 weeks)**

How to Write Fast Numerical Code

Spring 2011
Lecture 2

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

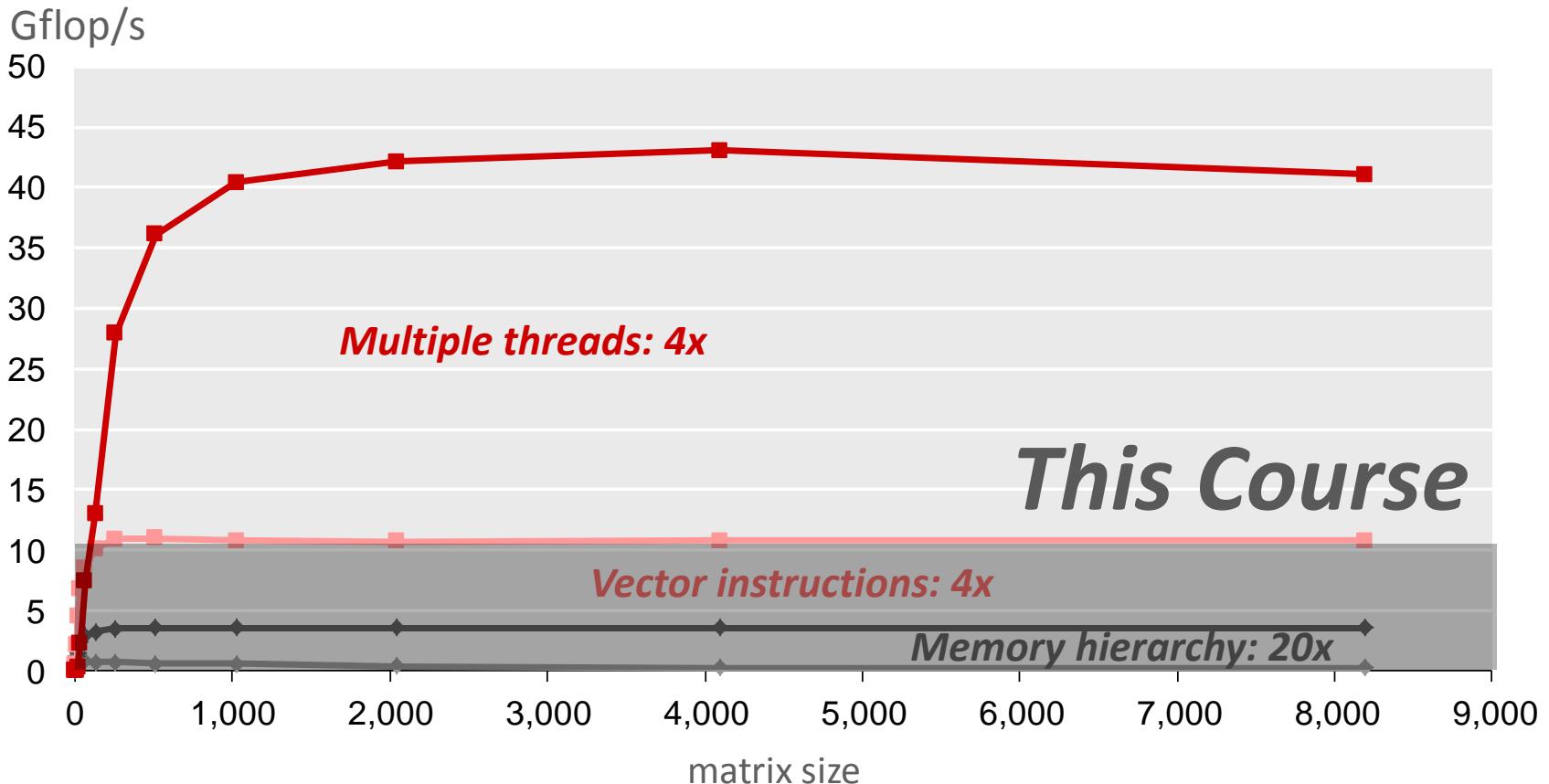
Technicalities

- **Research project: Let me know**
 - if you know with whom you will work
 - if you have already a project idea

- **Deadline: March 9th**

Last Time

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz



Today

- Problem and Algorithm
 - Asymptotic analysis: Do you know the O?
 - Cost analysis
-
- *Standard book:* Introduction to Algorithms (2nd edition), Cormen, Leiserson, Rivest, Stein, McGraw Hill 2001)

Problem

- ***Problem:*** Specification of the relationship between a given input and a desired output
- **Numerical problems: In- and Output are numbers**
(or lists, vectors, arrays, ... of numbers)
- **Examples**
 - Compute the discrete Fourier transform of a given vector x of length n
 - Matrix-matrix multiplication (MMM)
 - Compress an $n \times n$ image with a ratio ...
 - Sort a given list of integers
 - Multiply by 5, $y = 5x$, using only additions and shifts

Algorithm

- ***Algorithm:*** A precise description of a sequence of steps to solve a given problem.
- Numerical algorithms: These steps involve arithmetic computation (addition, multiplication, ...)
- Examples:
 - Cooley-Tukey fast Fourier transform
 - A description of MMM by definition
 - JPEG encoding
 - Mergesort
 - $y = x \ll 2 + x$

Tips for Presenting and Publishing

- If your topic is an algorithm, *you must*:
 - Give a formal problem specification, like:
Given; *We want to compute*.....
or
Input:; *Output*:
- Analyze the algorithm, at least asymptotic runtime in O-notation

Origin of the Word “Algorithm”

- Mathematician, astronomer and geographer; founder of Algebra (his book: Al'Jabr wa'al'Muqabilah)
- Al'Khowârizmî → *Algorithm*
Al'Jabr → *Algebra*
- Khowârizm is today the small Soviet city of Khiva
- Earlier word Algorism: The process of doing arithmetic using Arabic numerals
- Algorithm: since 1957 in Webster Dictionary



Abu Ja'far Mohammed ibn Mûsâ al'Khowârizmî (c. 825)

source:

<http://www.disc-conference.org/disc2000/mirror/khorezmi/>

image from <http://jeff560.tripod.com/>

Asymptotic Analysis of Algorithms & Problems

■ Analysis of Algorithms for

- Runtime
- Space = memory requirement (or footprint)

■ Runtime of an algorithm:

- Count “elementary” steps
(for numerical algorithms: usually floating point operations)
dependent on the input size n (more parameters may be necessary)
- State result in O-notation
- Example MMM (square and rectangular): $C = A * B + C$

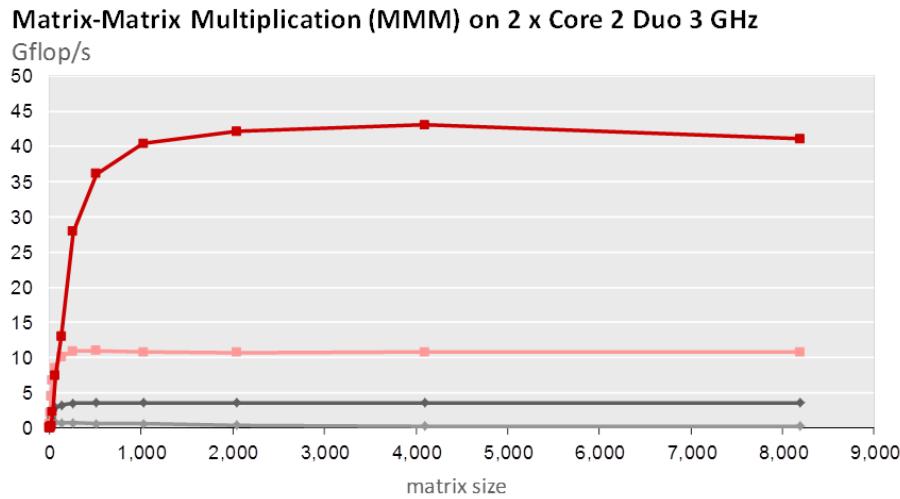
■ Runtime complexity of a problem = Minimum of the runtimes of all possible algorithms

- Result also stated in asymptotic O-notation

Complexity is a property of a problem, not of an algorithm

Valid?

- Is asymptotic analysis still valid given this?



- Yes: if the algorithm is $O(f(n))$, all memory effects are $O(f(n))$
- Vectorization, parallelization may introduce additional parameters
 - Vector length v
 - Number of processors p
 - Example: MMM

Do You Know The O?

- $O(f(n))$ is a ... ? set
- How are these related? $\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$
 - $O(f(n))$
 - $\Theta(f(n))$
 - $\Omega(f(n))$
- $O(2^n) = O(3^n)$? no
- $O(\log_2(n)) = O(\log_3(n))$ yes
- $O(n^2 + m) = O(n^2)$? no

Always Use Canonical Expressions

- Example:
 - *not* $O(2n + \log(n))$, *but* $O(n)$
- Canonical? If not replace:
 - $O(100)$ $O(1)$
 - $O(\log_2(n))$ $O(\log(n))$
 - $\Theta(n^{1.1} + n \log(n))$ $O(n^{1.1})$
 - $2n + O(\log(n))$ yes
 - $O(2n) + \log(n)$ $O(n)$
 - $\Omega(n \log(m) + m \log(n))$ yes

Master Theorem: Divide-And Conquer Algorithms

Recurrence

$$T(n) = aT(n/b) + f(n), \quad a \geq 1, b > 1$$

a subproblems of size n/b

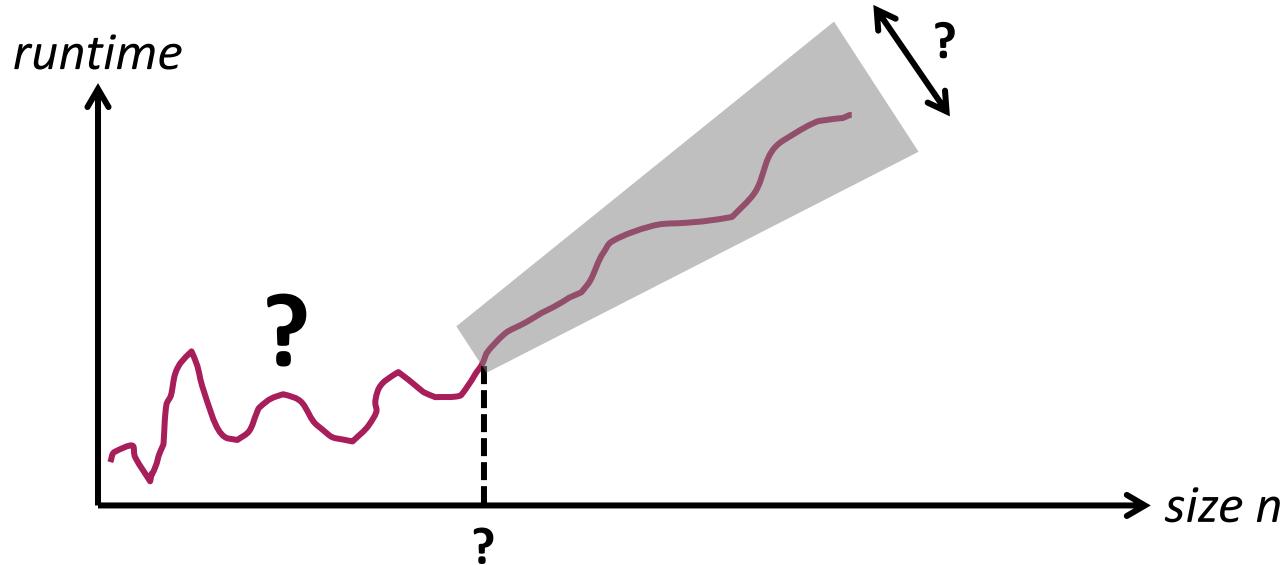
Solution

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & f(n) = O(n^{\log_b a - \epsilon}), \text{ for some } \epsilon > 0 \\ \Theta(n^{\log_b a} \log(n)), & f(n) = \Theta(n^{\log_b(a)}) \\ \Theta(f(n)), & f(n) = \Omega(n^{\log_b a + \epsilon}), \text{ for some } \epsilon > 0 \end{cases}$$

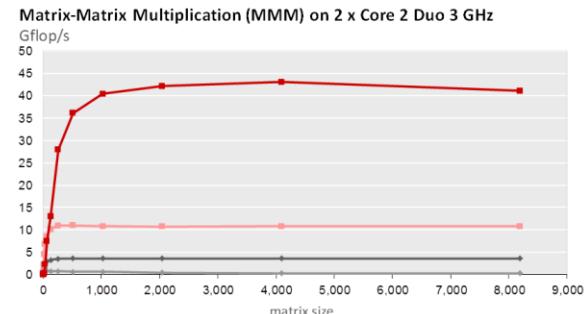
Stays valid if n/b is replaced by its floor or ceiling

Asymptotic Analysis: Limitations

- $\Theta(f(n))$ describes only the *eventual shape* of the runtime



- Constants matter
 - n^2 is likely better than $1000n^2$
 - $10000000000n$ is likely worse than n^2
- But remember: exact op count \neq runtime

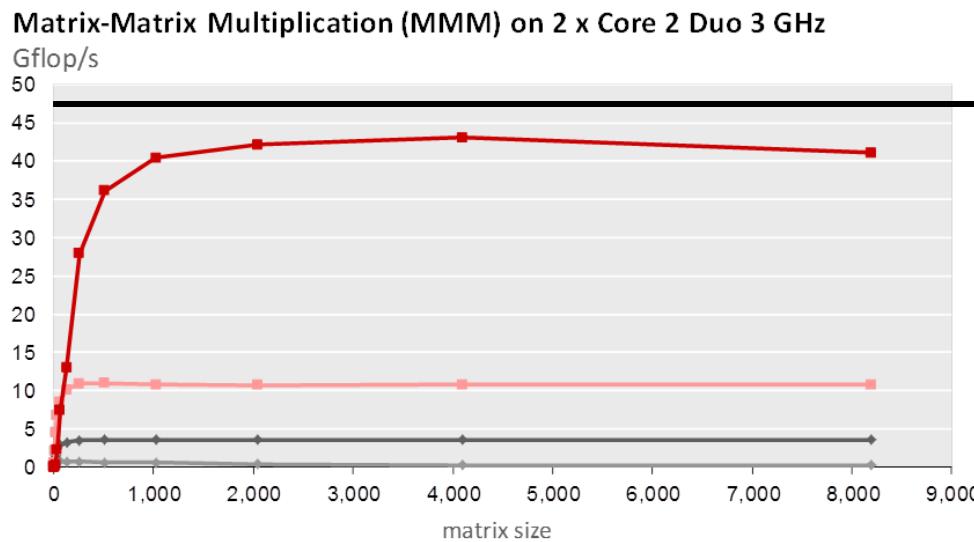


Refined Analysis for Numerical Problems

- ***Goal:*** determine exact “cost” of an algorithm
- **Approach (use MMM as running example):**
 - Fix an appropriate cost measure C : “what do I count”
 - For numerical problems typically floating point operations
 - Determine cost of algorithm as function $C(n)$ of input size n , or, more generally, of all relevant input parameters:
$$C(n_1, \dots, n_k)$$
 - Cost can be multi-dimensional
$$C(n_1, \dots, n_k) = (c_1, \dots, c_m)$$
- **Exact cost is:**
 - More precise than asymptotic runtime
 - Absolutely not the exact runtime

For Publications and Presentations

- Formally state the problem that you solve (as said before)
- State what is known about its complexity
- Analyze your algorithm (Example MMM):
 - Define your cost measure
 - Give cost as precisely as possible/meaningful
 - Enables performance analysis



*Peak performance
of this computer*

Cost Analysis

- Cost analysis of divide-and-conquer algorithms =
Solving recurrences
 - Great book: Graham, Knuth, Patashnik, “Concrete Mathematics,” 2nd edition, Addison Wesley 1994
 - Blackboard

How to Write Fast Numerical Code

Spring 2011
Lecture 3

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Organizational

- Class Monday 14.3. → Friday 18.3
- Mailing list
 - Everybody subscribed
 - Emails sent will go to everybody
 - Use to find project partners
- Email me if you know your partner

Cost Analysis

- **Exact solution of recurrences**

- Last time: first order
- Today: second (and higher) order (blackboard)

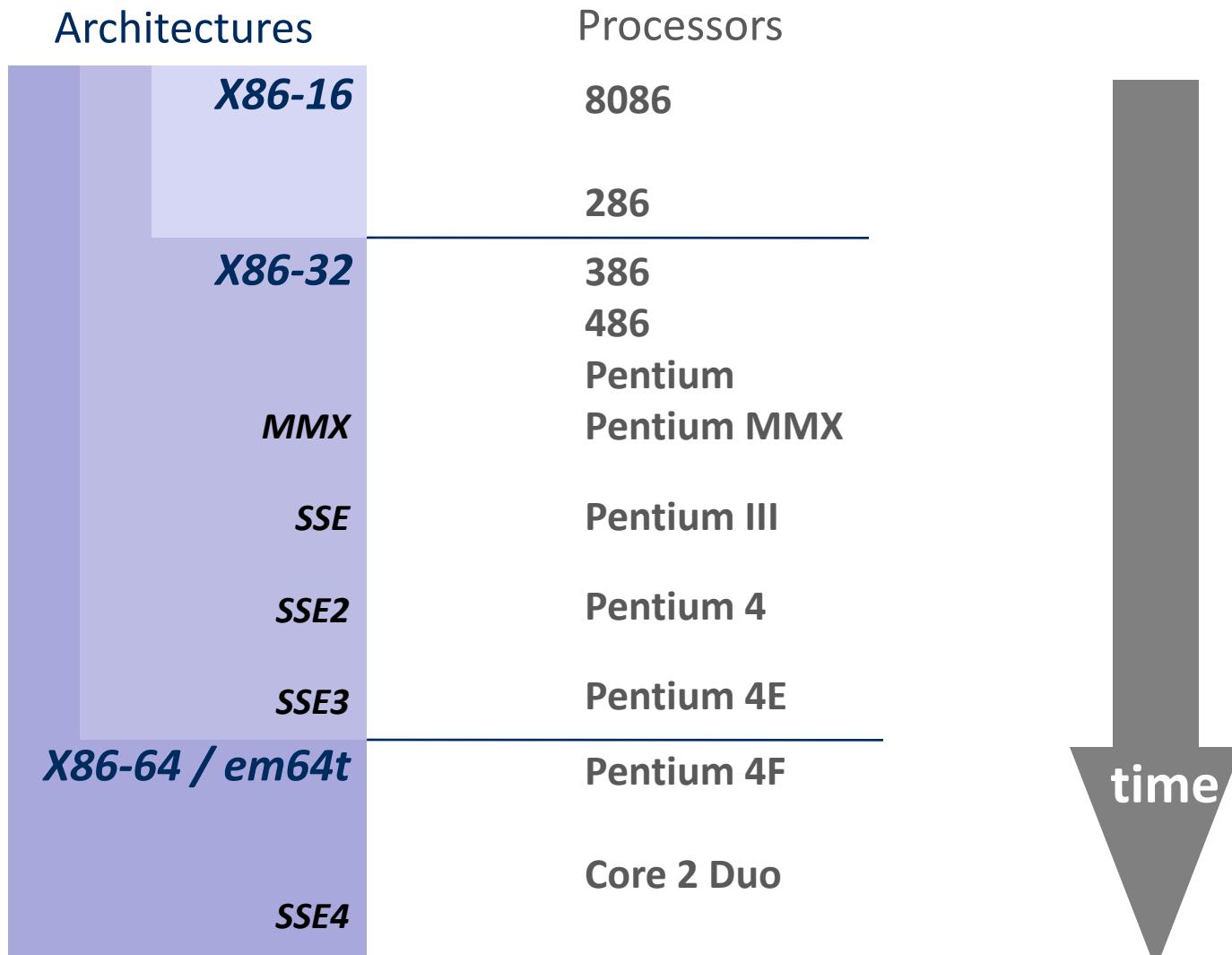
Today

- **Architecture**
- **Microarchitecture (numerical software point of view)**
- **First thoughts on fast code**

Definitions

- ***Architecture:*** (also instruction set architecture = ISA) The parts of a processor design that one needs to understand to write assembly code.
- ***Examples:*** instruction set specification, registers
- ***Counterexamples:*** cache sizes and core frequency
- **Example ISAs**
 - x86
 - ia
 - MIPS
 - POWER
 - SPARC

Intel Architectures (Focus Floating Point)

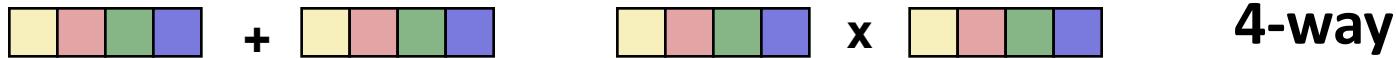


ia: often redefined as latest Intel architecture

ISA SIMD (Single Instruction Multiple Data) Vector Extensions

■ What is it?

- Extension of the ISA. Data types and instructions for the parallel computation on short (length 2-8) vectors of integers or floats



- Names: MMX, SSE, SSE2, ...

■ Why do they exist?

- **Useful:** Many applications have the necessary fine-grain parallelism
Then: speedup by a factor close to vector length
- **Doable:** Chip designers have enough transistors to play with

■ We will have an extra lecture on vector instructions

- What are the problems?
- How to use them efficiently

Definitions

- ***Microarchitecture:*** Implementation of the architecture.
- Includes caches, cache structure,
- Examples
 - Intel processors ([Wikipedia](#))
 - Intel [microarchitectures](#)

Microarchitecture: The View of the Computer Architect

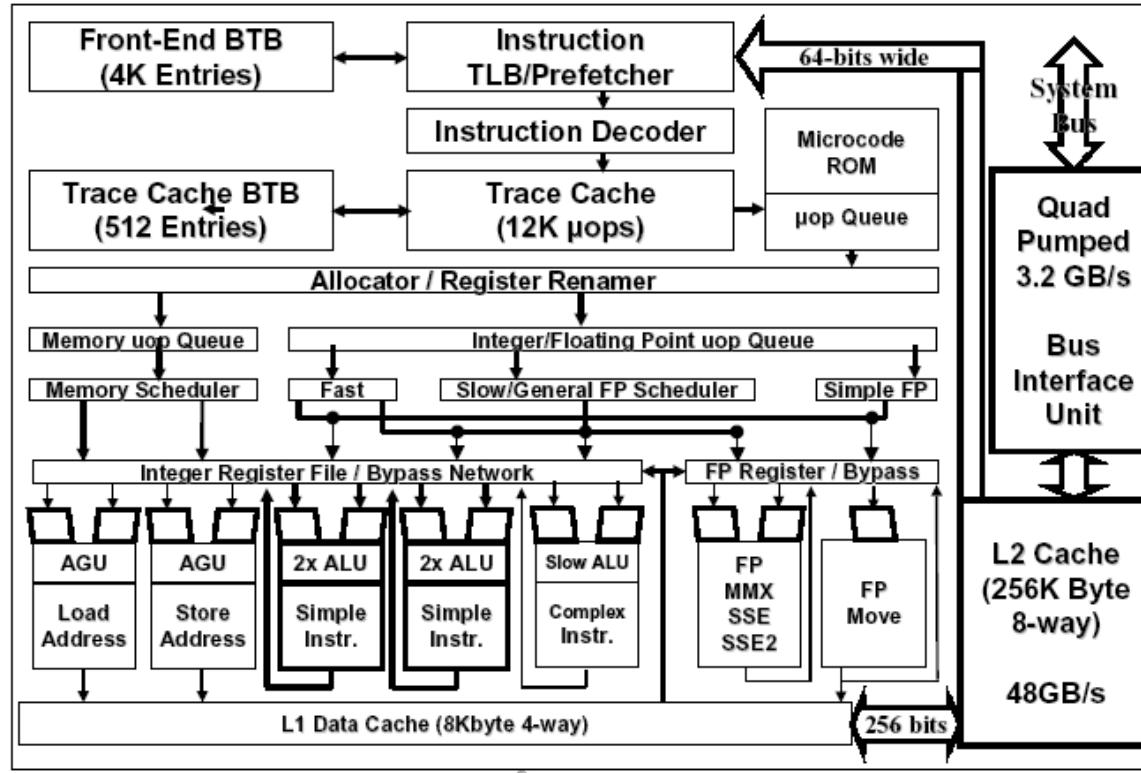
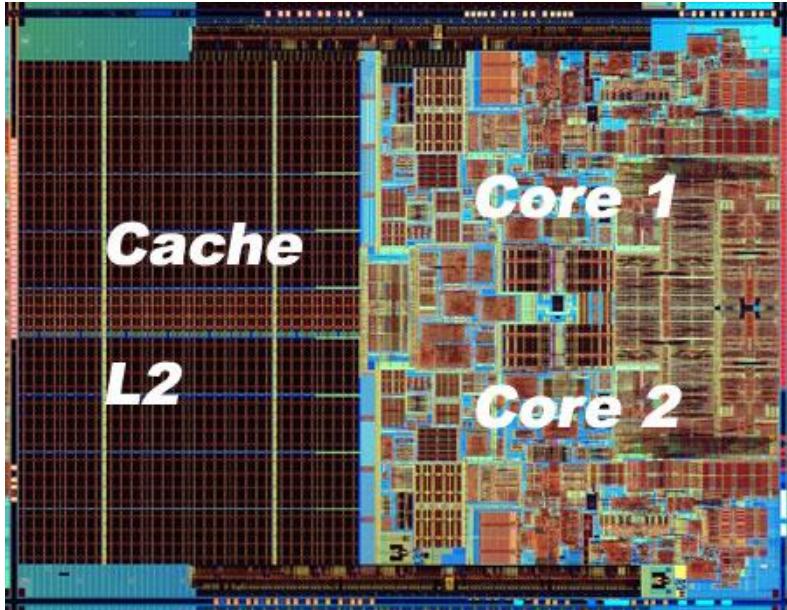


Figure 4: Pentium® 4 processor microarchitecture

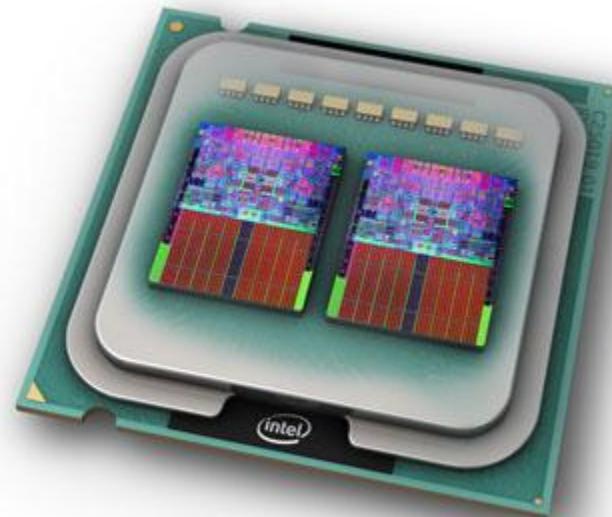
we take the software developers view ... (blackboard)

Source: "The Microarchitecture of the Pentium 4 Processor,"
Intel Technology Journal Q1 2001

Core 2 Duo



<http://www.pcmasters.de/hardware/review/intel-core-2-duo-e6700-codename-conroe-die-neue-generation.html>



**2 x Core 2 Duo
packaged**

[Detailed information about Core 2 Duo](#)

Floating Point Peak Performance

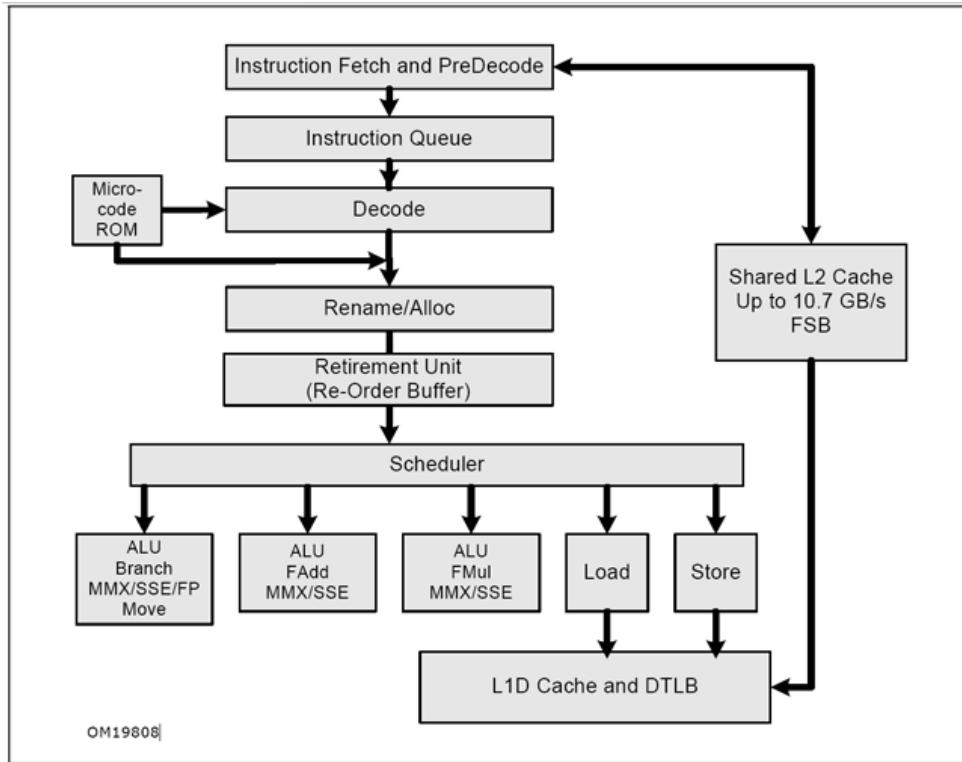


Figure 2-1. Intel Core Microarchitecture Pipeline Functionality

Theoretical peak performance (3 GHz, 1 core, no SIMD, double precision): **6 Gflop/s**

SIMD, 1 core, double precision: **12 Gflop/s**

SIMD, 1 core single precision: **24 Gflop/s**

2 or 4 cores: **multiply by 2 or 4**

Requires: computation has 50% adds and 50% mults

Latency/throughput (double)

FP Add: 3, 1

FP Mult: 5, 1

Performance: First Thought

- It is all about keeping the floating point units busy
 - Instruction level parallelism
 - Locality

How to Write Fast Numerical Code

Spring 2011
Lecture 4

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Organizational

- Class Monday 14.3. → Friday 18.3
- Office hours:
 - Markus: Tues 14–15:00
 - Georg: Wed 14–15:00
- Research projects

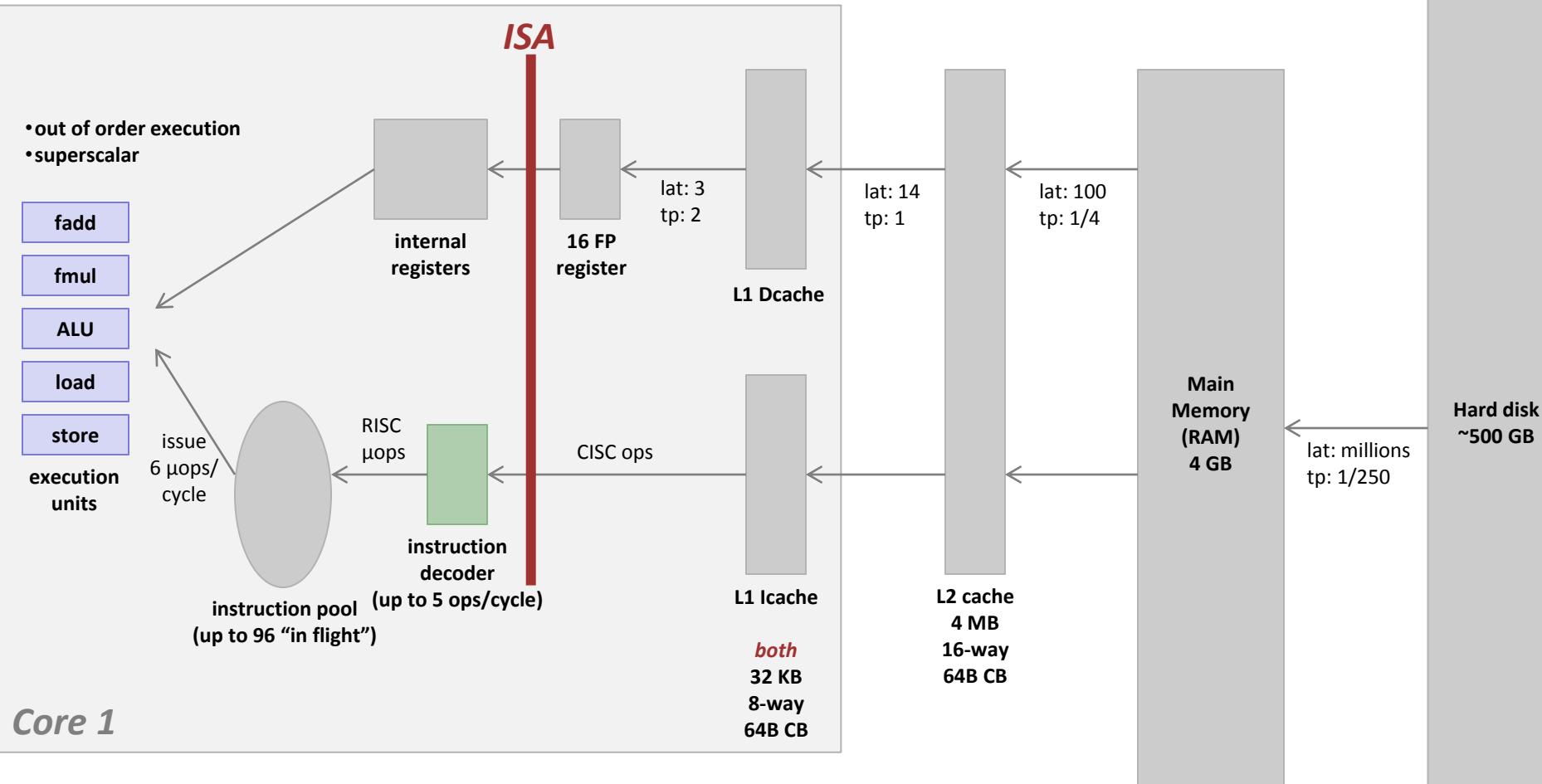
Abstracted Microarchitecture: Example Core (2008)

Throughput is measured in doubles/cycle

Latency in cycles for one double

1 double = 8 bytes

Rectangles not to scale



Memory hierarchy:

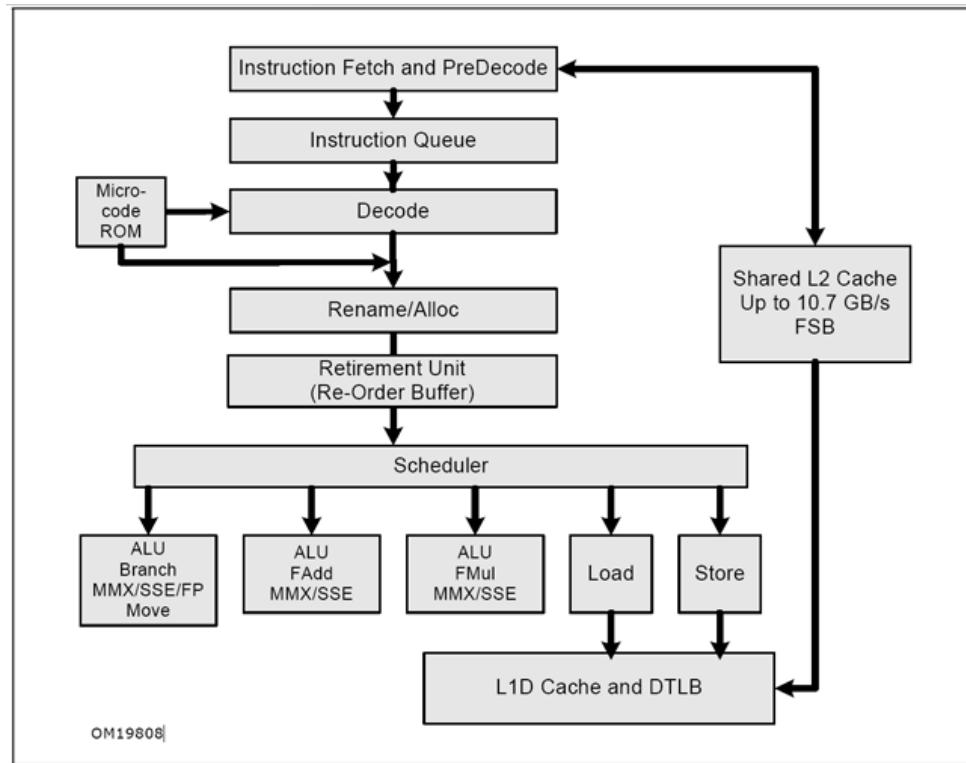
- Registers
- L1 cache
- L2 cache
- Main memory
- Hard disk

Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers

*Chapter 5 in **Computer Systems: A Programmer's Perspective**, 2nd edition,
Randal E. Bryant and David R. O'Hallaron, Addison Wesley 2010*

Core 2: Instruction Decoding and Execution Units



Latency/throughput (double)
FP Add: 3, 1
FP Mult: 5, 1

Figure 2-1. Intel Core Microarchitecture Pipeline Functionality

Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- **Benefit:** without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have
- Most CPUs since about 1998 are superscalar
- Intel: since Pentium Pro

Hard Bounds: Pentium 4 vs. Core 2

■ Pentium 4 (Nocona)

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	36/106	36/106
Single/Double FP Multiply	7	2
Single/Double FP Add	5	2
Single/Double FP Divide	32/46	32/46

■ Core 2

<i>Instruction</i>	<i>Latency</i>	<i>Cycles/Issue</i>
Load / Store	5	1
Integer Multiply	3	1
Integer/Long Divide	18/50	18/50
Single/Double FP Multiply	4/5	1
Single/Double FP Add	3	1
Single/Double FP Divide	18/32	18/32

Hard Bounds (cont'd)

- How many cycles at least if
 - Function requires n float adds?
 - Function requires n float ops (adds and mults)?
 - Function requires n int mults?

Performance in Numerical Computing

- Numerical computing =
computing dominated by floating point operations
- Example: Matrix multiplication
- Performance measure (in most cases) for a numerical function:

$$\frac{\text{\#floating point operations}}{\text{runtime [s]}}$$

- Theoretical peak performance on 3 GHz Core 2 (1 core)?
 - Scalar (no SSE): **6 Gflop/s**
 - SSE double precision: **12 Gflop/s**
 - SSE single precision: **24 Gflop/s**

Example Computation (on Pentium 4)

```
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

d[0] OP d[1] OP d[2] OP ... OP d[length-1]

■ Data Types

- Use different declarations for **data_t**
- **int**
- **float**
- **double**

■ Operations

- Use different definitions of **OP** and **IDENT**
- + / 0
- * / 1

Runtime of Combine4 (Pentium 4)

■ Use cycles/OP

```
void combine4(vec_ptr v,
    data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

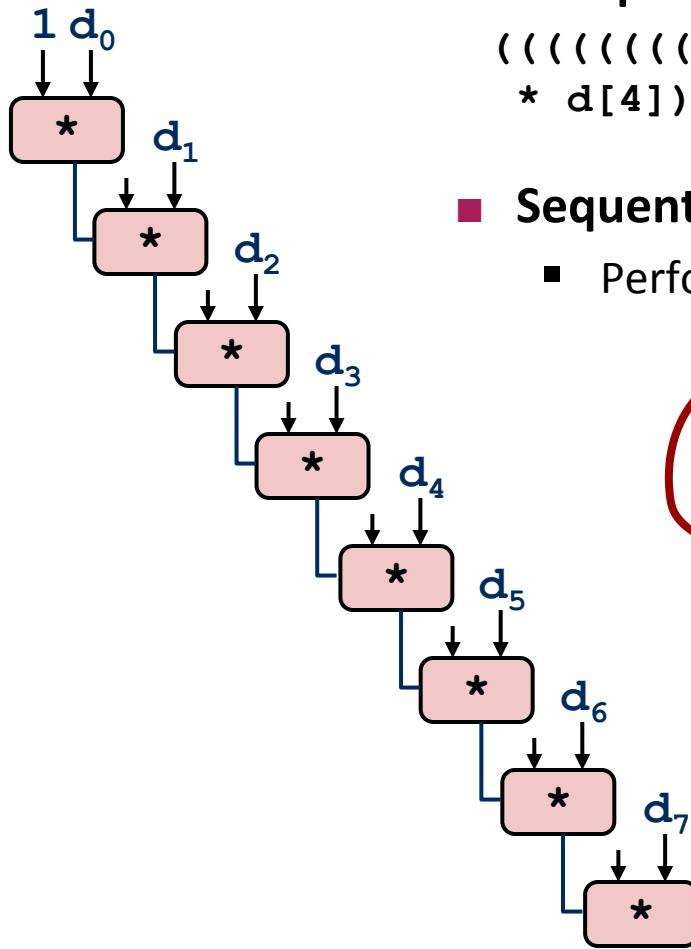
Cycles per OP

Method	Int (add/mult)	Float (add/mult)		
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

■ Questions:

- Explain red row
- Explain gray row

Combine4 = Serial Computation (OP = *)



- Computation (length=8)

```
(((((1 * d[0]) * d[1]) * d[2]) * d[3])  
 * d[4]) * d[5]) * d[6]) * d[7])
```

- Sequential dependence = no ILP! Hence,

- Performance: determined by latency of OP!

Cycles per element (or per OP)

Method	Int (add/mult)	Float (add/mult)		
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

Loop Unrolling

```
void unroll2(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

Method	Int (add/mult)	Float (add/mult)		
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

- Helps integer sum
- Others don't improve. *Why?*
 - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

Loop Unrolling with Reassociation

```
void unroll2_ra(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Can this change the result of the computation?
- Yes, for FP. *Why?*

Effect of Reassociation

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
bound	1.0	1.0	2.0	2.0

- **Nearly 2x speedup for Int *, FP +, FP ***

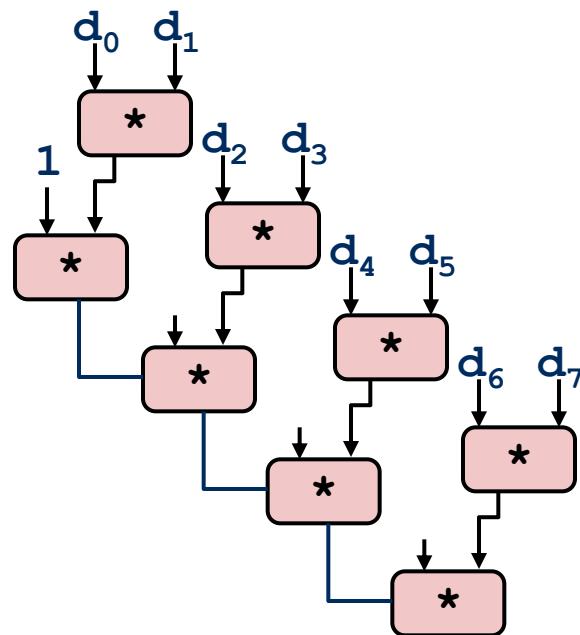
- Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

- Why is that? (next slide)

Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



■ What changed:

- Ops in the next iteration can be started early (no dependency)

■ Overall Performance

- N elements, D cycles latency/op
- Should be $(N/2+1)*D$ cycles:
cycle per OP $\approx D/2$
- Measured is slightly worse for FP

Loop Unrolling with Separate Accumulators

```
void unroll2_sa(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation

Effect of Separate Accumulators

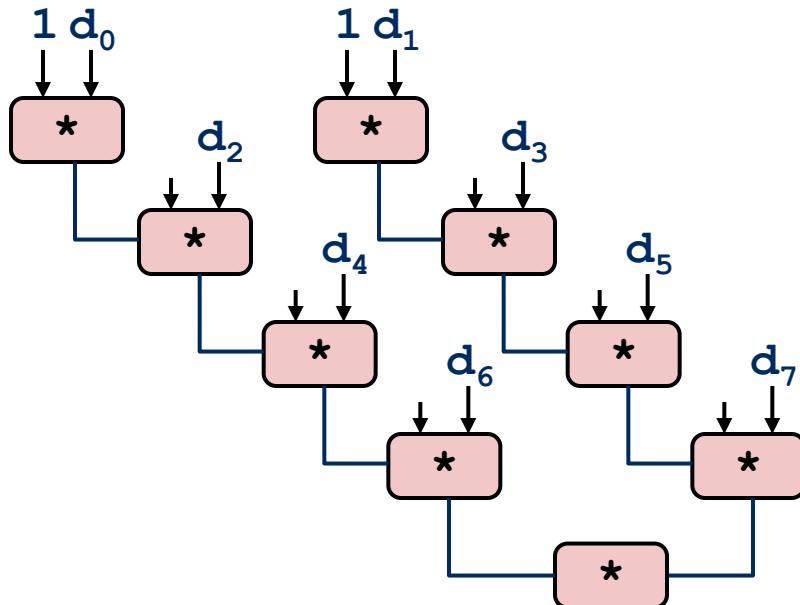
Method	Int (add/mult)	Float (add/mult)		
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-ra	1.56	5.0	2.75	3.62
unroll2-sa	1.50	5.0	2.5	3.5
bound	1.0	1.0	2.0	2.0

- Almost exact 2x speedup (over unroll2) for Int *, FP +, FP *
- Breaks sequential dependency in a “cleaner,” more obvious way

```
x0 = x0 OP d[i];  
x1 = x1 OP d[i+1];
```

Separate Accumulators

```
x0 = x0 OP d[i];  
x1 = x1 OP d[i+1];
```



■ What changed:

- Two independent “streams” of operations

■ Overall Performance

- N elements, D cycles latency/op
- Should be $(N/2+1)*D$ cycles:
cycles per OP $\approx D/2$

What Now?

Unrolling & Accumulating

■ Idea

- Use K accumulators
- Increase K until best performance reached
- Need to unroll by L, K divides L

■ Limitations

- Diminishing returns:
Cannot go beyond throughput limitations of execution units
- Large overhead for short lengths: Finish off iterations sequentially

Unrolling & Accumulating: Intel FP *

■ Case

- Pentium 4
- FP Multiplication
- Theoretical Limit: 2.00

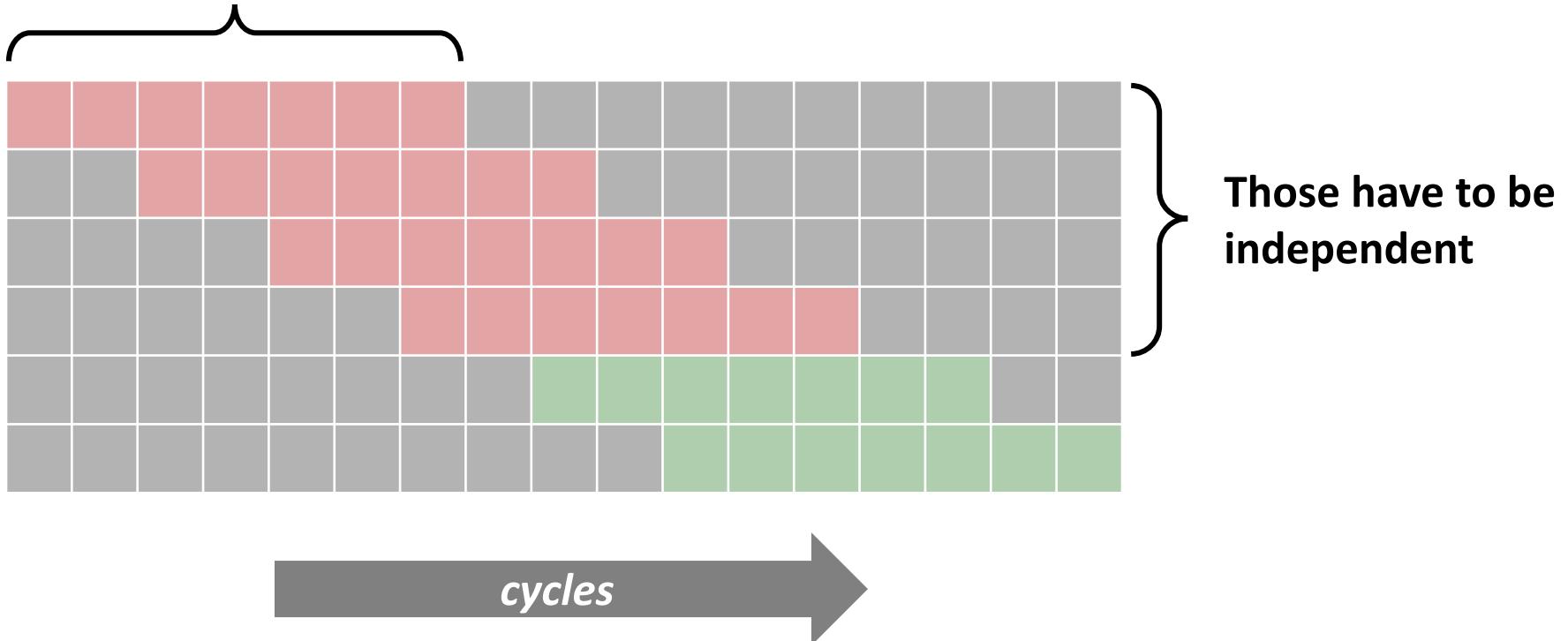
Accumulators

FP *	Unrolling Factor L								
K	1	2	3	4	6	8	10	12	
1	7.00	7.00		7.01		7.00			
2		3.50		3.50		3.50			
3			2.34						
4				2.01		2.00			
6					2.00			2.01	
8						2.01			
10							2.00		
12								2.00	

Why 4?

Why 4?

Latency: 7 cycles



Based on this insight: $K = \#\text{accumulators} = \text{ceil}(\text{latency}/\text{cycles per issue})$

Unrolling & Accumulating: Intel FP +

■ Case

- Pentium 4
 - FP Addition
 - Theoretical Limit: 2.00

Unrolling & Accumulating: Intel Int *

■ Case

- Pentium 4
 - Integer Multiplication
 - Theoretical Limit: 1.00

Unrolling & Accumulating: Intel Int +

Case

- Pentium 4
 - Integer addition
 - Theoretical Limit: 1.00

FP *	Unrolling Factor L								
	K	1	2	3	4	6	8	10	12
1	7.00	7.00			7.01		7.00		
2			3.50		3.50		3.50		
3				2.34					
4					2.01		2.00		
6						2.00			2.01
8							2.01		
10								2.00	
12									2.00

Pentium 4

FP *	Unrolling Factor L								
	K	1	2	3	4	6	8	10	12
1	4.00	4.00			4.00		4.01		
2			2.00		2.00		2.00		
3				1.34					
4					1.00		1.00		
6						1.00			1.00
8							1.00		
10								1.00	
12									1.00

Core 2

*FP * is fully pipelined*

Summary (ILP)

- Instruction level parallelism may have to be made explicit in program
- Potential blockers for compilers
 - Reassociation changes result (FP)
 - Too many choices, no good way of deciding
- Unrolling
 - By itself does often nothing (branch prediction works usually well)
 - But may be needed to enable additional transformations (here: reassociation)
- How to program this example?
 - Solution 1: program generator generates alternatives and picks best
 - Solution 2: use model based on latency and throughput

Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
 - Overview
 - Removing unnecessary procedure calls
 - Code motion
 - Strength reduction
 - Sharing of common subexpressions
 - Optimization blocker: Procedure calls
 - Optimization blocker: Memory aliasing
 - Summary

*Compiler is likely
to do that*

Optimizing Compilers



- Use optimization flags, *default is no optimization* (-O0)!
- Good choices for gcc: -O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers

Example

```
double a[4][4];
double b[4][4];
double c[4][4]; # set to zero

/* Multiply 4 x 4 matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i*4+j] += a[i*4 + k]*b[k*4 + j];
}
```

- Compiled without flags:
~1300 cycles
- Compiled with -O3 -m64 -march=... -fno-tree-vectorize
~150 cycles
- Core 2 Duo

Prevents use of SSE



Optimizing Compilers

- Compilers are *good* at: mapping program to machine
 - register allocation
 - code selection and ordering (instruction scheduling)
 - dead code elimination
 - eliminating minor inefficiencies
- Compilers are *not good* at: algorithmic restructuring
 - For example to increase ILP, locality, etc.
 - Cannot deal with choices
- Compilers are *not good* at: overcoming “optimization blockers”
 - potential memory aliasing
 - potential procedure side-effects

Limitations of Optimizing Compilers

- *If in doubt, the compiler is conservative*
- Operate under fundamental constraints
 - Must not change program behavior under any possible condition
 - Often prevents it from making optimizations when would only affect behavior under pathological conditions
- Most analysis is performed only within procedures
 - Whole-program analysis is too expensive in most cases
- Most analysis is based only on *static* information
 - Compiler has difficulty anticipating run-time inputs
 - Not good at evaluating or dealing with choices

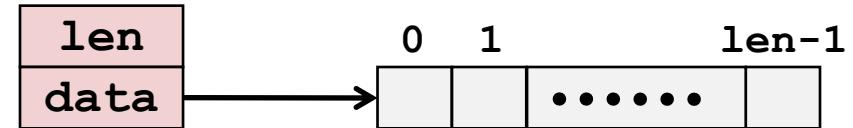
Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
 - Overview
 - *Removing unnecessary procedure calls*
 - Code motion
 - Strength reduction
 - Sharing of common subexpressions
 - Optimization blocker: Procedure calls
 - Optimization blocker: Memory aliasing
 - Summary

*Compiler is likely
to do that*

Example: Data Type for Vectors

```
/* data structure for vectors */  
typedef struct{  
    int len;  
    double *data;  
} vec;
```



```
/* retrieve vector element and store at val */  
int get_vec_element(*vec, idx, double *val)  
{  
    if (idx < 0 || idx >= v->len)  
        return 0;  
    *val = v->data[idx];  
    return 1;  
}
```

Example: Summing Vector Elements

```
/* retrieve vector element and store at val */
int get_vec_element(*vec, idx, double *val)
{
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double val;

    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &val);
        *res += val;
    }
    return res;
}
```

Overhead for every fp +:

- One fct call
- One <
- One \geq
- One $\|$
- One memory variable access

Slowdown:

probably 10x or more

Removing Procedure Call

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double val;

    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &val);
        *res += val;
    }
    return res;
}
```

```
/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double *data = get_vec_start(v);

    for (i = 0; i < n; i++)
        *res += data[i];
    return res;
}
```

Removing Procedure Calls

- Procedure calls can be very expensive
- Bound checking can be very expensive
- Abstract data types can easily lead to inefficiencies
 - Usually avoided for in superfast numerical library functions
- *Watch your innermost loop!*
- *Get a feel for overhead versus actual computation being performed*

Organization

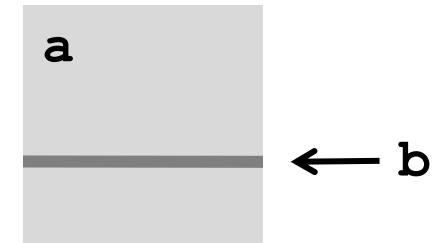
- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
 - Overview
 - Removing unnecessary procedure calls
 - **Code motion**
 - Strength reduction
 - Sharing of common subexpressions
 - Optimization blocker: Procedure calls
 - Optimization blocker: Memory aliasing
 - Summary

*Compiler is likely
to do that*

Code Motion

- Reduce frequency with which computation is performed
 - If it will always produce same result
 - Especially moving code out of loop (loop-invariant code motion)
- Sometimes also called precomputation

```
void set_row(double *a, double *b,
             long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```



```
long j;
int ni = n*i;
for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```

Organization

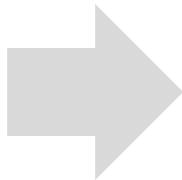
- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
 - Overview
 - Removing unnecessary procedure calls
 - Code motion
 - ***Strength reduction***
 - Sharing of common subexpressions
 - Optimization blocker: Procedure calls
 - Optimization blocker: Memory aliasing
 - Summary

*Compiler is likely
to do that*

Strength Reduction

- Replace costly operation with simpler one
- Example: Shift/add instead of multiply or divide
 - $16 \times x \rightarrow x \ll 4$
- Example: Recognize sequence of products

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



```
int ni = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
        a[ni + j] = b[j];
    ni += n;
}
```

Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
 - Overview
 - Removing unnecessary procedure calls
 - Code motion
 - Strength reduction
 - *Sharing of common subexpressions*
 - Optimization blocker: Procedure calls
 - Optimization blocker: Memory aliasing
 - Summary

*Compiler is likely
to do that*

Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

*3 mults: $i*n$, $(i-1)*n$, $(i+1)*n$*

```
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n      + j-1];
right = val[i*n      + j+1];
sum = up + down + left + right;
```

*1 mult: $i*n$*

```
int inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
 - Overview
 - Removing unnecessary procedure calls
 - Code motion
 - Strength reduction
 - Sharing of common subexpressions
- ***Optimization blocker: Procedure calls***
- Optimization blocker: Memory aliasing
- Summary

*Compiler is likely
to do that*

How to Write Fast Numerical Code

Spring 2011
Lecture 5

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Organizational

- *Class Monday 14.3. → Friday 18.3*

- **Office hours:**

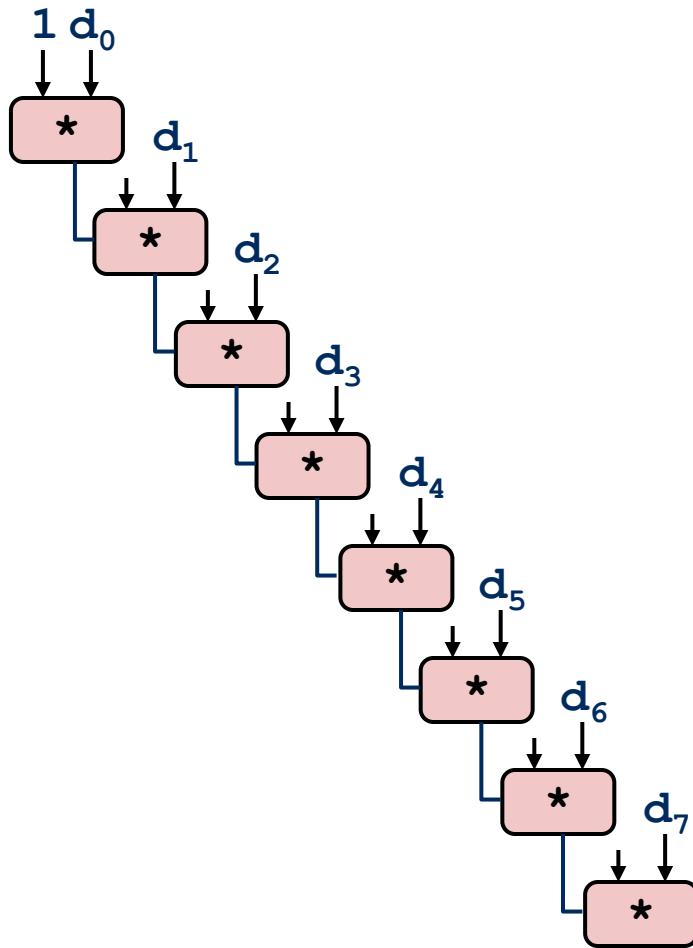
- Markus: Tues 14–15:00
 - Georg: Wed 14–15:00

- **Research projects**

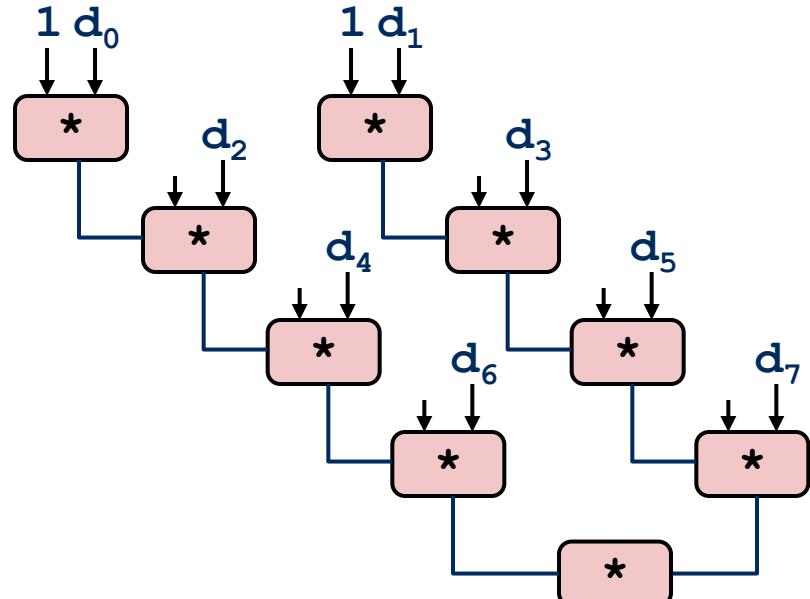
- 11 groups, 23 people
 - I need to approve the projects

Last Time: ILP

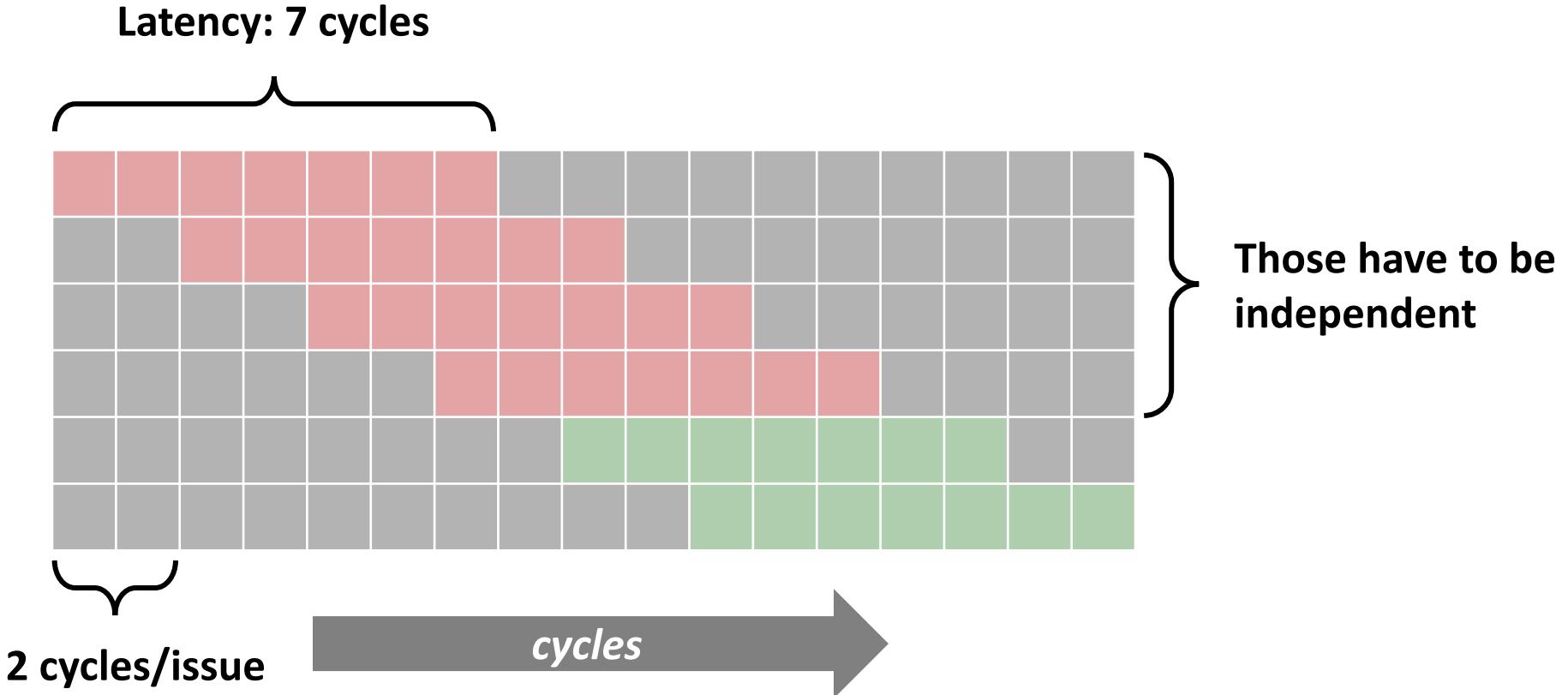
- Latency/throughput (Pentium 4 fp mult: 7/2)



Twice as fast



Last Time: Why ILP?



Based on this insight: $K = \#\text{accumulators} = \text{ceil}(\text{latency}/\text{cycles per issue})$

Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
 - Overview
 - Removing unnecessary procedure calls
 - Code motion
 - Strength reduction
 - Sharing of common subexpressions
- ***Optimization blocker: Procedure calls***
- Optimization blocker: Memory aliasing
- Summary

*Compiler is likely
to do that*

Optimization Blocker #1: Procedure Calls

- Procedure to convert string to lower case

```
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

$O(n^2)$ instead of $O(n)$

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

$O(n)$

Improving Performance

```
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

```
void lower(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion/precomputation

Optimization Blocker: Procedure Calls

- Why couldn't compiler move `strlen` out of inner loop?
 - Procedure may have side effects
- *Compiler usually treats procedure call as a black box that cannot be analyzed*
 - Consequence: conservative in optimizations
- In this case the compiler may actually do if `strlen` is recognized as built-in function

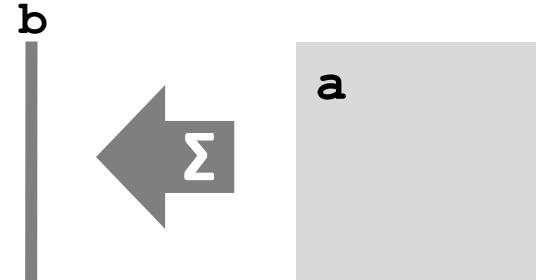
Organization

- Instruction level parallelism (ILP): an example
- Optimizing compilers and optimization blockers
 - Overview
 - Removing unnecessary procedure calls
 - Code motion
 - Strength reduction
 - Sharing of common subexpressions
 - Optimization blocker: Procedure calls
 - ***Optimization blocker: Memory aliasing***
 - Summary

*Compiler is likely
to do that*

Optimization Blocker: Memory Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```



- Code updates **b [i]** (= memory access) on every iteration
- Does compiler optimize this away? **No!**

Reason: Possible Memory Aliasing

- If memory is accessed, compiler assumes the possibility of side effects
- Example:

```
/* Sums rows of n x n matrix a
   and stores in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
{ 0,    1,    2,
  4,    8,   16},
32,   64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of B:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

Removing Aliasing

```
/* Sums rows of n x n matrix a
   and stores in vector b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

■ Scalar replacement:

- Copy array elements *that are reused* into temporary variables
- Perform computation on those variables
- Enables register allocation and instruction scheduling
- Assumes no memory aliasing (otherwise possibly incorrect)

Optimization Blocker: Memory Aliasing

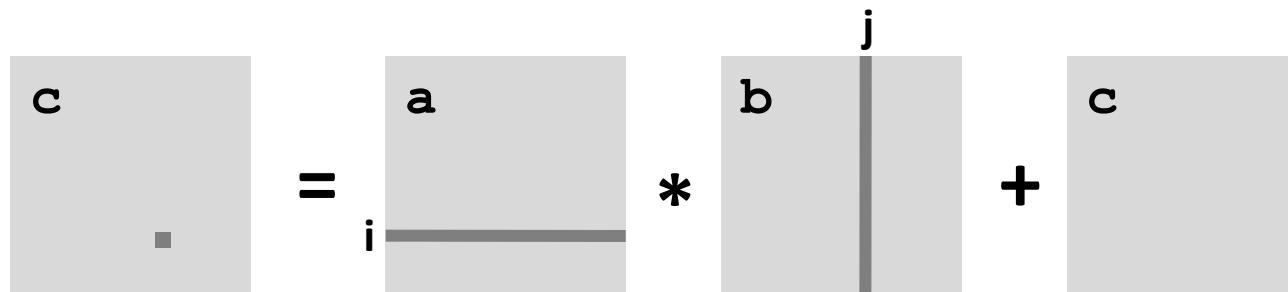
- **Memory aliasing:**
Two different memory references write to the same location
- **Easy to have happen in C**
 - Since allowed to do address arithmetic
 - Direct access to storage structures
- **Hard to analyze = compiler cannot figure it out**
 - Hence is conservative
- **Solution: Scalar replacement in innermost loop**
 - Copy memory variables ***that are reused*** into local variables
 - Basic scheme:
 - ***Load:*** $t1 = a[i], t2 = b[i+1], \dots$
 - ***Compute:*** $t4 = t1 * t2; \dots$
 - ***Store:*** $a[i] = t12, b[i+1] = t7, \dots$

More Difficult Example

- Matrix multiplication: $C = A * B + C$

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```



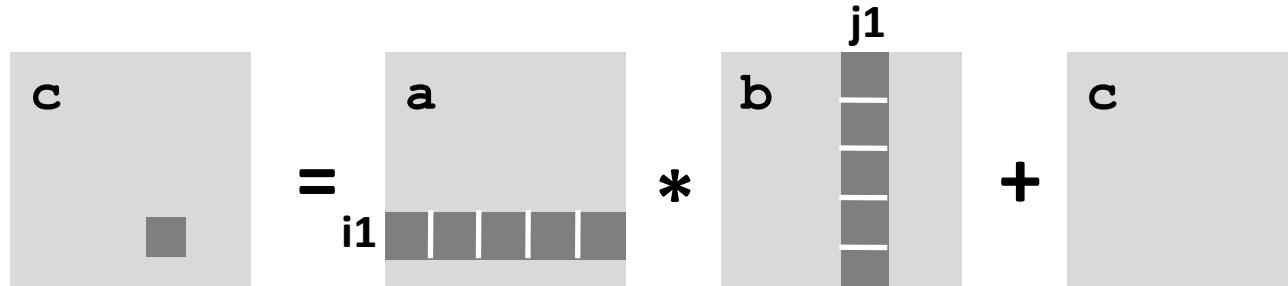
- Which array elements are reused?
- All of them! *But how to take advantage?*

Step 1: Blocking (Here: 2 x 2)

- Blocking, also called tiling = partial unrolling + loop exchange
 - Assumes associativity (= compiler will not do it)

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=2)
        for (j = 0; j < n; j+=2)
            for (k = 0; k < n; k+=2)
                for (i1 = i; i1 < i+2; i1++)
                    for (j1 = j; j1 < j+2; j1++)
                        for (k1 = k; k1 < k+2; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



Step 2: Unrolling Inner Loops

```
c = (double *) calloc(sizeof(double), n*n);  
  
/* Multiply n x n matrices a and b */  
void mmm(double *a, double *b, double *c, int n) {  
    int i, j, k;  
    for (i = 0; i < n; i+=2)  
        for (j = 0; j < n; j+=2)  
            for (k = 0; k < n; k+=2)  
                <body>  
}
```

<body>

$$\begin{aligned} c[i*n + j] &= a[i*n + k]*b[k*n + j] + a[i*n + k+1]*b[(k+1)*n + j] \\ &\quad + c[i*n + j] \\ c[(i+1)*n + j] &= a[(i+1)*n + k]*b[k*n + j] + a[(i+1)*n + k+1]*b[(k+1)*n + j] \\ &\quad + c[(i+1)*n + j] \\ c[i*n + (j+1)] &= a[i*n + k]*b[k*n + (j+1)] + a[i*n + k+1]*b[(k+1)*n + (j+1)] \\ &\quad + c[i*n + (j+1)] \\ c[(i+1)*n + (j+1)] &= a[(i+1)*n + k]*b[k*n + (j+1)] \\ &\quad + a[(i+1)*n + k+1]*b[(k+1)*n + (j+1)] + c[(i+1)*n + (j+1)] \end{aligned}$$

- Every array element $a[\dots]$, $b[\dots]$, $c[\dots]$ used twice
- Now scalar replacement can be applied
(so again: loop unrolling is done with a purpose)

Organization

- Instruction level parallelism (ILP): an example
- **Optimizing compilers and optimization blockers**
 - Overview
 - Removing unnecessary procedure calls
 - Code motion
 - Strength reduction
 - Sharing of common subexpressions
 - Optimization blocker: Procedure calls
 - Optimization blocker: Memory aliasing
 - *Summary*

*Compiler is likely
to do that*

Summary

- *One can easily loose 10x, 100x in runtime or even more*



- What matters besides operation count:
 - Coding style (unnecessary procedure calls, unrolling, reordering, ...)
 - Algorithm structure (instruction level parallelism, locality, ...)
 - Data representation (complicated structs or simple arrays)

Summary: Optimize at Multiple Levels

■ Algorithm:

- Evaluate different algorithm choices
- Restructuring may be needed (ILP, locality)

■ Data representations:

- Careful with overhead of complicated data types
- Best are arrays

■ Procedures:

- Careful with overhead
- They are black boxes for the compiler

■ Loops:

- Often need to be restructured (ILP, locality)
- Unrolling often necessary to enable other optimizations
- Watch the innermost loop bodies

Numerical Functions

- **Use arrays if possible**
- **Unroll to some extent**
 - To make ILP explicit
 - To enable scalar replacement and hence register allocation for variables that are reused

Organization

- **Benchmarking: Basics**

Section 3.2 in the tutorial <http://spiral.ece.cmu.edu:8080/publications/abstract.jsp?id=100>

Benchmarking

- ***First:*** Verify your code!
- Measure runtime in seconds for a set of relevant input sizes
- Determine performance [flop/s]
 - Assumes negligible number of other ops (division, sin, cos, ...)
 - Needs arithmetic cost:
 - *Obtained statically (cost analysis since you understand the algorithm)*
 - *or dynamically (tool that counts, or replace ops by counters through macros)*
 - Compare to theoretical peak performance
- ***Careful:*** Different algorithms may have different op count, i.e., best flop/s is not always best runtime

How to measure runtime?

- **C clock()**
 - process specific, low resolution, very portable
- **gettimeofday**
 - measures wall clock time, higher resolution, somewhat portable
- **Performance counter (e.g., TSC on Pentiums)**
 - measures cycles (i.e., also wall clock time), highest resolution, not portable
- **Careful:**
 - measure only what you want to measure
 - ensure proper machine state
(e.g., cold or warm cache = input data is or is not in cache)
 - measure enough repetitions
 - check how reproducible; if not reproducible: fix it
- **Getting proper measurements is not easy at all!**

Example: Timing MMM

- Assume **MMM(A,B,C,n)** computes

$$C = C + AB, \quad A, B, C \text{ are } nxn \text{ matrices}$$

```
double time_MMM(int n)
{ // allocate
    double *A=(double*)malloc(n*n*sizeof(double));
    double *B=(double*)malloc(n*n*sizeof(double));
    double *C=(double*)malloc(n*n*sizeof(double));

    // initialize
    for(int i=0; i<n*n; i++){
        A[i] = B[i] = C[i] = 0.0;
    }

    init_MMM(A,B,C,n); // if needed

    // warm up cache (for warm cache timing)
    MMM(A,B,C,n);

    // time
    ReadTime(t0);
    for(int i=0; i<TIMING_REPEATITIONS; i++)
        MMM(A,B,C,n);
    ReadTime(t1);

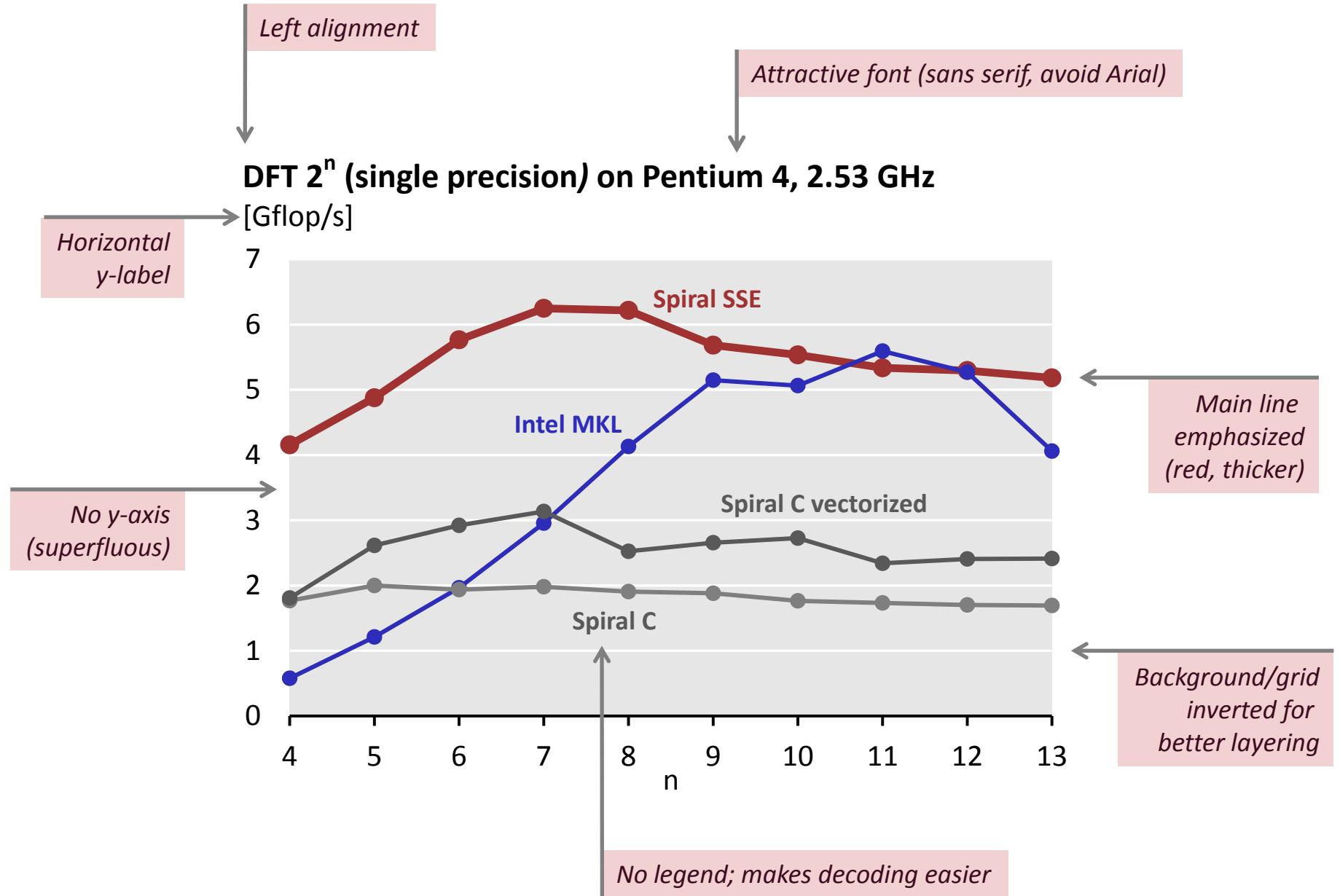
    // compute runtime
    return (double)((t1-t0)/TIMING_REPEATITIONS);
}
```

Problems with Timing

- Too few iterations: inaccurate non-reproducible timing
- Too many iterations: system events interfere
- Machine is under load: produces side effects
- Multiple timings performed on the same machine
- Bad data alignment of input/output vectors: align to multiples of cache line
(on Core: address is divisible by 64)
- Time stamp counter (if used) overflows
- Machine was not rebooted for a long time: state of operating system causes problems
- Computation is input data dependent: choose representative input data
- Computation is inplace and data grows until an exception is triggered
(computation is done with NaNs)
- You work on a laptop that has dynamic frequency scaling
- Always check whether timings make sense, are reproducible

Benchmarks in Writing

- Specify platform, compiler and version, compiler flags used
- Plot: Very readable
 - Title, x-label, y-label should be there
 - Fonts large enough
 - Enough contrast (no yellow on white please)
 - Proper number format
 - **No:** 13.254687; **yes:** 13.25
 - **No:** 2.0345e-05 s; **yes:** 20.3 μ s
 - **No:** 100000 B; **maybe:** 100,000 B; **yes:** 100 KB



How to Write Fast Numerical Code

Spring 2011
Lecture 6

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Organizational

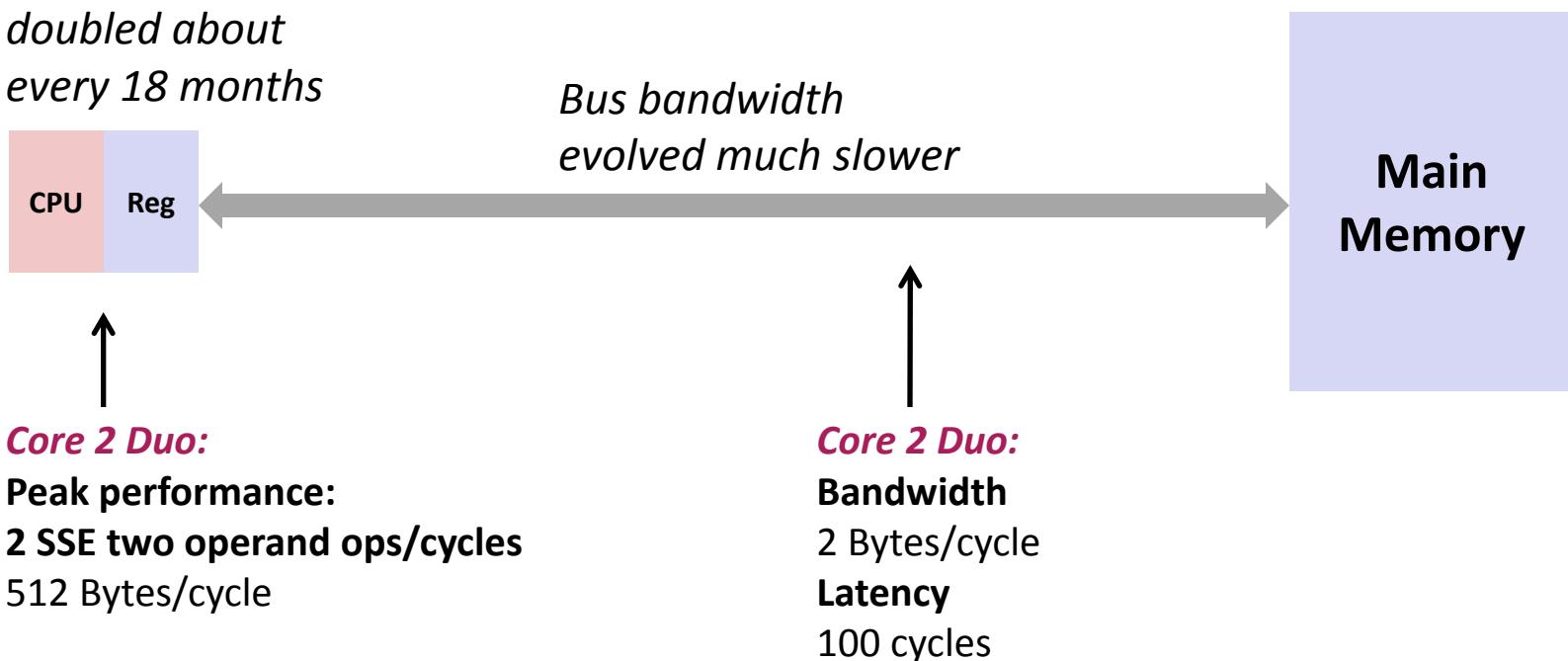
- *Class this Friday 18.3*
- Exam?
 - Monday April 11 (Sechseläuten, afternoon is off)
 - Friday April 15

Organization

- Temporal and spatial locality
- Caches

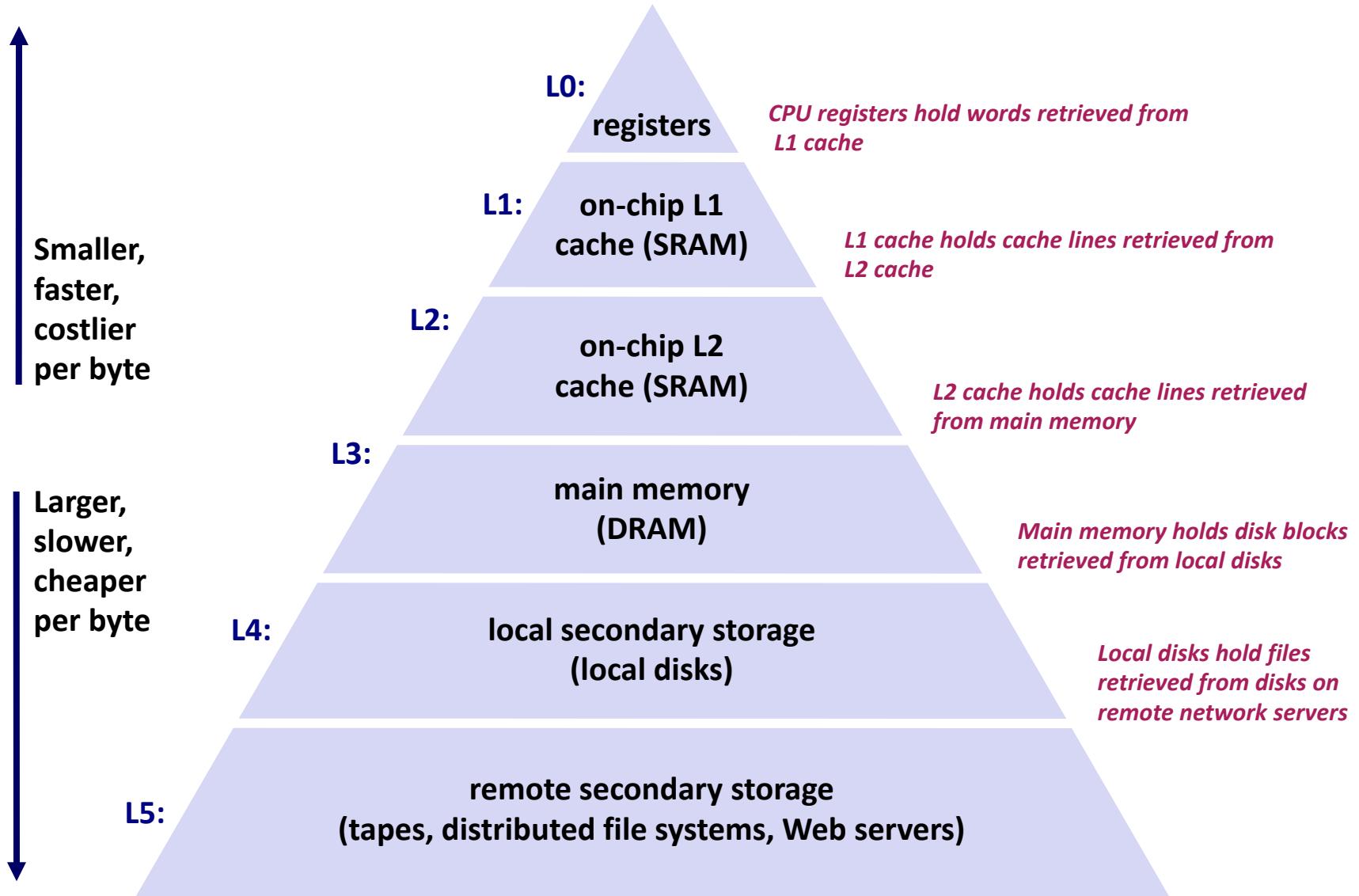
Problem: Processor-Memory Bottleneck

*Processor performance
doubled about
every 18 months*



Solution: Caches/Memory hierarchy

Typical Memory Hierarchy



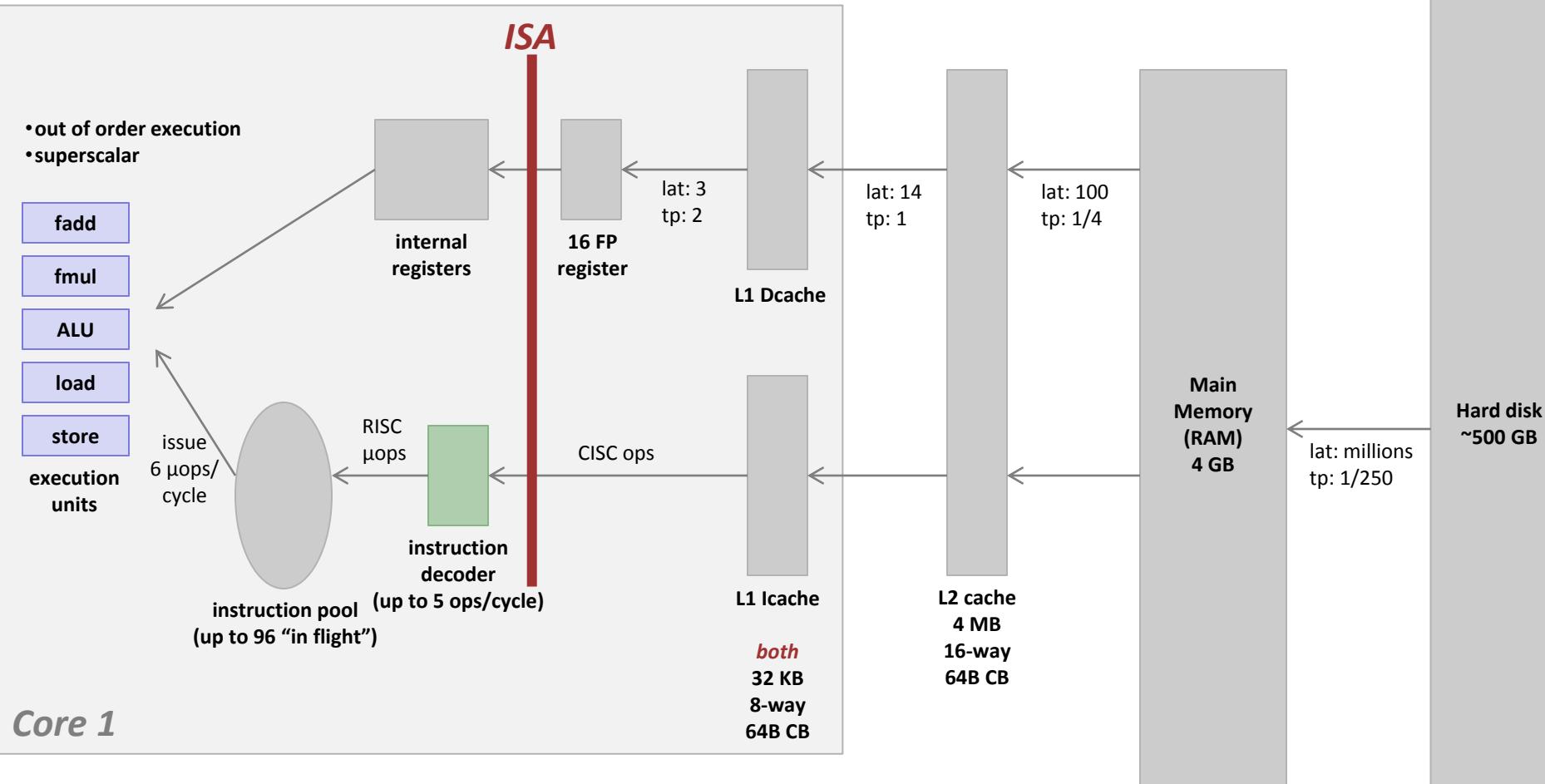
Abstracted Microarchitecture: Example Core (2008)

Throughput is measured in doubles/cycle

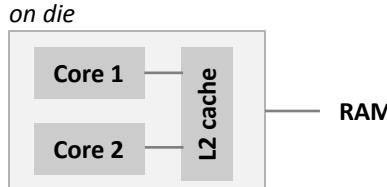
Latency in cycles for one double

1 double = 8 bytes

Rectangles not to scale



Core 2 Duo:



Memory hierarchy:

- Registers
- L1 cache
- L2 cache
- Main memory
- Hard disk

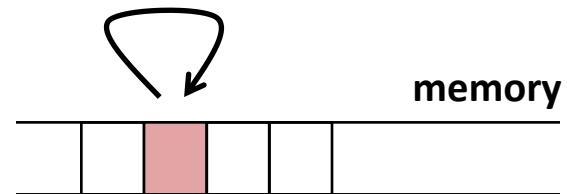
Why Caches Work: Locality

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

History of locality

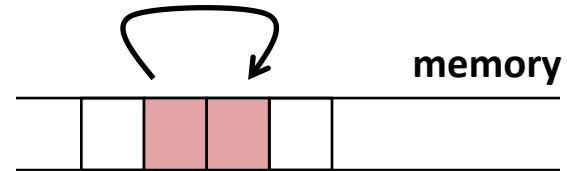
- **Temporal locality:**

- Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**

- Items with nearby addresses tend to be referenced close together in time



Example: Locality?

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

- **Data:**
 - Temporal: **sum** referenced in each iteration
 - Spatial: array **a []** accessed in stride-1 pattern
- **Instructions:**
 - Temporal: loops cycle through the same instructions
 - Spatial: instructions referenced in sequence
- *Being able to assess the locality of code is a crucial skill for a performance programmer*

Locality Example #1

```
int sum_array_rows(int a[M] [N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example #2

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Locality Example #3

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

- How to improve locality?

Reuse (Inherent Temporal Locality)

- *Reuse of an algorithm:*

$$\frac{\text{Number of operations}}{\text{Size of input} + \text{size of output data}}$$

- **Typically:**
no. operations = arithmetic cost = no. floating point adds and mults
- **Intuitively measures how often every input element is on average needed in the computation**
- **Examples:**

- Matrix multiplication $C = AB + C$ $\frac{2n^3}{3n^2} = \frac{2}{3}n = O(n)$

- Discrete Fourier transform $\approx \frac{5n \log_2(n)}{2n} = \frac{5}{2} \log_2(n) = O(\log(n))$

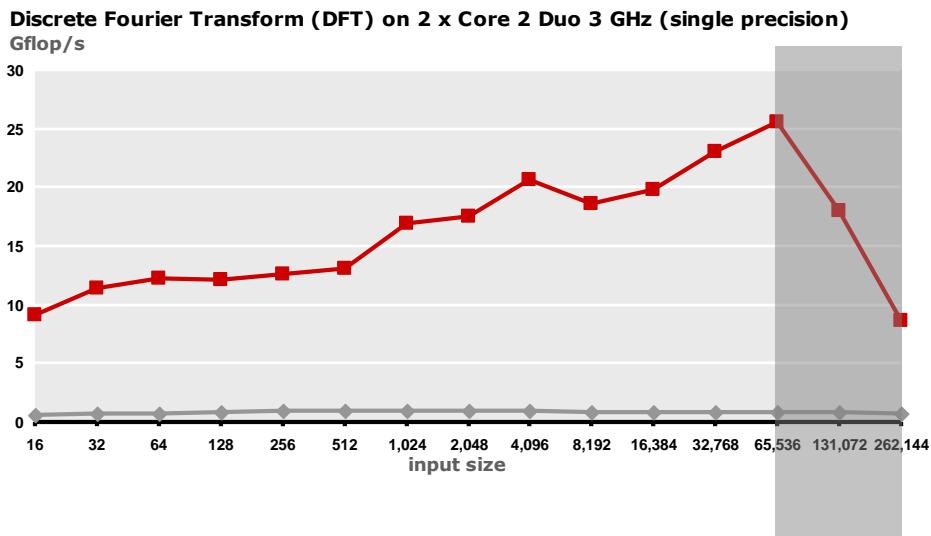
- Adding two vectors $x = x+y$ $\frac{n}{2n} = \frac{1}{2} = O(1)$

CPU bound versus Memory bound

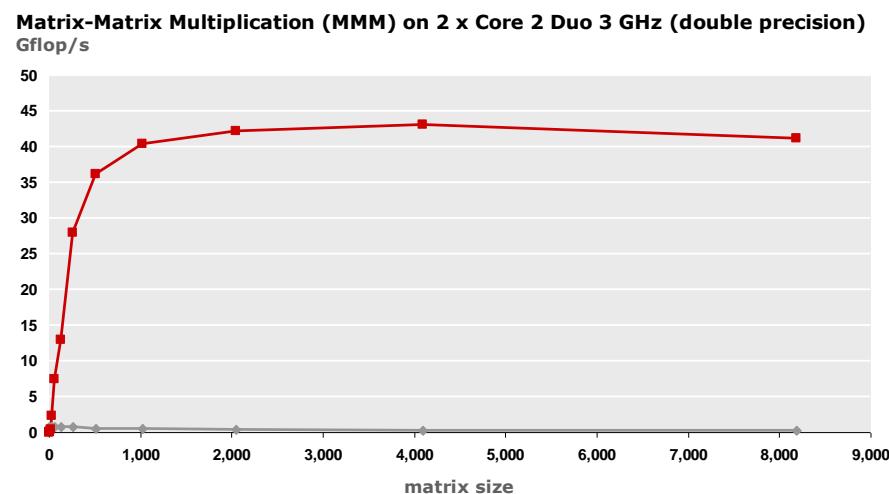
- Definitions are not precise
- An algorithm with high reuse is called *CPU bound*
 - Most time is spent computing
 - Will run faster if CPU is faster
- An algorithm with low reuse is called *memory bound*
 - Most time spent transferring data in the memory hierarchy
 - Will run faster if memory bus is faster
- *Performance optimization:* Make sure that high reuse actually translates into few cache misses, i.e., into temporal locality with respect to the cache

Effects

FFT: $O(\log(n))$ reuse



MMM: $O(n)$ reuse



Up to 40-50% peak

Performance drop outside L2 cache

Most time spent transferring data

Up to 80-90% peak

Performance can be maintained

Cache miss time compensated/hidden by computation

How to Write Fast Numerical Code

Spring 2011
Lecture 7

Instructor: Markus Püschel

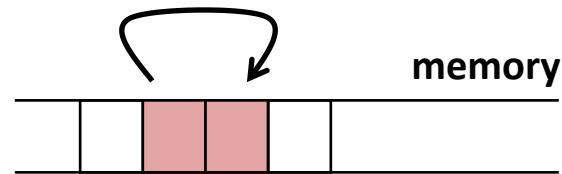
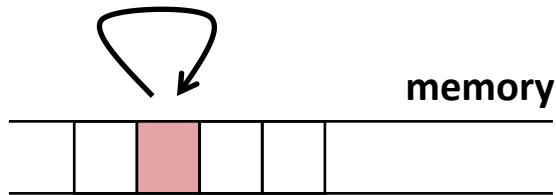
TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

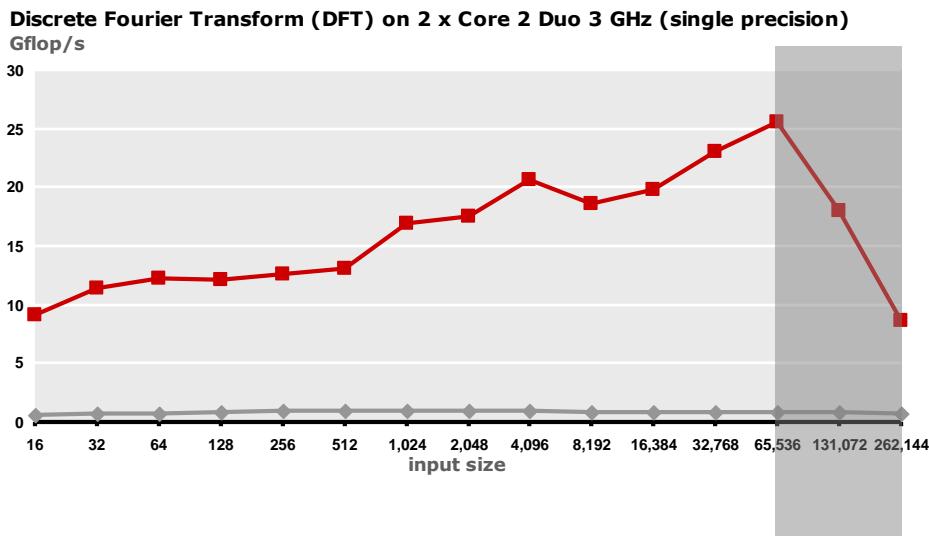
Last Time: Locality

- Temporal and Spatial

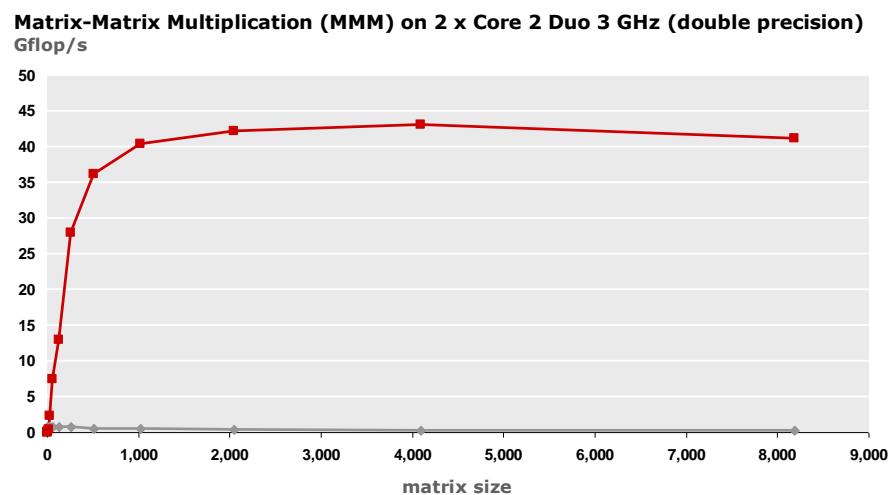


Last Time: Reuse

FFT: $O(\log(n))$ reuse



MMM: $O(n)$ reuse



Today

- Caches

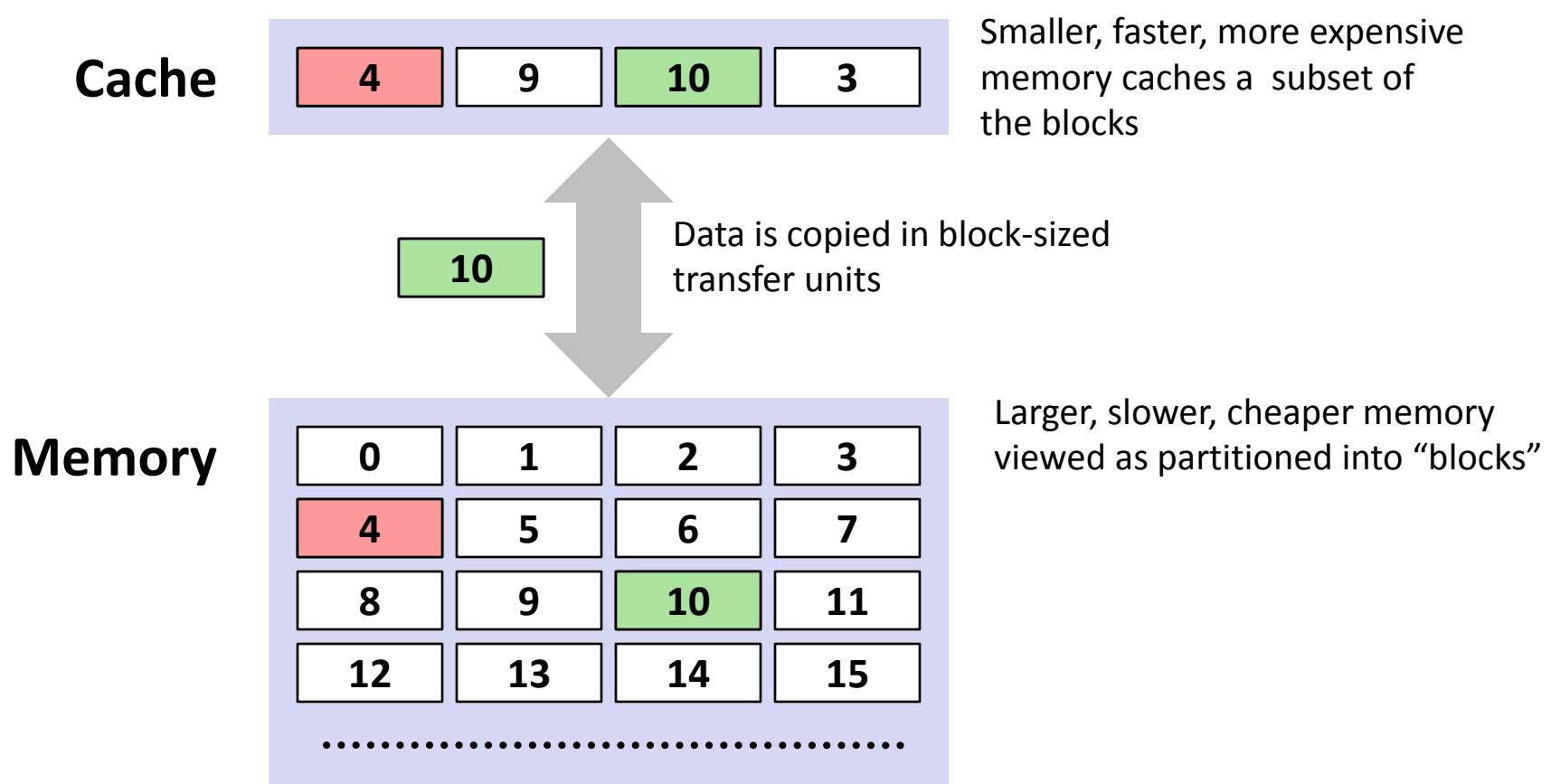
Cache

- ***Definition:*** Computer memory with short access time used for the storage of frequently or recently used instructions or data

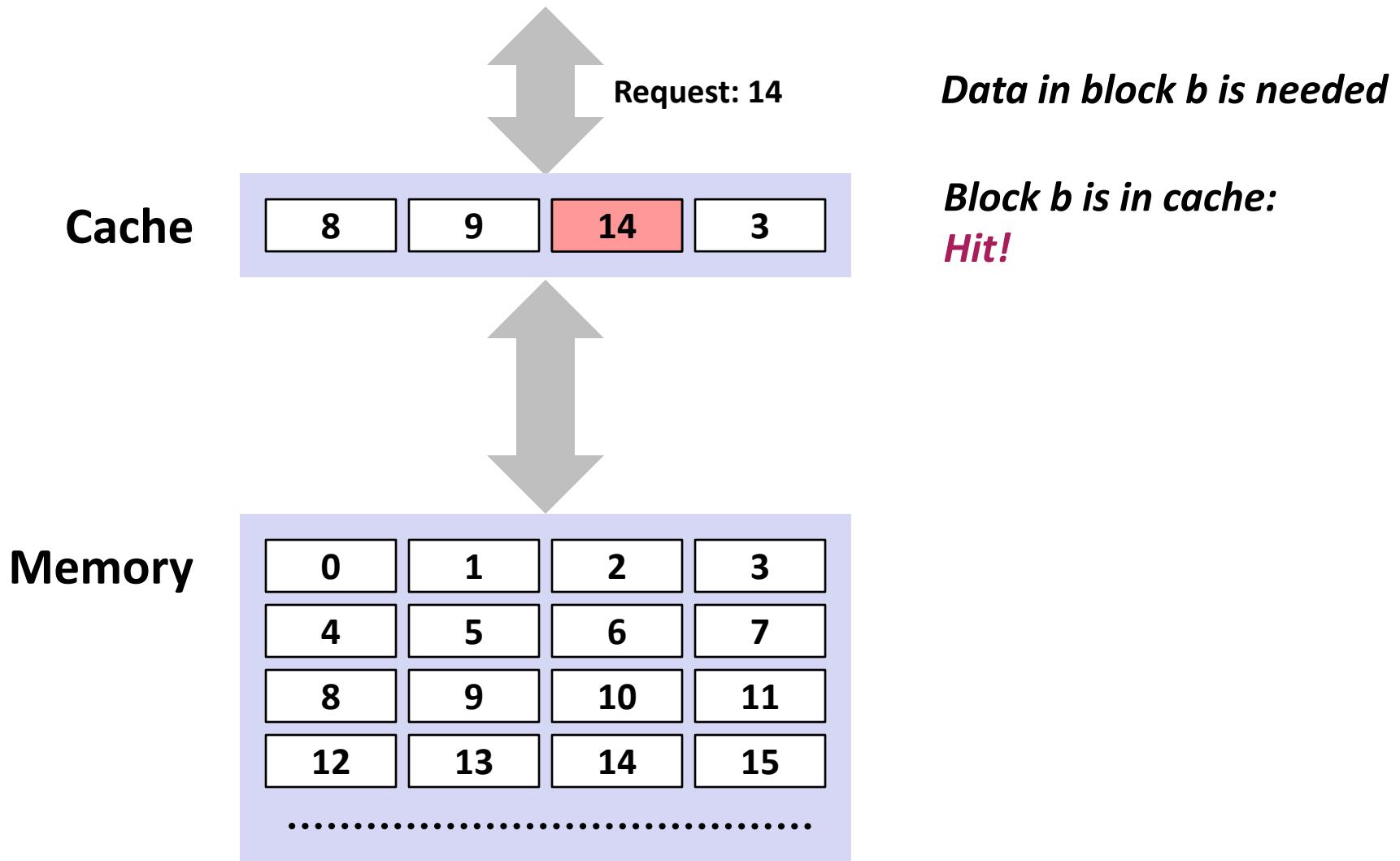


- Naturally supports ***temporal locality***
- ***Spatial locality*** is supported by transferring data in blocks
 - Core 2: one block = 64 B = 8 doubles

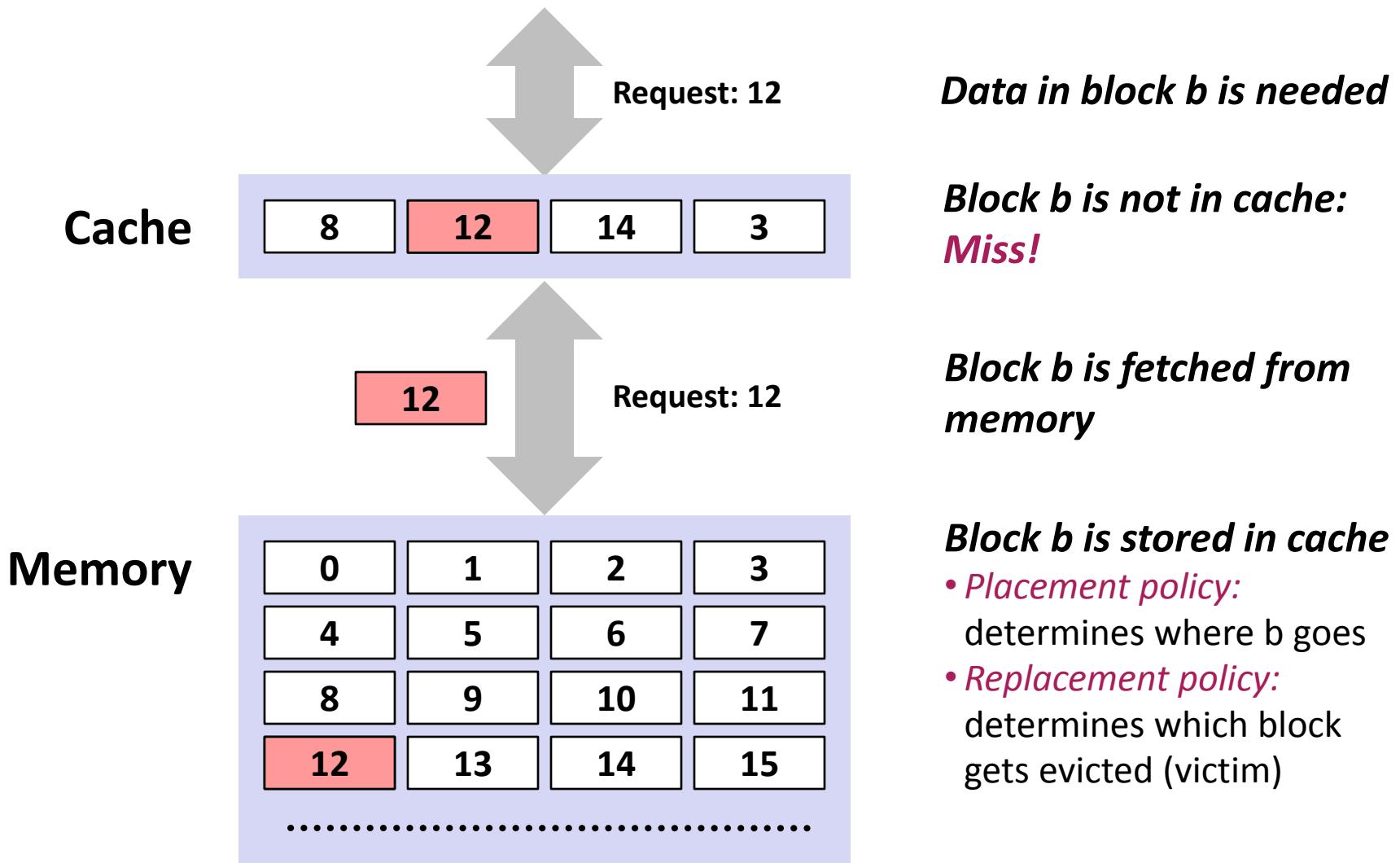
General Cache Mechanics



General Cache Concepts: Hit



General Cache Concepts: Miss



Types of Cache Misses (The 3 C's)

- ***Compulsory (cold)*** miss
 - Occurs on first access to a block
- ***Capacity*** miss
 - Occurs when working set is larger than the cache
- ***Conflict*** miss
 - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot

Cache Performance Metrics

■ Miss Rate

- Fraction of memory references not found in cache: misses / accesses
 $= 1 - \text{hit rate}$

■ Hit Time

- Time to deliver a block in the cache to the processor
- Core 2:
 - 3 clock cycles for L1
 - 14 clock cycles for L2

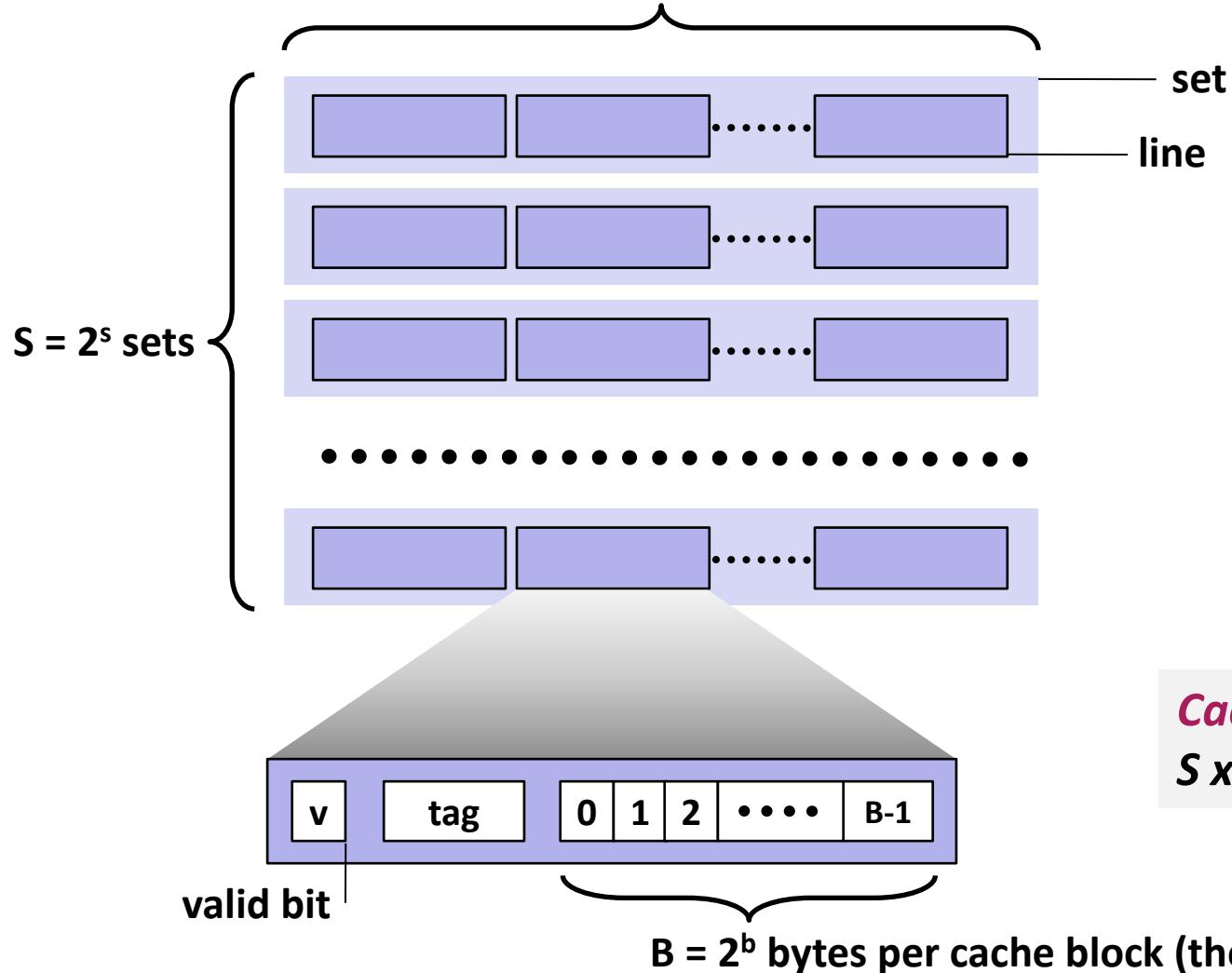
■ Miss Penalty

- Additional time required because of a miss
- Core 2: about 100 cycles for L2 miss

General Cache Organization (S, E, B)

$E = 2^e$ lines per set

$E = \text{associativity}$, $E=1$: direct mapped

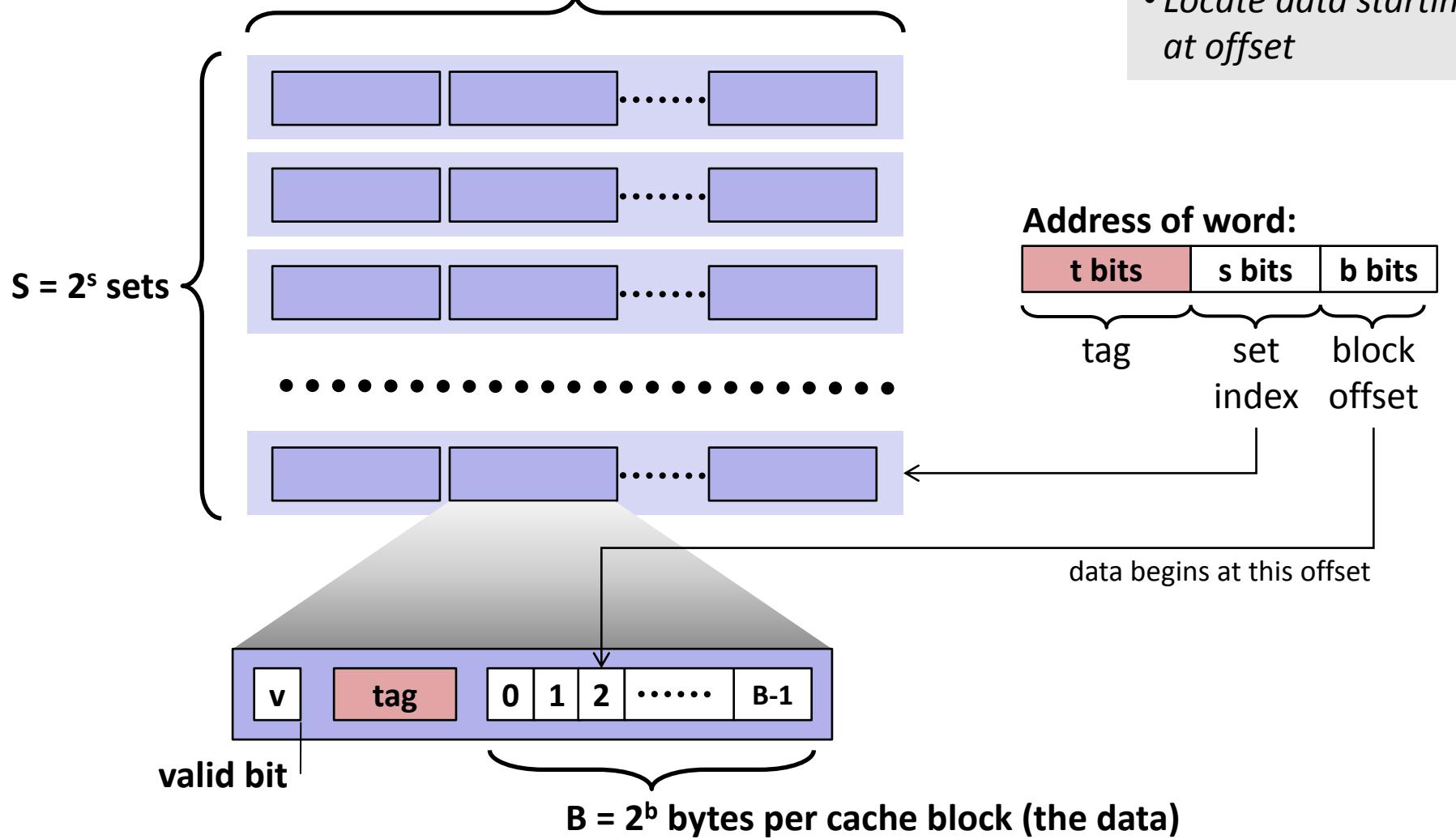


Cache size:
 $S \times E \times B$ data bytes

Cache Read

$E = 2^e$ lines per set

$E = \text{associativity}$, $E=1$: direct mapped



- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Example (S=8, E=1)

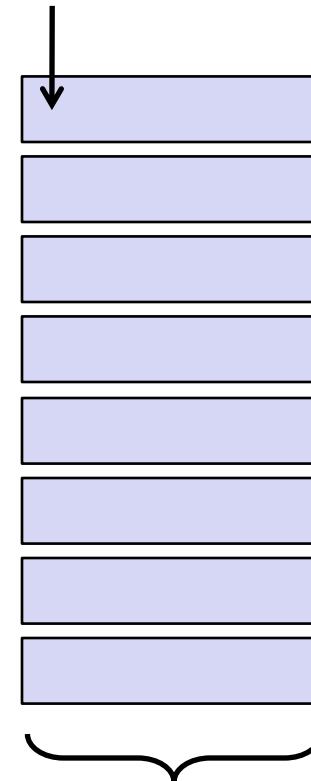
```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j
assume: cold (empty) cache,
 $a[0][0]$ goes here



B = 32 byte = 4 doubles

blackboard

Example (S=4, E=2)

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

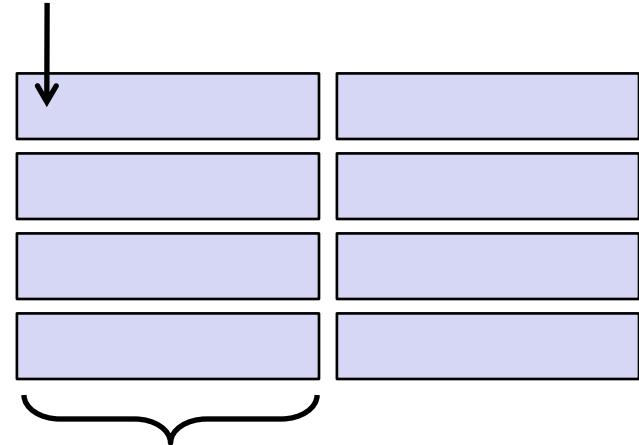
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here



B = 32 byte = 4 doubles

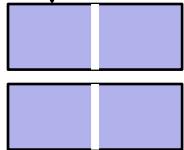
blackboard

What about writes?

- What to do on a write-hit?
 - ***Write-through***: write immediately to memory
 - ***Write-back***: defer write to memory until replacement of line
(needs a valid bit)
- What to do on a write-miss?
 - ***Write-allocate***: load into cache, update line in cache
 - ***No-write-allocate***: writes immediately to memory
- Core 2:
 - Write-back + Write-allocate

Small Example, Part 1

$x[0]$



Cache:

$E = 1$ (direct mapped)
 $S = 2$
 $B = 16$ (2 doubles)

Array (accessed twice in example)

$x = x[0], \dots, x[7]$

```
% Matlab style code
for j = 0:1
    for i = 0:7
        access(x[i])
```

Access pattern: 0123456701234567
Hit/Miss: MHMHMHMHMHMHMHMHMH

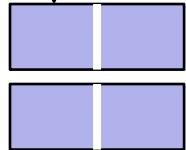
Result: 8 misses, 8 hits

Spatial locality: yes

Temporal locality: no

Small Example, Part 2

x[0]
↓



Cache:

E = 1 (direct mapped)
S = 2
B = 16 (2 doubles)

Array (accessed twice in example)

x = x[0], ..., x[7]

```
% Matlab style code
for j = 0:1
    for i = 0:2:7
        access(x[i])
    for i = 1:2:7
        access(x[i])
```

Access pattern: 0246135702461357
Hit/Miss: M M M M M M M M M M M M M M M M

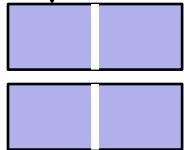
Result: 16 misses

Spatial locality: no

Temporal locality: no

Small Example, Part 3

$x[0]$



Cache:

$E = 1$ (direct mapped)
 $S = 2$
 $B = 16$ (2 doubles)

Array (accessed twice in example)

$x = x[0], \dots, x[7]$

```
% Matlab style code
for j = 0:1
    for k = 0:1
        for i = 0:3
            access(x[i+4*j])
```

Access pattern: 0123012345674567
Hit/Miss: MHMHMHMHMHMHMHMHMH

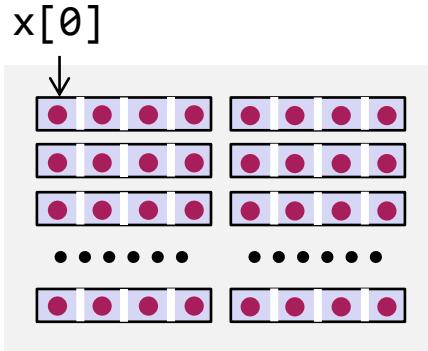
Result: 4 misses, 8 hits (is optimal, why?)

Spatial locality: yes

Temporal locality: yes

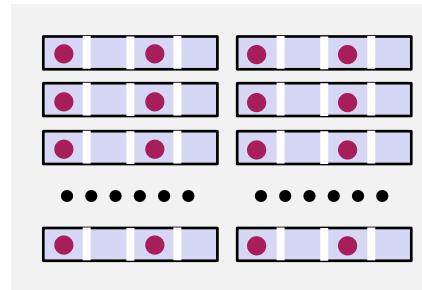
The Killer: Two-Power Strided Access

$x = x[0], \dots, x[n-1]$, $n \gg \text{cache size}$



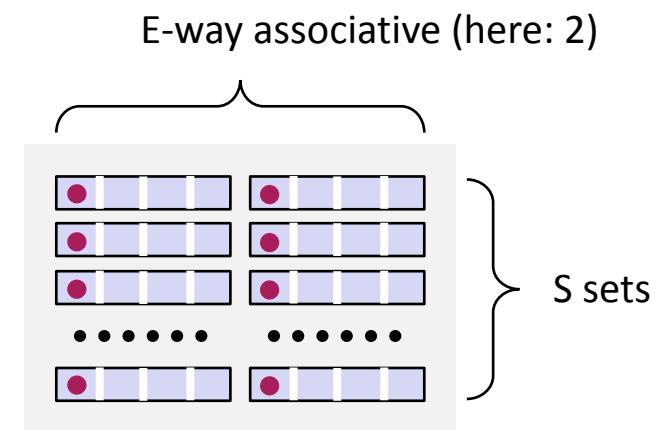
Stride 1: 0123...

Spatial locality
Full cache used



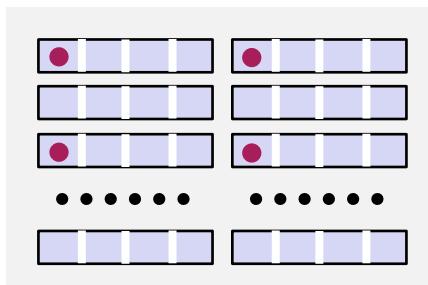
Stride 2: 0 2 4 6 ...

Some spatial locality
1/2 cache used



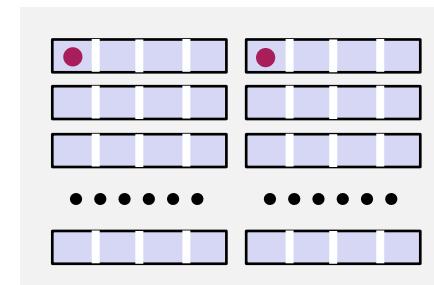
Stride 4: 0 4 8 12 ...

No spatial locality
1/4 cache used



Stride 8: 0 8 16 24 ...

No spatial locality
1/8 cache used



Stride 4S: 0 4S 8S 16S ...

No spatial locality
1/(4S) cache used

*Same for
larger stride*

The Killer: Where Does It Occur?

- Accessing two-power size 2D arrays (e.g., images) columnwise
 - 2d Transforms
 - Stencil computations
 - Correlations
- Various transform algorithms
 - Fast Fourier transform
 - Wavelet transforms
 - Filter banks

Today

- Linear algebra software: history, LAPACK and BLAS
- Blocking: key to performance
- MMM
- ATLAS: MMM program generator

Linear Algebra Algorithms: Examples

- Solving systems of linear equations
 - Eigenvalue problems
 - Singular value decomposition
 - LU/Cholesky/QR/... decompositions
 - ... and many others
-
- Make up most of the numerical computation across disciplines (sciences, computer science, engineering)
 - Efficient software is extremely relevant

The Path to LAPACK

■ EISPACK and LINPACK

- Libraries for linear algebra algorithms
- Developed in the early 70s
- Jack Dongarra, Jim Bunch, Cleve Moler, Pete Stewart, ...
- LINPACK still used as benchmark for the [TOP500](#) ([Wiki](#)) list of most powerful supercomputers

■ Problem:

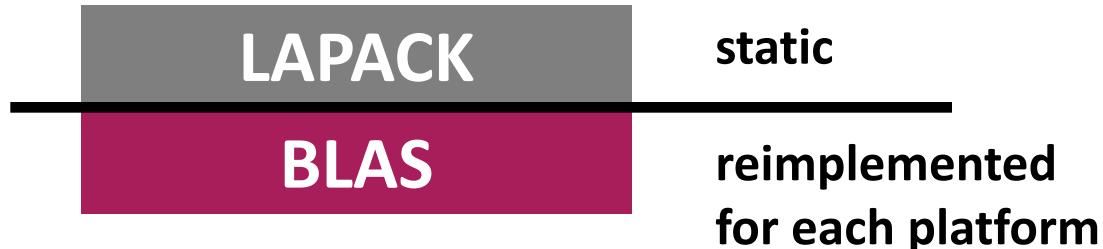
- Implementation “vector-based,” i.e., little locality in data access
- Low performance on computers with deep memory hierarchy
- Became apparent in the 80s

■ Solution: LAPACK

- Reimplement the algorithms “block-based,” i.e., with locality
- Developed late 1980s, early 1990s
- Jim Demmel, Jack Dongarra et al.

LAPACK and BLAS

- Basic Idea:



- Basic Linear Algebra Subroutines (BLAS, [list](#))

- BLAS 1: vector-vector operations (e.g., vector sum) *Reuse: O(1)*
- BLAS 2: matrix-vector operations (e.g., matrix-vector product) *Reuse: O(1)*
- BLAS 3: matrix-matrix operations (e.g., MMM) *Reuse: O(n)*

- LAPACK implemented on top of BLAS

- Using BLAS 3 as much as possible

Why is BLAS3 so important?

- Using BLAS3 = blocking = enabling reuse
- Cache analysis for blocking MMM (blackboard)
- ***Blocking*** (for the memory hierarchy) is the single most important optimization for dense linear algebra algorithms
- ***Unfortunately:*** The introduction of multicore processors requires a reimplementation of LAPACK
just multithreading BLAS is not good enough

How to Write Fast Numerical Code

Spring 2011
Lecture 8

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Reuse (Inherent Temporal Locality)

■ *Reuse of an algorithm:*

$$\frac{\text{Number of operations}}{\text{Size of input} + \text{size of output data}}$$

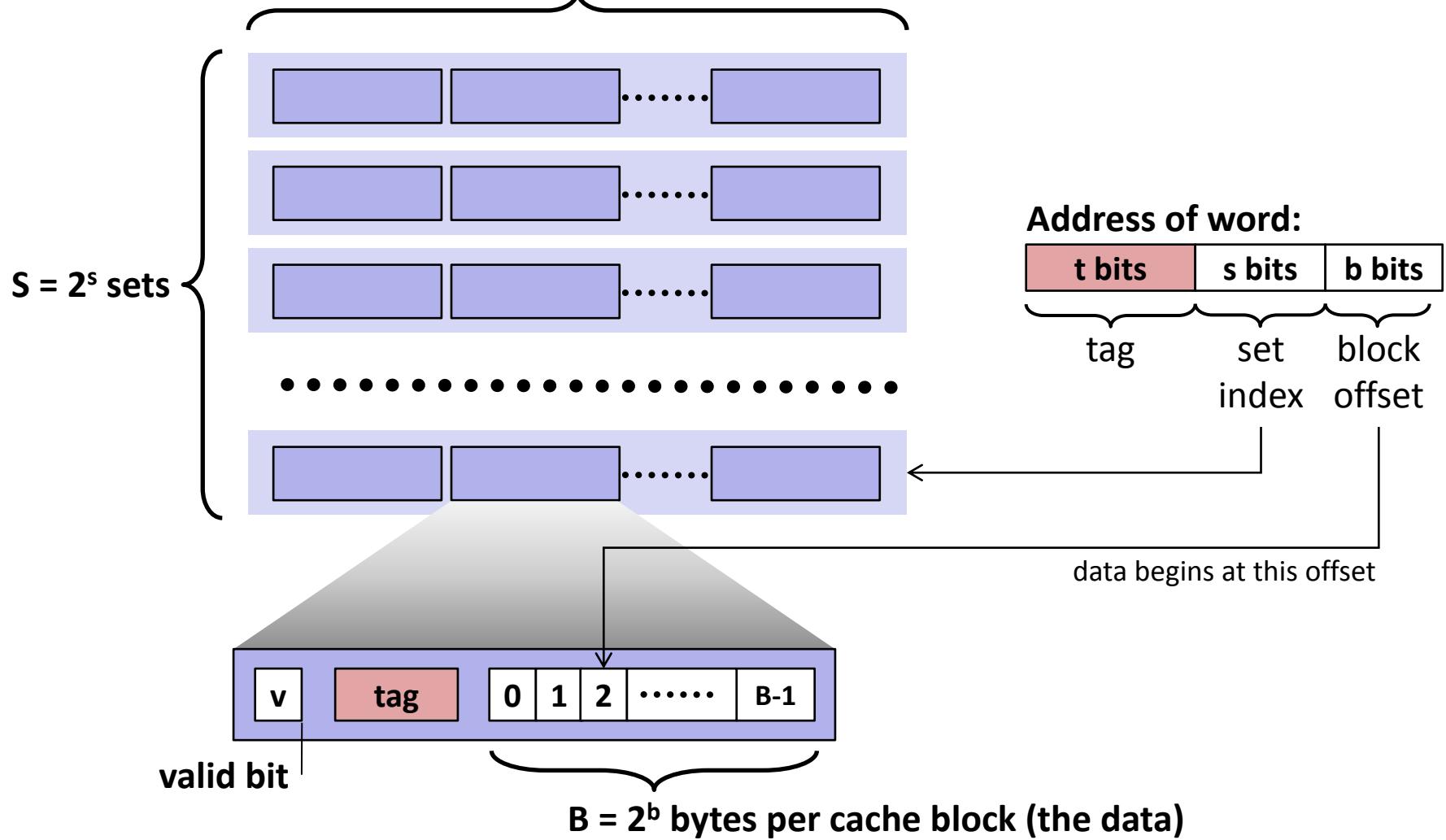

Minimal number of
Memory accesses

■ Examples:

Last Time: Caches

$E = 2^e$ lines per set

$E = \text{associativity}$, $E=1$: direct mapped



Last Time: Blocking

A diagram illustrating matrix multiplication. On the left, there is a large gray rectangle representing a matrix. An equals sign follows it. To the right of the equals sign is another large gray rectangle. A multiplication sign (*) is placed between the second rectangle and a third gray rectangle. Above the third rectangle, a bracket labeled n indicates its width. To the right of the multiplication operation, the text *Cache misses* is written in red, and the formula $(9/8)n^3$ is displayed.

Cache misses

$$(9/8)n^3$$

A diagram illustrating matrix multiplication using blocking. On the left, there is a large gray rectangle. An equals sign follows it. To the right of the equals sign is another large gray rectangle. A multiplication sign (*) is placed between the second rectangle and a third gray rectangle. The third gray rectangle is divided into four vertical blocks of equal width. To the right of the multiplication operation, the formula $n^3/(4B)$ is displayed.

$$n^3/(4B)$$

Today

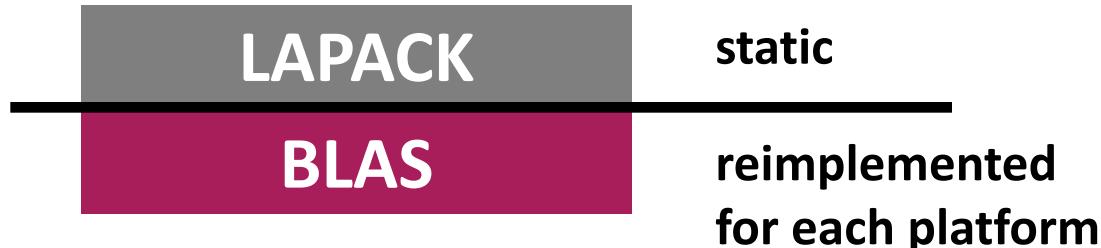
- Linear algebra software: LAPACK and BLAS
- MMM
- ATLAS: MMM program generator

Linear Algebra Algorithms: Examples

- Solving systems of linear equations
 - Eigenvalue problems
 - Singular value decomposition
 - LU/Cholesky/QR/... decompositions
 - ... and many others
-
- Make up most of the numerical computation across disciplines (sciences, computer science, engineering)
 - Efficient software is extremely relevant

LAPACK and BLAS

- Basic Idea:



- Basic Linear Algebra Subroutines (BLAS, [list](#))

- BLAS 1: vector-vector operations (e.g., vector sum) *Reuse: O(1)*
- BLAS 2: matrix-vector operations (e.g., matrix-vector product) *Reuse: O(1)*
- BLAS 3: matrix-matrix operations (e.g., MMM) *Reuse: O(n)*

- LAPACK implemented on top of BLAS

- Using BLAS 3 as much as possible

Why is BLAS3 so important?

- Using BLAS3 = blocking
- Reuse $O(1) \rightarrow O(n)$
- Cache analysis for blocking MMM (blackboard)
- ***Blocking*** (for the memory hierarchy) is the single most important optimization for dense linear algebra algorithms
- ***Unfortunately:*** The introduction of multicore processors requires a reimplemention of LAPACK
just multithreading BLAS is not good enough

Matlab

- Invented in the late 70s by Cleve Moler
- Commercialized (MathWorks) in 84
- Motivation: Make LINPACK, EISPACK easy to use
- Matlab uses LAPACK and other libraries but can only call it *if you operate with matrices and vectors and do not write your own loops*
 - A^*B (calls MMM routine)
 - $A\backslash b$ (calls linear system solver)

Today

- Linear algebra software: history, LAPACK and BLAS
- **MMM**
- ATLAS: MMM program generator

MMM by Definition

- Usually computed as $C = AB + C$
- Cost as computed before
 - n^3 multiplications + n^3 additions = $2n^3$ floating point operations
 - = $O(n^3)$ runtime
- Blocking
 - Increases locality (see previous example)
 - Does not decrease cost
- Can we do better?

Strassen's Algorithm

- Strassen, V. "Gaussian Elimination is Not Optimal," *Numerische Mathematik* 13, 354-356, 1969
Until then, MMM was thought to be $\Theta(n^3)$
- Recurrence $T(n) = 7T(n/2) + O(n^2)$:
Multiplies two $n \times n$ matrices in $O(n^{\log_2(7)}) \approx O(n^{2.808})$
- Crossover point, in terms of cost: $n=654$, but ...
 - Structure more complex → performance crossover much later
 - Numerical stability inferior
- Can we do better?

MMM Complexity: What is known

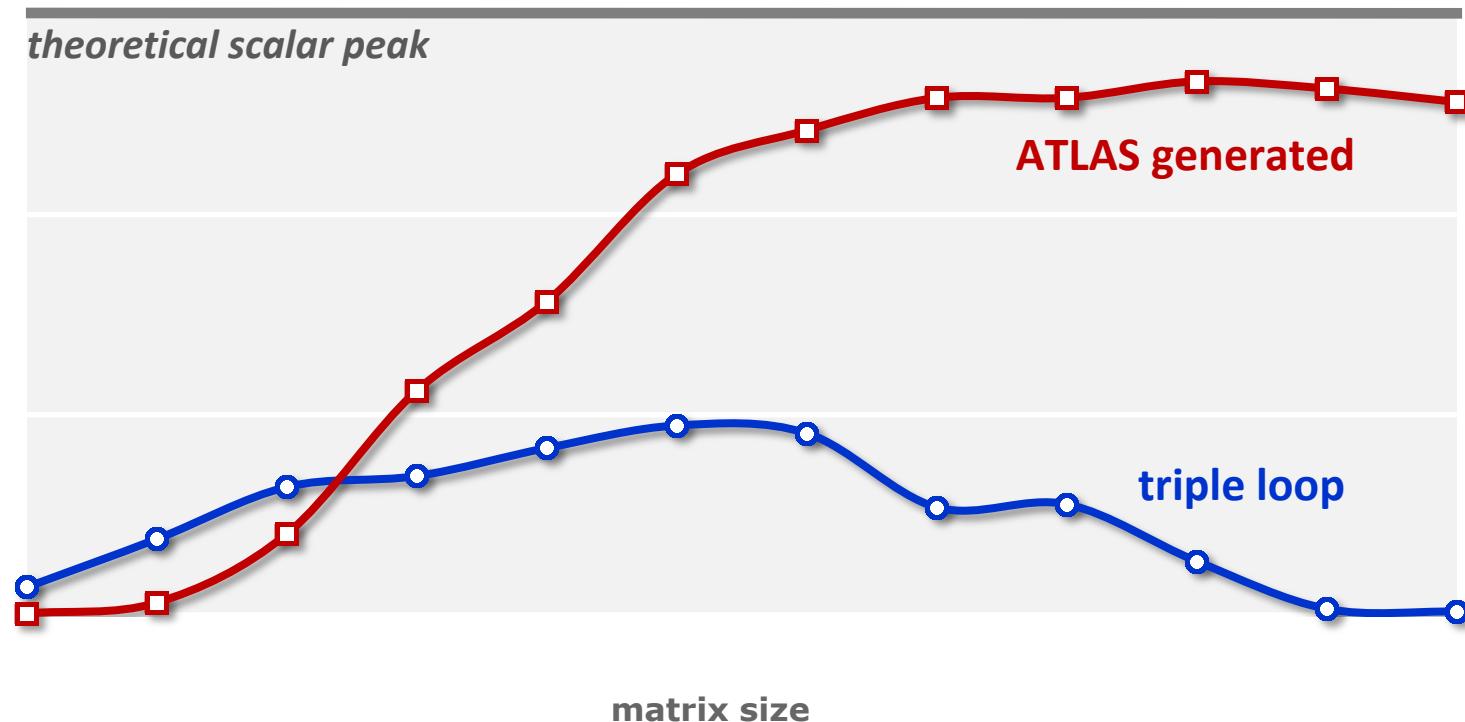
- Coppersmith, D. and Winograd, S. "Matrix Multiplication via Arithmetic Programming," *J. Symb. Comput.* 9, 251-280, 1990
- MMM is $O(n^{2.376})$
- MMM is obviously $\Omega(n^2)$
- It could well be $\Theta(n^2)$
- Compare this to matrix-vector multiplication:
 - Known to be $\Theta(n^2)$ (Winograd), i.e., boring

Today

- Linear algebra software: history, LAPACK and BLAS
- MMM
- ATLAS: MMM program generator

MMM: Memory Hierarchy Optimization

MMM (square real double) Core 2 Duo 3Ghz



- Intel compiler icc –O2
- Huge performance difference for large sizes
- Great case study to learn memory hierarchy optimization

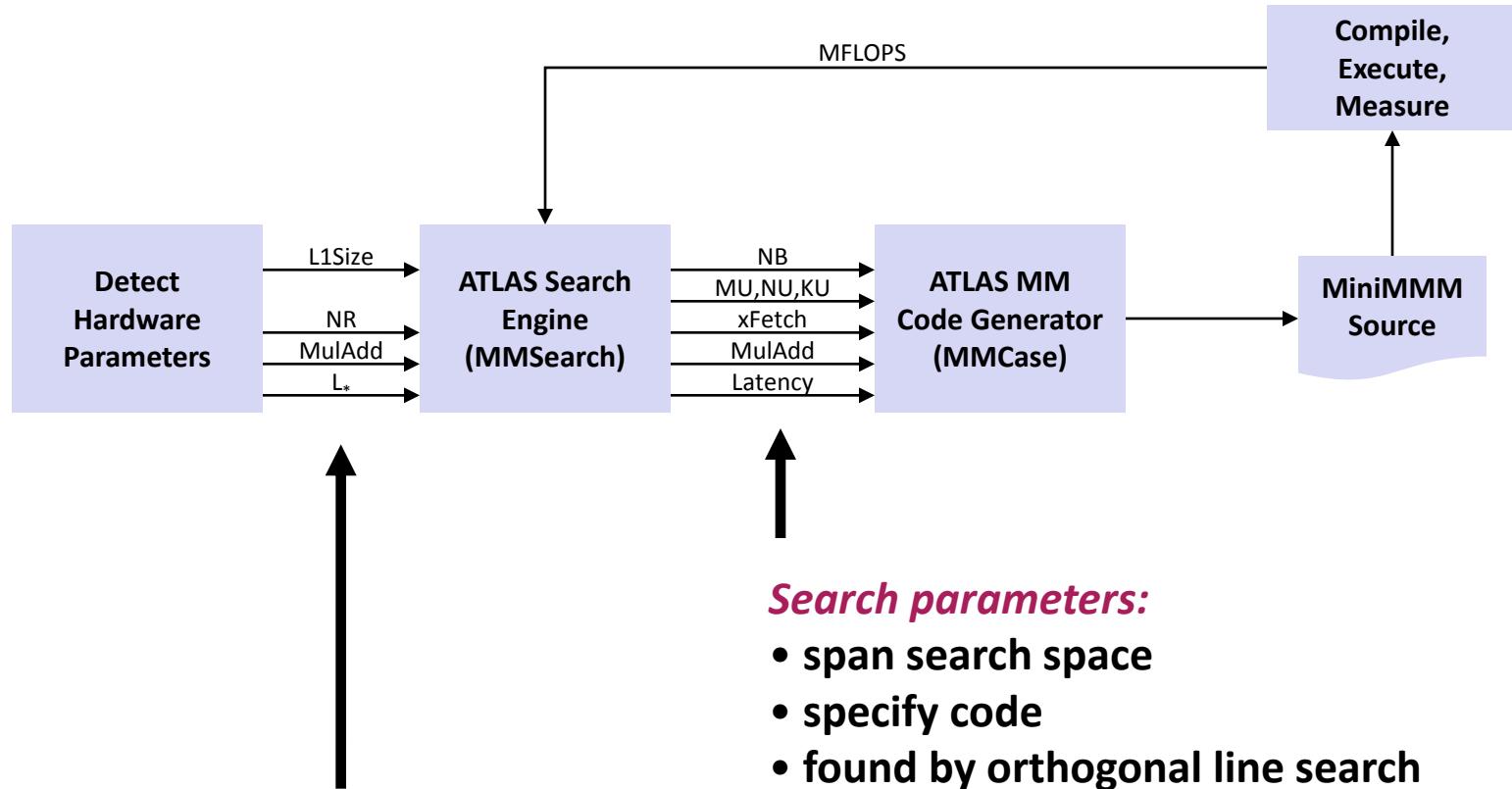
ATLAS

- Successor of PhiPAC, BLAS program generator ([web](#))
- Idea: automatic porting



- People can also contribute handwritten code
- The generator uses empirical search over implementation alternatives to find the fastest implementation
no vectorization or parallelization: so not really used anymore
- We focus on BLAS3 MMM
- Search only over cost $2n^3$ algorithms
(cost equal to triple loop)

ATLAS Architecture



Hardware parameters:

- L1Size: size of L1 data cache
- NR: number of registers
- MulAdd: fused multiply-add available?
- L_* : latency of FP multiplication

How ATLAS Works

- Blackboard
- References:
 - "Automated Empirical Optimization of Software and the ATLAS project" by R. Clint Whaley, Antoine Petitet and Jack Dongarra. *Parallel Computing*, 27(1-2):3-35, 2001
 - K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, Is Search Really Necessary to Generate High-Performance BLAS?, Proceedings of the IEEE, 93(2), pp. 358–386, 2005. Link.

Our presentation is based on this paper

How to Write Fast Numerical Code

Spring 2011
Lecture 10

Instructor: Markus Püschel

TA: Georg Ofenbeck

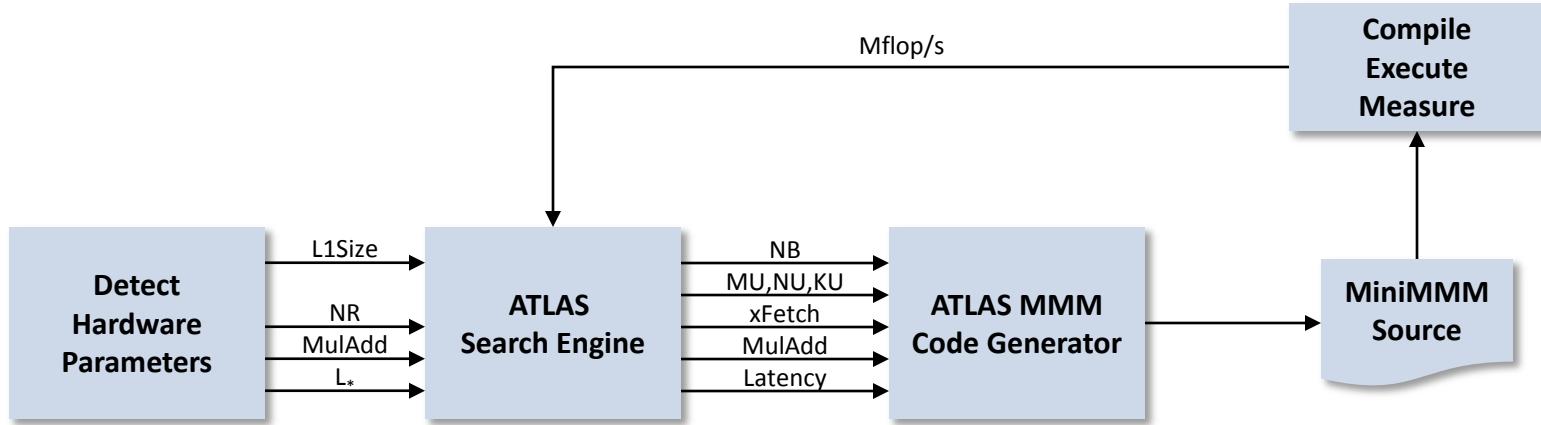


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Today

- ATLAS: Principles
- Model-based ATLAS
- K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill,
Is Search Really Necessary to Generate High-Performance BLAS?,
Proceedings of the IEEE, 93(2), pp. 358–386, 2005. [Link](#).

Last Time: ATLAS



- Blocks MMM into mini-MMMs
- Searches for fastest (highest-performance) mini-MMM
- Mini-MMM choices encoded by parameters (N_B , M_U , N_U , ...)
- Parameter space bounded through microarchitecture parameters
for example: $N_B \leq \sqrt{\text{cache size}}$

How it Worked: From Triple Loop to ...

```
// MMM loop-nest
```

```
for i = 0:NB:N-1  
  for j = 0:NB:M-1  
    for k = 0:NB:K-1
```

- *ij or ji depending on N and M*
- *Blocking for cache*

```
// mini-MMM loop nest
```

```
for i' = i:MU:i+NB-1  
  for j' = j:NU:j+NB-1  
    for k' = k:KU:k+NB-1
```

- *Blocking for registers*

```
// micro-MMM loop nest
```

```
for k'' = k':1:k'+KU-1  
  for i'' = i':1:i'+MU-1  
    for j'' = j':1:j'+NU-1
```

- *unrolling*
- *scalar replacement*
- *add/mult interleaving*
- *skewing*

Search parameters: $N_B, M_U, N_U, K_U, L_S, \dots$

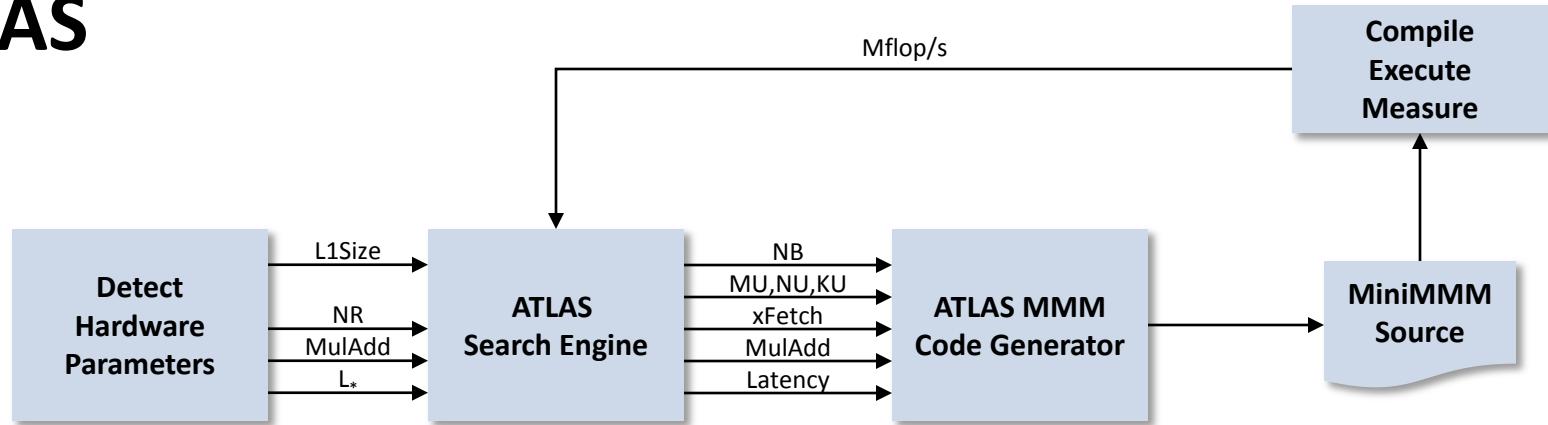
Principles used in ATLAS Optimization

- **Optimization for memory hierarchy = increasing locality**
 - Blocking for cache, blocking for registers
 - Done by loop tiling and loop exchange
- **Fast basic blocks for small sizes (micro-MMM):**
 - Loop order chosen for ILP
 - Loop unrolling to enable scalar replacement
(enables register allocation and instruction scheduling)
 - Add/mult interleaving and skewing (improving ILP)
- **Search for the fastest (mini-MMM) over a relevant set of algorithm/implementation alternatives**

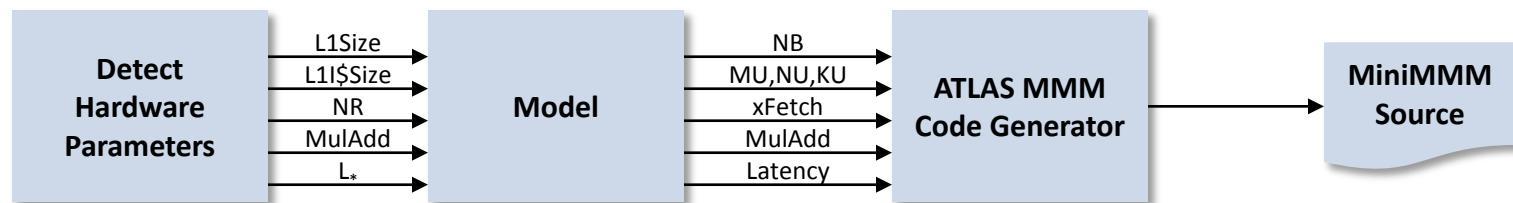
MMM: So far

- We learned a set of optimization techniques for the memory hierarchy
- But there are degrees of freedom
- *Practical problem:* How to choose them without using search?
- *Scientific problem:* How to choose them from an understanding of the microarchitecture?

ATLAS



Model-Based ATLAS



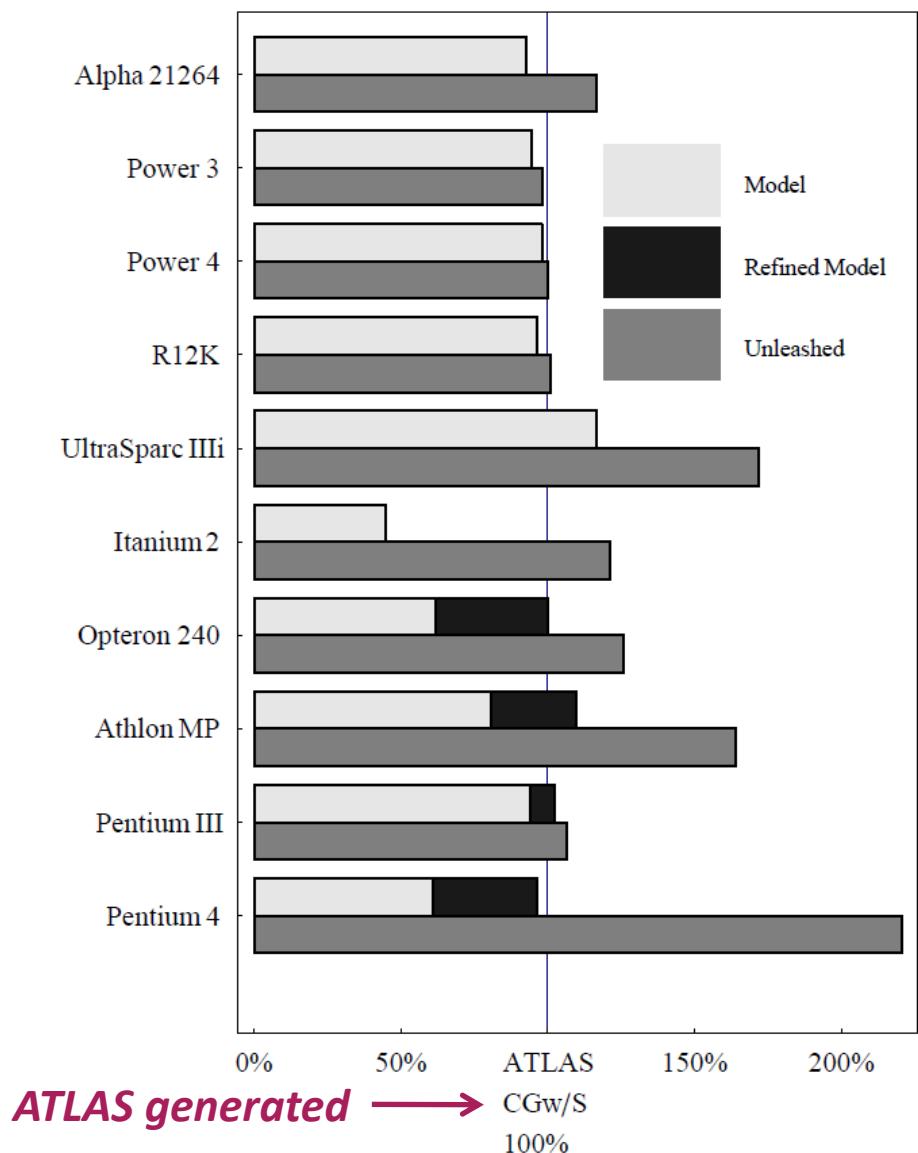
- Search for parameters replaced by model to compute them
- More hardware parameters needed

Model-Based ATLAS: Details

- Blackboard

Experiments

- *Unleashed*: Not generated = hand-written contributed code
- *Refined model* for computing register tiles on x86
- Blocking is for L1 cache
- *Result*: Model-based is comparable to search-based (except Itanium)



graph: Pingali, Yotov, Cornell U.

How to Write Fast Numerical Code

Spring 2011
Lecture 11

Instructor: Markus Püschel

TA: Georg Ofenbeck



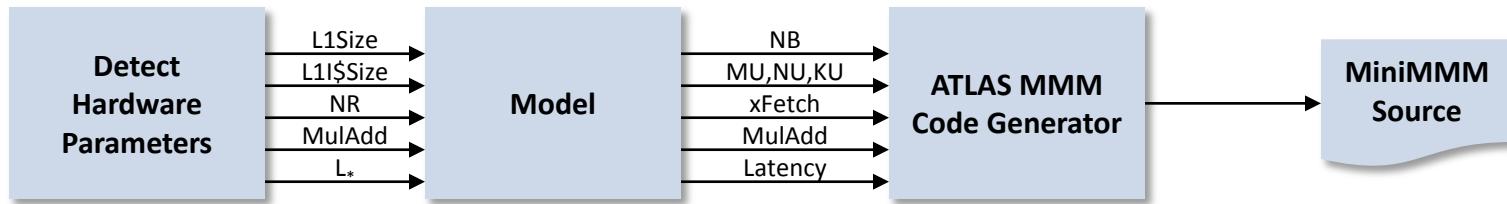
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Miscellaneous

- Intel compiler icc:

- Free for linux
- Who needs it for Visual Studio?

Last Time: Model-Based ATLAS



- Search for parameters replaced by model to compute them
- More hardware parameters needed

Today: Remaining Details

- Register renaming and the refined model for x86
- TLB effects

Dependencies

■ Read-after-write (RAW) or true dependency

$$\begin{array}{ll} W \quad r_1 = r_3 + r_4 & \text{nothing can be done} \\ R \quad r_2 = 2r_1 & \text{no ILP} \end{array}$$

■ Write after read (WAR) or antidependency

$$\begin{array}{ll} R \quad r_1 = r_2 + r_3 & \text{dependency only by} \\ W \quad r_2 = r_4 + r_5 & \text{name} \rightarrow \text{rename} \end{array}$$

$$\begin{array}{l} r_1 = r_2 + r_3 \\ r = r_4 + r_5 \end{array} \quad \text{now ILP}$$

■ Write after write (WAW) or output dependency

$$\begin{array}{ll} W \quad r_1 = r_2 + r_3 \\ \dots \\ W \quad r_1 = r_4 + r_5 & \text{dependency only by} \\ & \text{name} \rightarrow \text{rename} \end{array}$$

$$\begin{array}{l} r_1 = r_2 + r_3 \\ \dots \\ r = r_4 + r_5 \end{array} \quad \text{now ILP}$$

Resolving WAR

$$R \quad r_1 = r_2 + r_3$$

$$W \quad r_2 = r_4 + r_5$$

*dependency only by
name → rename*

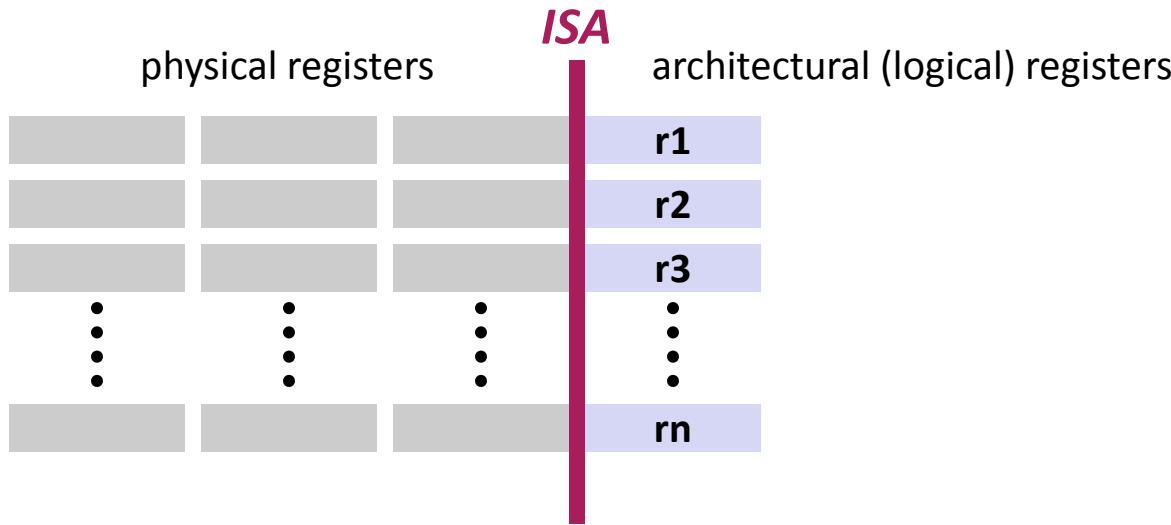
$$r_1 = r_2 + r_3$$

$$r = r_4 + r_5$$

now ILP

- **Compiler:** Use a different register, $r = r_6$
- **Hardware (if supported):** register renaming
 - Requires a separation of architectural and physical registers
 - Requires more physical than architectural registers

Register Renaming



- Hardware manages mapping architectural → physical registers
- More physical than logical registers
- Hence: more instances of each r_i can be created
- Used in superscalar architectures (e.g., Intel Core) to increase ILP by resolving WAR dependencies

Scalar Replacement Again

- How to avoid WAR and WAW in your basic block source code
- Solution: Single static assignment (SSA) code:
 - Each variable is assigned exactly once

no duplicates

```
<more>
s266 = (t287 - t285);
s267 = (t282 + t286);
s268 = (t282 - t286);
s269 = (t284 + t288);
s270 = (t284 - t288);
s271 = (0.5*(t271 + t280));
s272 = (0.5*(t271 - t280));
s273 = (0.5*((t281 + t283) - (t285 + t287)));
s274 = (0.5*(s265 - s266));
t289 = ((9.0*s272) + (5.4*s273));
t290 = ((5.4*s272) + (12.6*s273));
t291 = ((1.8*s271) + (1.2*s274));
t292 = ((1.2*s271) + (2.4*s274));
a122 = (1.8*(t269 - t278));
a123 = (1.8*s267);
a124 = (1.8*s269);
t293 = ((a122 - a123) + a124);
a125 = (1.8*(t267 - t276));
t294 = (a125 + a123 + a124);
t295 = ((a125 - a122) + (3.6*s267));
t296 = (a122 + a125 + (3.6*s269));
<more>
```

Micro-MMM Standard Model

- $MU^*NU + MU + NU \leq NR - \text{ceil}((Lx+1)/2)$
- Core: $MU = 2$, $NU = 3$

reuse in *a, b, c*

- Code sketch ($KU = 1$)

```
rc1 = c[0,0], ..., rc6 = c[1,2] // 6 registers
loop over k {
    load a // 2 registers
    load b // 3 registers
    compute // 6 indep. mults, 6 indep. adds, reuse a and b
}
c[0,0] = rc1, ..., c[1,2] = rc6
```

Extended Model (x86)

- MU = 1, NU = NR - 2 = 14

$$\begin{array}{c} \bullet \\ \hline a \end{array} \quad \begin{array}{c} \bullet \\ \hline b \end{array} = \begin{array}{c} \bullet \\ \hline c \end{array} \quad reuse\ in\ c$$

- Code sketch (KU = 1)

```
rc1 = c[0], ..., rc14 = c[13] // 14 registers
loop over k {
    load a                // 1 register
    {rb = b[1]             // 1 register
     rb = rb*a            // mult (two-operand)
     rc1 = rc1 + rb      // add (two-operand)
     {rb = b[2]             // reuse register (WAR: renaming resolves it)
      rb = rb*a
      rc2 = rc2 + rb
      ...
    }
    c[0] = rc1, ..., c[13] = rc14
```

Summary:

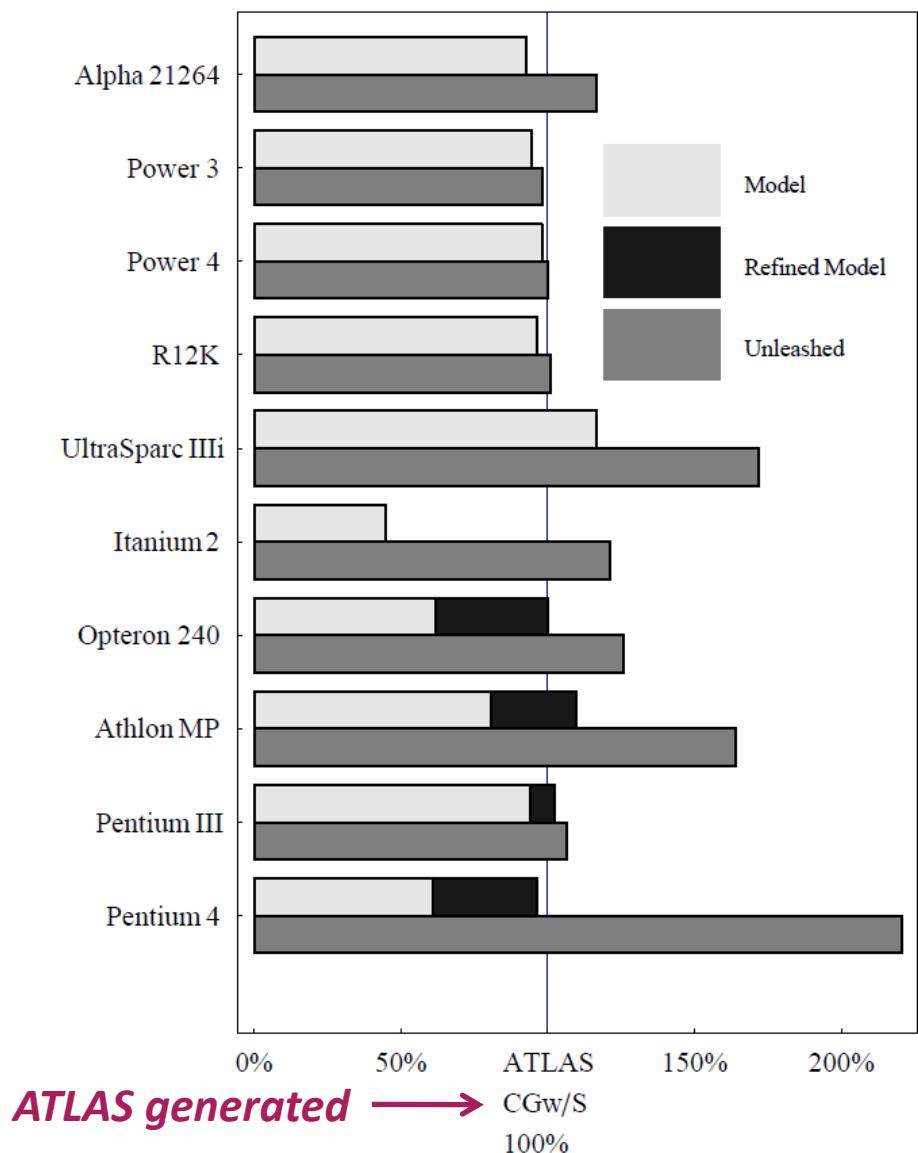
- no reuse in a and b
- + larger tile size for c

Today: Remaining Details

- Register renaming and the refined model for x86
- **TLB effects**
 - Blackboard

Experiments

- *Unleashed*: Not generated = hand-written contributed code
- *Refined model* for computing register tiles on x86
- Blocking is for L1 cache
- *Result*: Model-based is comparable to search-based (except Itanium)



graph: Pingali, Yotov, Cornell U.

How to Write Fast Numerical Code

Spring 2011
Lecture 12

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Miscellaneous

- Start of research project
- No class next Monday, April 11th (Sechseläuten)
- Midterm exam: Friday, April 15th

Today

■ Linear algebra algorithms and optimization

- Solving linear systems (Gauss elimination)
- Matrix inversion
- Determinant

Reminder: LAPACK

- Implements linear algebra algorithms
- Implemented on top of BLAS using BLAS 3 as much as possible (by “blocking” the algorithms)

*Linear system solving
Matrix inversion
Singular value decomposition
... and more*

LAPACK

BLAS

BLAS 1: vector-vector ops

BLAS 2: matrix-vector ops

BLAS 3: matrix-matrix ops

Example: Linear Systems and Related

- Solving linear systems
- PLU factorization
- Matrix inversion
- Determinant

Complexity

- *Source:* Bürgisser, Clausen, Shokrollahi “Algebraic Complexity Theory,” Springer 1997, pp. 426
- *Definition:* $P(n)$, $n > 0$, a sequence of problems (n = problem size), complexity measure = number of adds + mults, then

$$w(P) = \inf(g \mid \text{complexity}(P(n)) = O(n^g))$$

- Problems:
 - $\text{MMM}(n)$: multiplying two $n \times n$ matrices
 - $\text{MInv}(n)$: inverting an $n \times n$ matrix
 - $\text{PLU}(n)$: computing PLU factorization of an $n \times n$ matrix
 - $\text{Det}(n)$: computing the determinant of an $n \times n$ matrix

Complexity Results

- Example (we had that before): $2 \leq w(\text{MMM}(n)) < 2.38$

- *Theorem:*

$$w(\text{MMM}(n)) = w(\text{MInv}(n)) = w(\text{PLU}(n)) = w(\text{Det}(n))$$

- Cost of the usual implementations:

- $\text{MMM}(n) = 2n^3 + O(n^2)$
- $\text{MInv}(n) = 8/3 n^3 + O(n^2)$
- $\text{PLU}(n) = 2/3 n^3 + O(n^2)$
- $\text{Det}(n) = 2/3 n^3 + O(n^2)$

How it's Implemented

- Blackboard

Chapter 2 in James W. Demmel, Applied Numerical Linear Algebra, SIAM, 1997

How to Write Fast Numerical Code

Spring 2011
Lecture 13

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Miscellaneous

- No class next Monday, April 11th (Sechseläuten)
- Midterm exam: Friday, April 15th

Today

- Solving linear systems
- Matrix inversion
- PLU factorization
- Determinant

Linear system solving

Matrix inversion

Singular value decomposition

... and more

LAPACK

BLAS

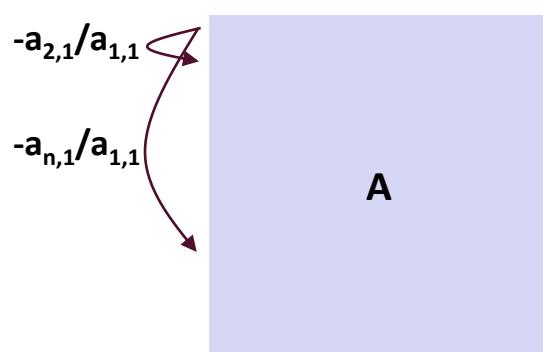
BLAS 1: vector-vector ops

BLAS 2: matrix-vector ops

BLAS 3: matrix-matrix ops

Chapter 2 in James W. Demmel, Applied Numerical Linear Algebra, SIAM, 1997

Gauss Elimination and LU Factorization

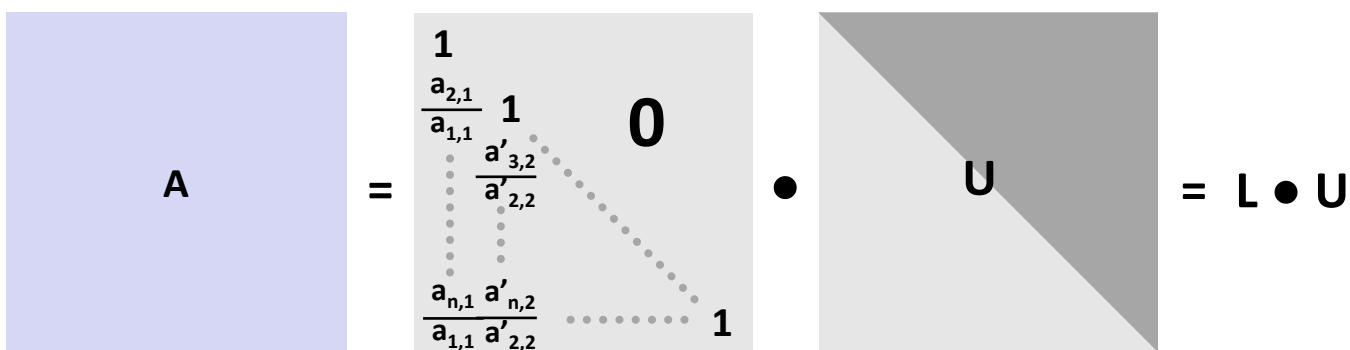
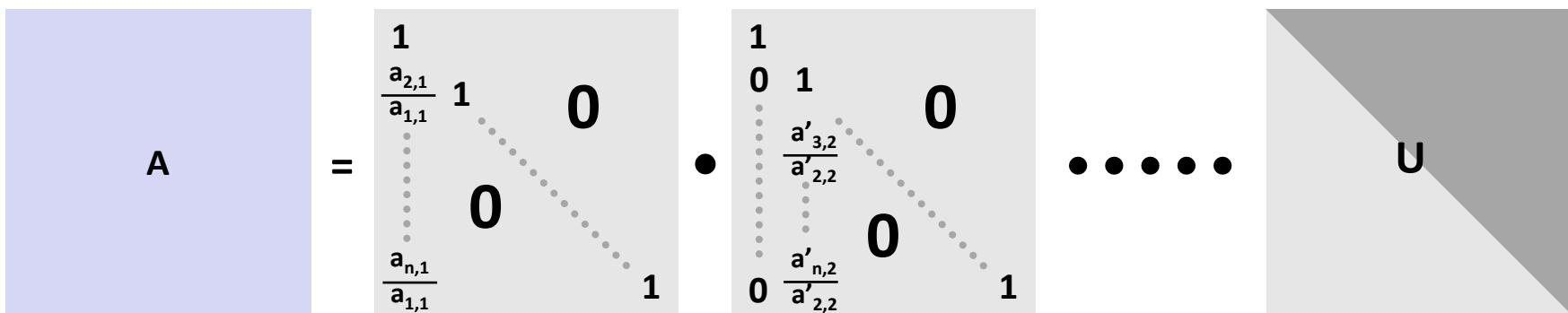
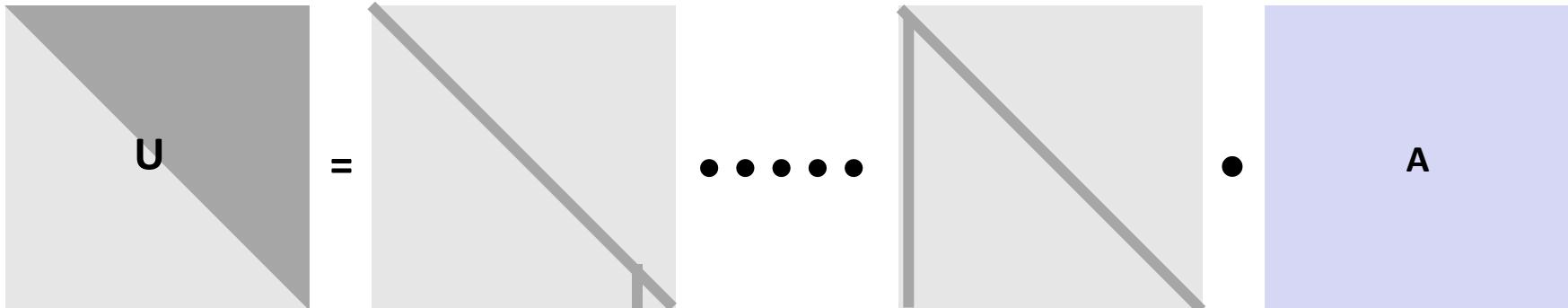


$$A = [a_{k,l}]_{1 \leq k, l \leq n}$$

$$\begin{matrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & & \\ \vdots & & \\ 0 & & \end{matrix} \quad A' = \begin{matrix} 1 & & & \\ -\frac{a_{2,1}}{a_{1,1}} & 1 & & \\ \vdots & \vdots & & \\ -\frac{a_{n,1}}{a_{1,1}} & & 1 & \\ & & & 1 \end{matrix} \bullet \begin{matrix} & & \\ & & \\ & & \\ & & \\ & & \end{matrix} A$$

$$\begin{matrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & a'_{2,2} & \cdots & a'_{1,n} \\ \vdots & \vdots & & \\ 0 & 0 & & \end{matrix} \quad A' = \begin{matrix} 1 & & & \\ 0 & 1 & & \\ \vdots & \vdots & & \\ 0 & -\frac{a'_{3,2}}{a'_{2,2}} & & \\ \vdots & \vdots & & \\ 0 & -\frac{a'_{n,2}}{a'_{2,2}} & & \\ & & & 1 \end{matrix} \bullet \begin{matrix} & & \\ & & \\ & & \\ & & \\ & & \end{matrix} \bullet \begin{matrix} & & \\ & & \\ & & \\ & & \\ & & \end{matrix} A$$

after n-1 steps



Summary

■ Gauss elimination is LU factorization

- We assume that the occurring diagonal elements $a_{1,1}, a'_{2,2}, \dots$ are all $\neq 0$ (otherwise the LU factorization does not exist)
- U = upper triangular
- L = lower triangular (1's on diagonal)
- L contains the multipliers occurring in Gauss elimination

■ Now $Ax = b$ is $LUX = b$ and can be solved in two steps:

- Solve $Ly = b$
- Solve $Ux = y$
- Cost: $n^2 + O(n)$ for each step = $2n^2 + O(n)$

LU Factorization: Algorithm

- From straightforward algorithm to optimized BLAS 3 based one (blackboard)

Chapter 2 in James W. Demmel, Applied Numerical Linear Algebra, SIAM, 1997

How to Write Fast Numerical Code

Spring 2011
Lecture 15

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Reuse Again

■ *Reuse of an algorithm:*

$$\frac{\text{Number of operations}}{\text{Size of input} + \text{size of output data}}$$


Minimal number of
Memory accesses

■ Examples:

- Matrix multiplication $C = AB + C$

$$\frac{2n^3}{3n^2} = \frac{2}{3}n = O(n)$$

- Discrete Fourier transform

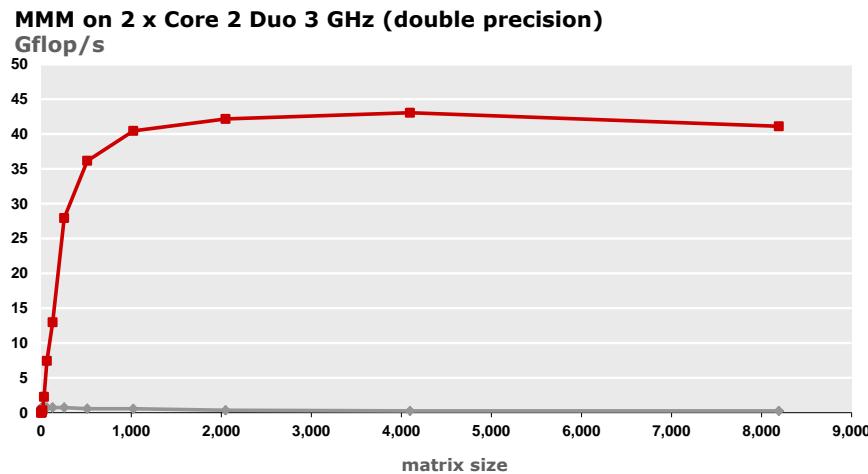
$$\approx \frac{5n \log_2(n)}{2n} = \frac{5}{2} \log_2(n) = O(\log(n))$$

- Adding two vectors $x = x+y$

$$\frac{n}{2n} = \frac{1}{2} = O(1)$$

Effects

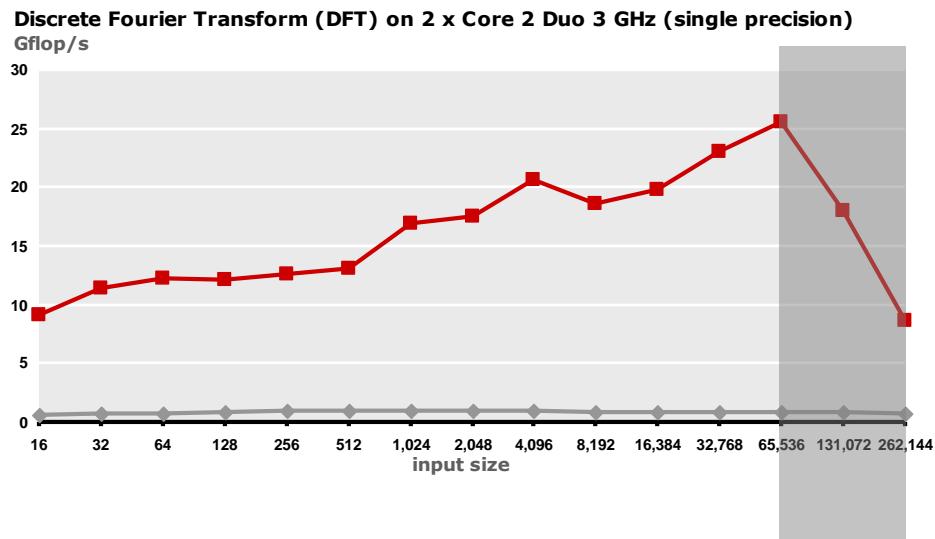
MMM: $O(n)$ reuse



Performance maintained even when data does not fit into caches

Drop will happen once data does not fit into main memory

FFT: $O(\log(n))$ reuse



Performance drop when data does not fit into largest cache

Outside cache: Runtime only determined by memory accesses (memory bound)

Memory Bound Computation

- Typically: Computations with $O(1)$ reuse
- Performance bound based on data traffic may be tighter than performance bound obtained by op count

Example

■ Vector addition: $z = x + y$ on Core 2

```
void vectorsum(double *x, double *y, double *z, int n)
{
    int i;

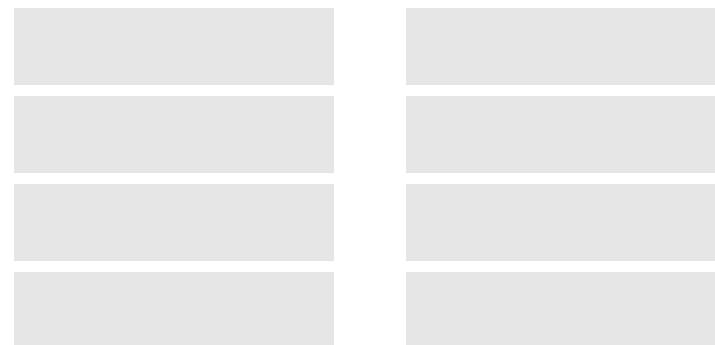
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

Reuse: 1/3

■ Core 2:

- Peak performance (no SSE):
- Throughput L1 cache:
- Throughput L2 cache:
- Throughput Main memory:

Resulting bounds



Example

■ Vector addition: $z = x + y$ on Core 2

```
void vectorsum(double *x, double *y, double *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

Reuse: 1/3

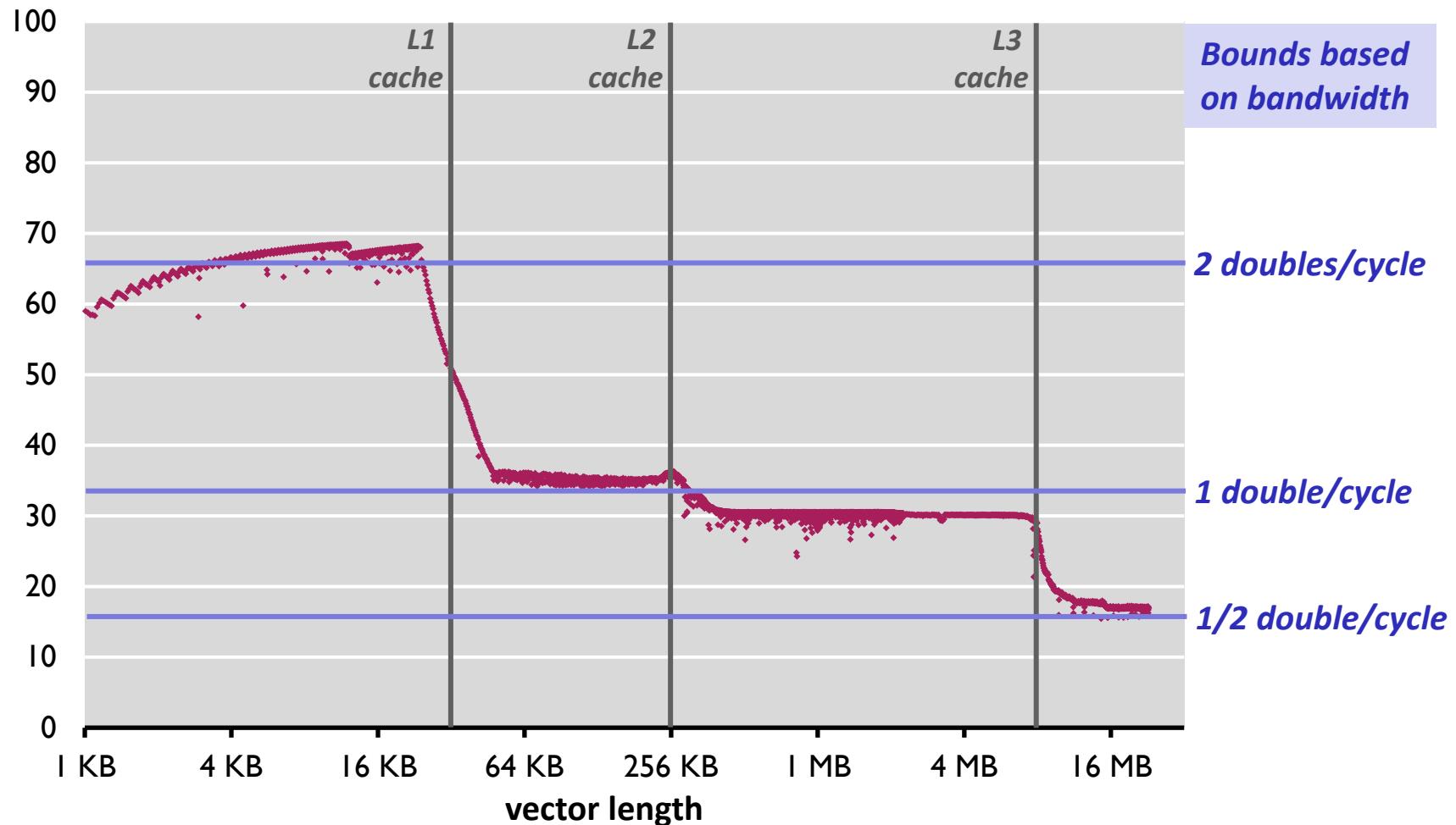
■ Core 2:

- | | <i>Resulting bounds</i> | |
|------------------------------|-----------------------------|------------------------|
| ■ Peak performance (no SSE): | 1 add/cycle | n cycles |
| ■ Throughput L1 cache: | 2 doubles/cycle | $\frac{3}{2} n$ cycles |
| ■ Throughput L2 cache: | 1 doubles/cycle | $3n$ cycles |
| ■ Throughput Main memory: | $\frac{1}{4}$ doubles/cycle | $12n$ cycles |

Memory-Bound Computation

$z = x + y$ on Core i7 (one core, no SSE), `icc 12.0 /O2 /fp:fast /Qipo`

Percentage peak performance (peak = 1 add/cycle)

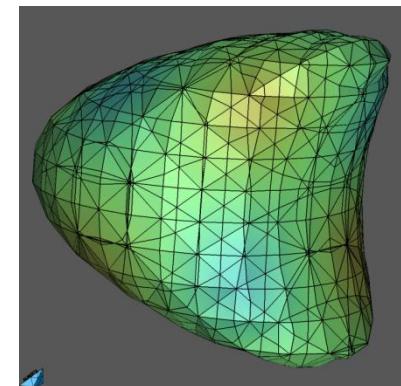
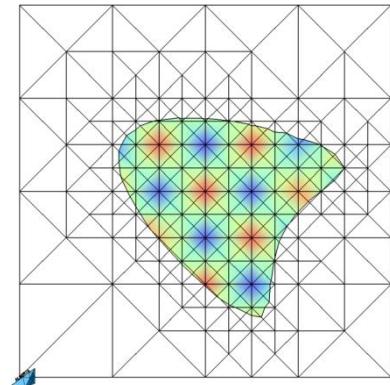


Today

- Sparse matrix-vector multiplication (MVM)
- Sparsity/Bebop
- References:
 - Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. *SPARSITY: An Optimization Framework for Sparse Matrix Kernels*, Int'l Journal of High Performance Comp. App., 18(1), pp. 135-158, 2004
 - Vuduc, R.; Demmel, J.W.; Yelick, K.A.; Kamil, S.; Nishtala, R.; Lee, B.; *Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply*, pp. 26, Supercomputing, 2002
 - [Sparsity/Bebop](#) website

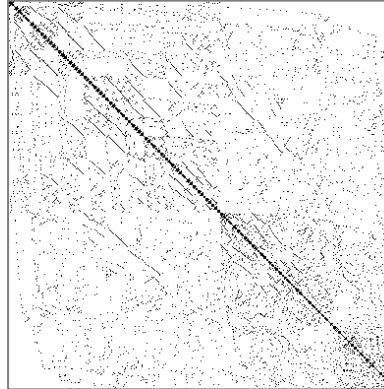
Sparse Linear Algebra

- Very different characteristics from dense linear algebra (LAPACK etc.)
- Applications:
 - finite element methods
 - PDE solving
 - physical/chemical simulation
(e.g., fluid dynamics)
 - linear programming
 - scheduling
 - signal processing (e.g., filters)
 - ...
- Core building block: Sparse MVM



Sparse MVM (SMVM)

- $y = y + Ax$, A sparse but known

$$\begin{array}{c|c|c|c} & = & + & \\ \text{y} & & \text{y} & \\ & & + & \\ & & \text{A} & \\ & & & \bullet \\ & & & \text{x} \end{array}$$


- Typically executed many times for fixed A
- What is reused?
- Reuse dense versus sparse MVM?

Storage of Sparse Matrices

- **Standard storage is obviously inefficient**
 - Many zeros are stored
 - As a consequence, reuse is decreased
- **Several sparse storage formats are available**
- **Most popular: Compressed sparse row (CSR) format**
 - blackboard

CSR

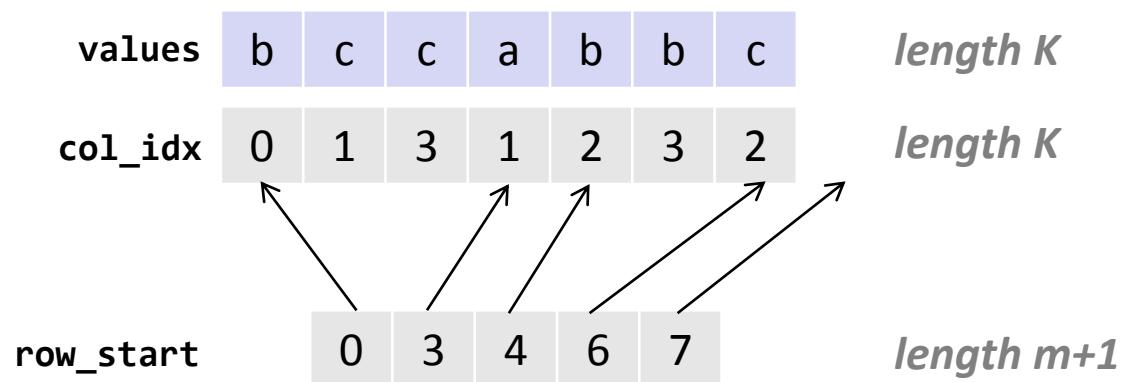
■ Assumptions:

- A is $m \times n$
- K nonzero entries

A as matrix

b	c		c
	a		
		b	b
		c	

A in CSR:



■ Storage: $\Theta(\max(K, m))$, typically $\Theta(K)$

Sparse MVM Using CSR

$$\mathbf{y} = \mathbf{y} + \mathbf{A}\mathbf{x}$$

```
void smvm(int m, const double* value, const int* col_idx,
          const int* row_start, const double* x, double* y)
{
    int i, j;
    double d;

    /* loop over rows */
    for (i = 0; i < m; i++) {
        d = y[i]; /* scalar replacement since reused */

        /* loop over non-zero elements in row i */
        for (j = row_start[i]; j < row_start[i+1]; j++, col_idx++, value++) {
            d += value[j] * x[col_idx[j]];
        }
        y[i] = d;
    }
}
```

CSR + sparse MVM: Advantages?

CSR

- **Advantages:**

- Only nonzero values are stored
- All arrays are accessed consecutively in MVM (spatial locality)

- **Disadvantages:**

- x is not reused
- Insertion costly

Impact of Matrix Sparsity on Performance

- Addressing overhead (dense MVM vs. dense MVM in CSR):
 - ~ 2x slower (Mflop/s, example only)
- Irregular structure
 - ~ 5x slower (Mflop/s, example only) for “random” sparse matrices
- Fundamental difference between MVM and sparse MVM (SMVM):
 - Sparse MVM is input **dependent** (sparsity pattern of A)
 - Changing the order of computation (blocking) requires changing the data structure (CSR)

Bebop/Sparsity: SMVM Optimizations

- *Idea:* Register blocking
- *Reason:* Reuse x to reduce memory traffic
- *Execution:* Block SMVM $y = y + Ax$ into micro MVMs
 - Block size $r \times c$ becomes a parameter
 - Consequence: Change A from CSR to $r \times c$ block-CSR (BCSR)
- BCSR: Blackboard

BCSR (Blocks of Size $r \times c$)

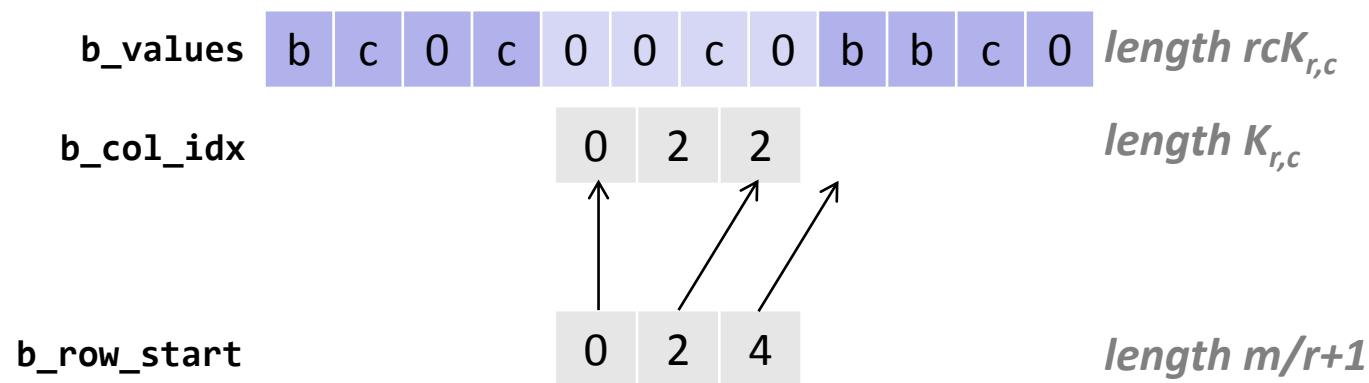
■ Assumptions:

- A is $m \times n$
- Block size $r \times c$
- $K_{r,c}$ nonzero blocks

A as matrix ($r = c = 2$)

b	c		c
	a		
		b	b
	c		

A in BCSR ($r = c = 2$):



■ Storage: $\Theta(rck_{r,c})$, $rck_{r,c} \geq K$

Sparse MVM Using 2 x 2 BCSR

```
void smvm_2x2(int bm, const int *b_row_start, const int *b_col_idx,
               const double *b_value, const double *x, double *y)
{
    int i, j;
    double d0, d1, c0, c1;

    /* loop over block rows */
    for (i = 0; i < bm; i++, y += 2) {
        d0 = y[i];      /* scalar replacement */
        d1 = y[i+1];

        /* dense micro MVM */
        for (j = b_row_start[i]; j < b_row_start[i+1]; j++, b_col_idx++, b_value += 2*2) {
            c0 = x[b_col_idx[j]+0]; /* scalar replacement */
            c1 = x[b_col_idx[j]+1];
            d0 += b_value[0] * c0;
            d1 += b_value[2] * c0;
            d0 += b_value[1] * c1;
            d1 += b_value[3] * c1;
        }
        y[i]    = d0;
        y[i+1] = d1;
    }
}
```

BCSR

■ Advantages:

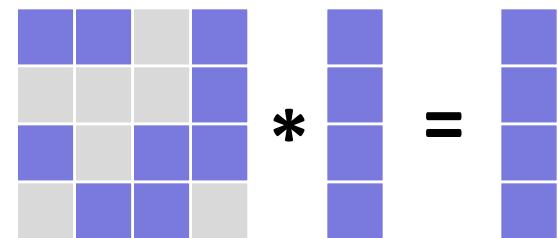
- Reuse of x and y (same as for dense MVM)
- Reduces storage for indexes

■ Disadvantages:

- Storage for values of A increased (zeros added)
- Computational overhead (also due to zeros)

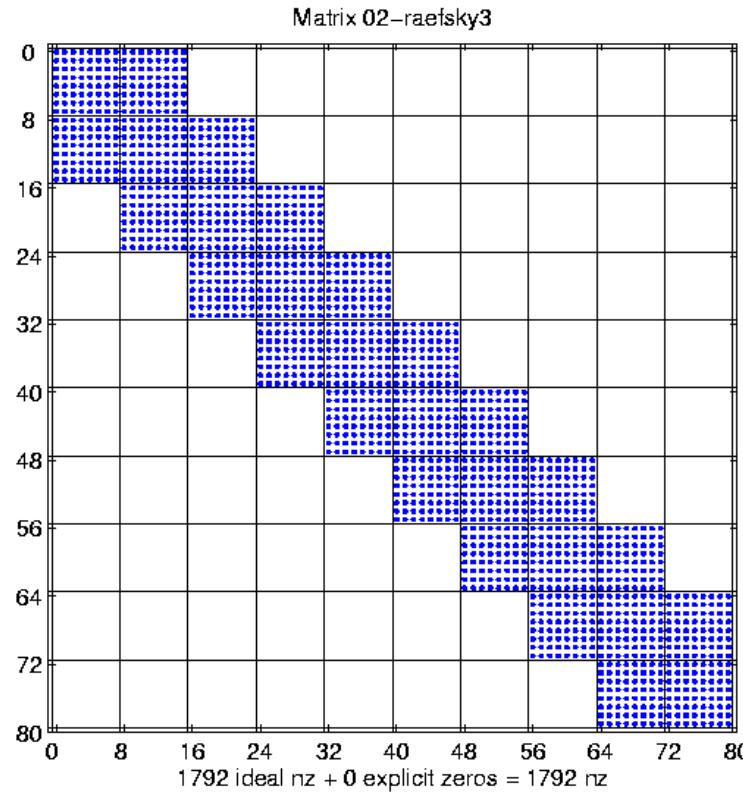
■ Main factors (since memory bound):

- *Plus:* increased reuse on x + reduced index storage
= reduced memory traffic
- *Minus:* more zeros = increased memory traffic


$$\begin{matrix} & \begin{matrix} \text{blue} & \text{blue} & \text{gray} & \text{blue} \\ \text{gray} & \text{blue} & \text{gray} & \text{blue} \\ \text{blue} & \text{gray} & \text{blue} & \text{blue} \\ \text{gray} & \text{blue} & \text{blue} & \text{gray} \end{matrix} & * & \begin{matrix} \text{blue} \\ \text{blue} \\ \text{gray} \\ \text{blue} \end{matrix} & = & \begin{matrix} \text{blue} \\ \text{blue} \\ \text{blue} \\ \text{blue} \end{matrix} \end{matrix}$$

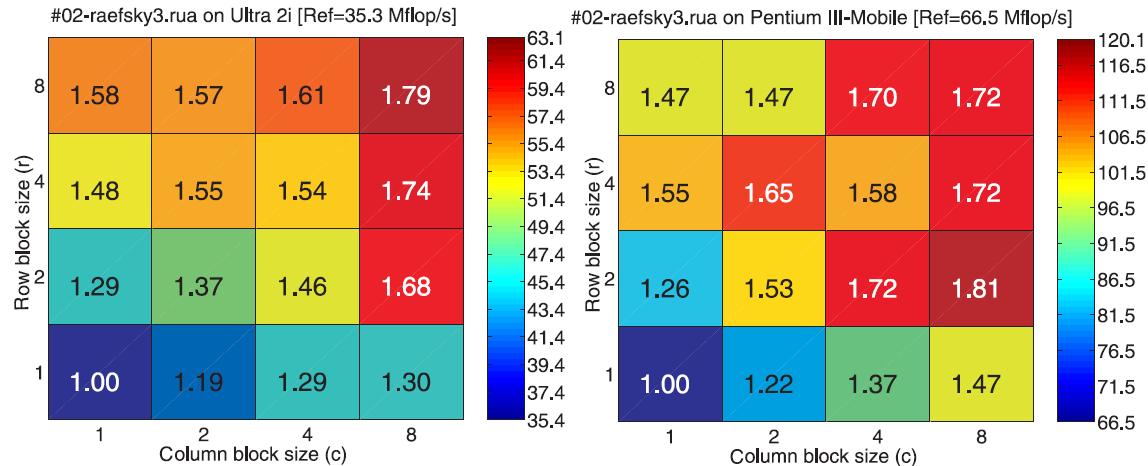
Which Block Size ($r \times c$) is Optimal?

- *Example:* about $20,000 \times 20,000$ matrix with perfect 8×8 block structure, 0.33% non-zero entries
- *In this case:* No overhead when blocked $r \times c$, with r,c divides 8

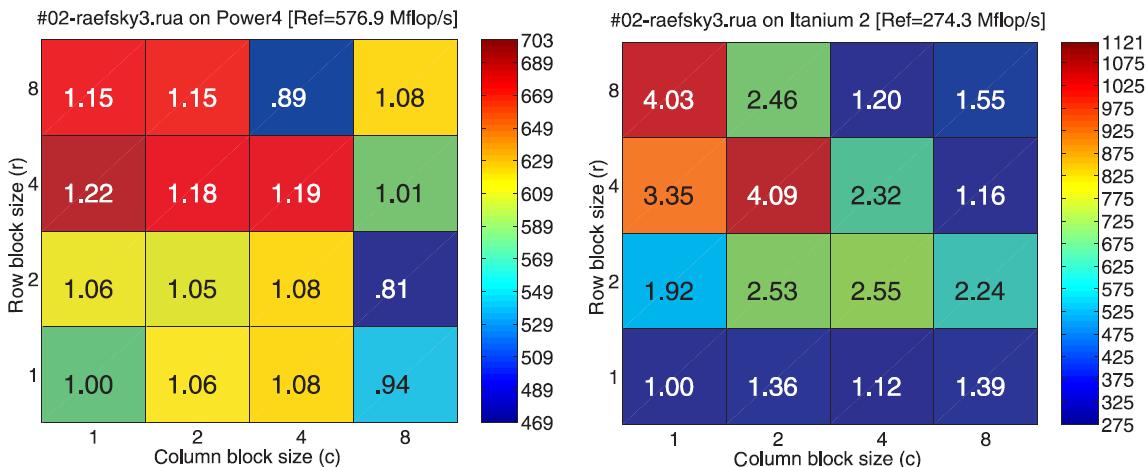


source: R. Vuduc, LLNL

Speed-up through $r \times c$ Blocking



- machine dependent
- hard to predict

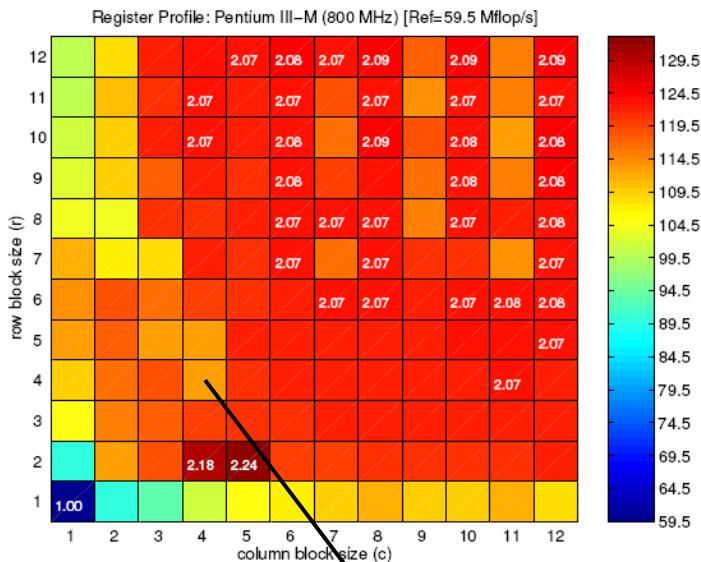


How to Find the Best Blocking for given A?

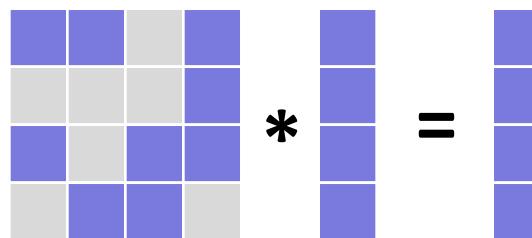
- Best block size is hard to predict (see previous slide)
- ***Solution 1:*** Searching over all $r \times c$ within a range, e.g., $1 \leq r, c \leq 12$
 - Conversion of A in CSR to BCSR roughly as expensive as 10 SMVMs
 - Total cost: 1440 SMVMs
 - Too expensive
- ***Solution 2: Model***
 - Estimate the gain through blocking
 - Estimate the loss through blocking
 - Pick best ratio

Model: Example

Gain by blocking (dense MVM)



Overhead (average) by blocking



$$16/9 = 1.77$$

1.4

$$1.4/1.77 = 0.79 \text{ (no gain)}$$

Model: Doing that for all r and c and picking best

Model

- **Goal:** find best $r \times c$ for $y = y + Ax$
- **Gain** through $r \times c$ blocking (estimation):

$$G_{r,c} = \frac{\text{dense MVM performance in } r \times c \text{ BCSR}}{\text{dense MVM performance in CSR}}$$

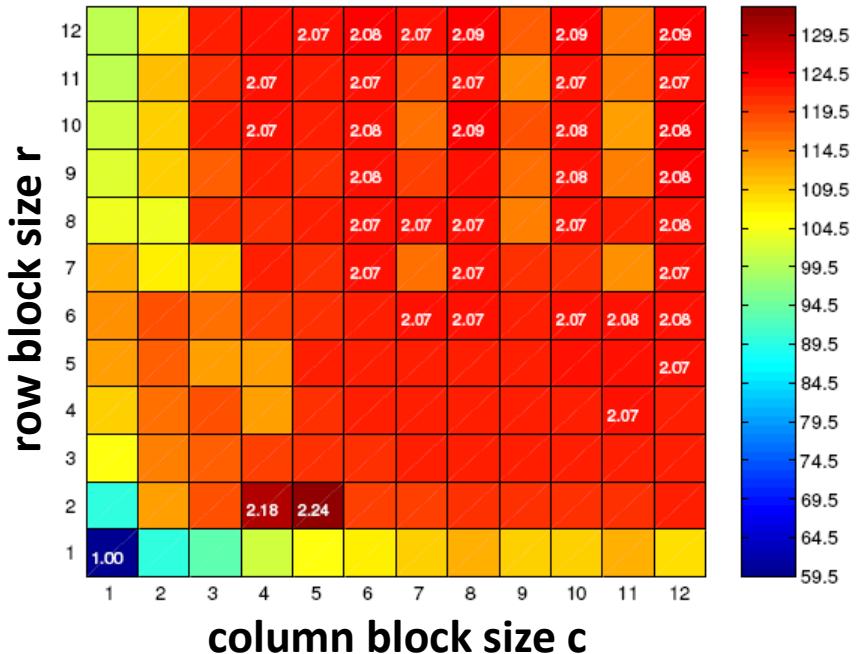
- dependent on machine, independent of sparse matrix
- **Overhead** through $r \times c$ blocking (estimation)
 - scan part of matrix A

$$O_{r,c} = \frac{\text{number of matrix values in } r \times c \text{ BCSR}}{\text{number of matrix values in CSR}}$$

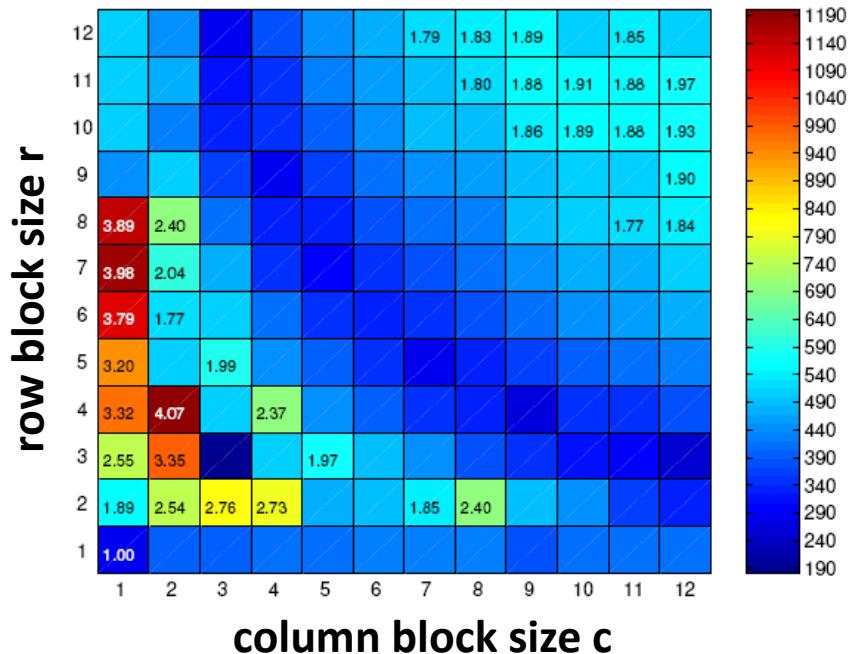
- independent of machine, dependent on sparse matrix
- **Expected gain:** $G_{r,c}/O_{r,c}$

Gain from Blocking (Dense Matrix in BCSR)

Pentium III



Itanium 2



- machine dependent
- hard to predict

Typical Result

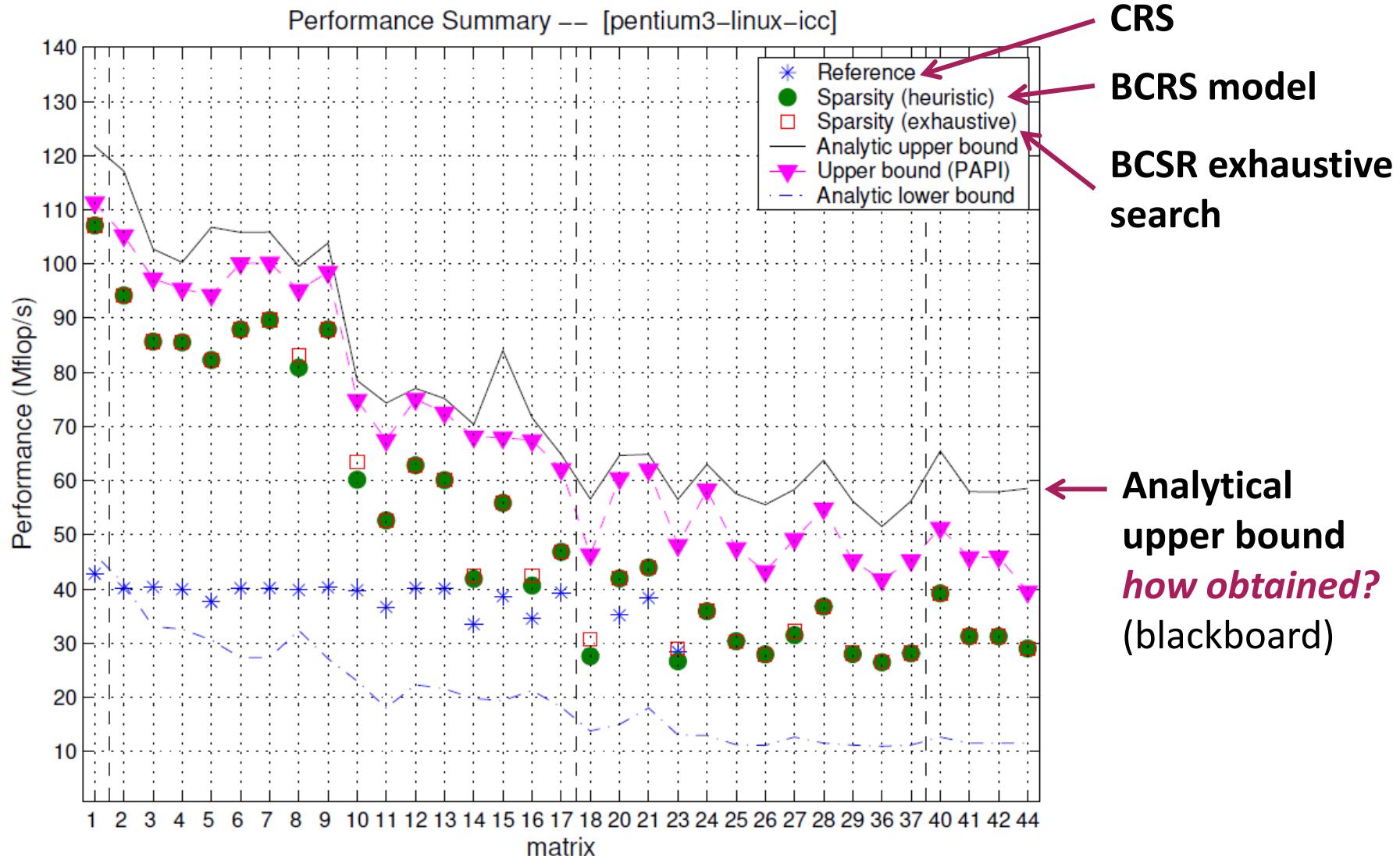


Figure: Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Comp. App., 18(1), pp. 135-158, 2004

How to Write Fast Numerical Code

Spring 2011
Lecture 16

Instructor: Markus Püschel

TA: Georg Ofenbeck

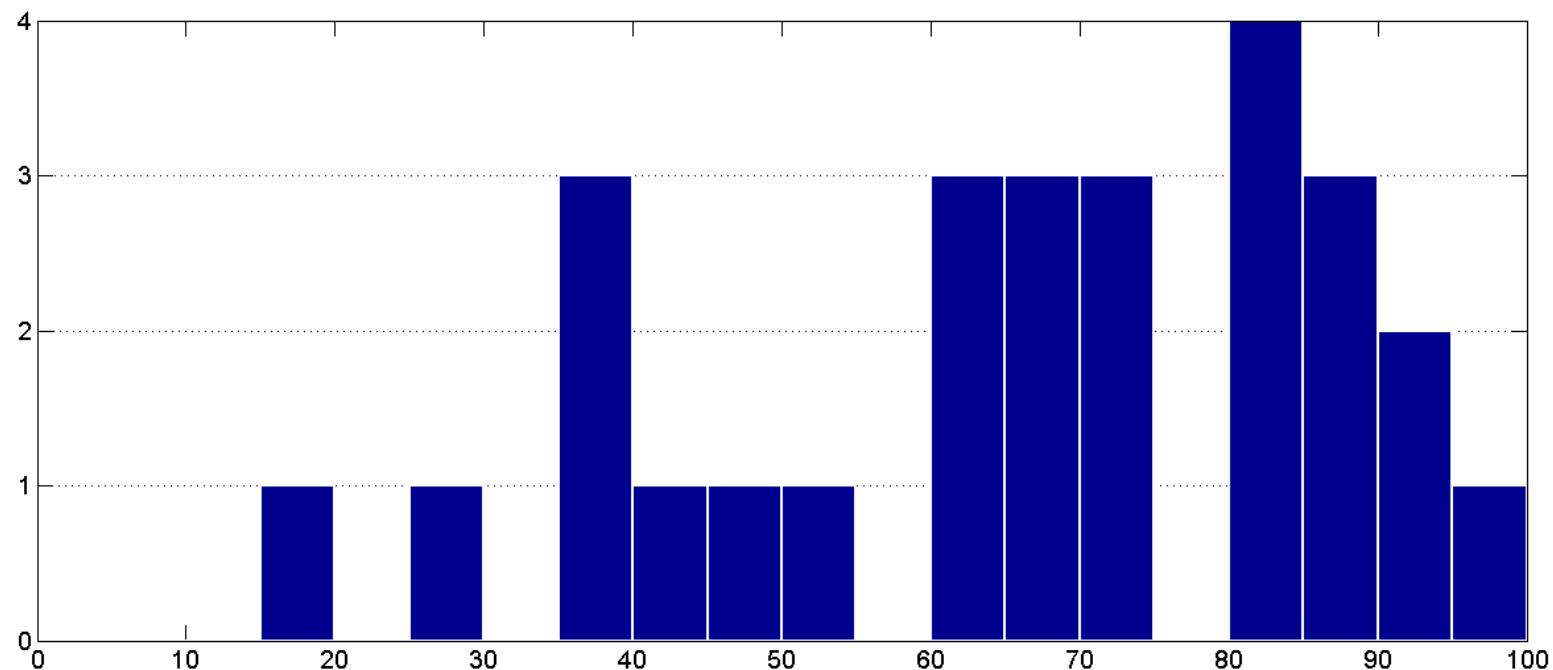


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Midterm

27 people

average: 65



Today

- SMVM continued

Sparse MVM (SMVM)

- $y = y + Ax$, A sparse but known

$$\begin{array}{c|c|c|c} & = & & \\ \text{y} & & \text{y} & \\ & + & & \\ & & \text{A} & \\ \text{y} & & & \bullet \\ & & & x \end{array}$$

CSR

■ Assumptions:

- A is $m \times n$
- K nonzero entries

A as matrix

b	c		c
	a		
		b	b
		c	

A in CSR:

values	b	c	c	a	b	b	c	<i>length K</i>
col_idx	0	1	3	1	2	3	2	<i>length K</i>
row_start	0	3	4	6	7			<i>length m+1</i>



BCSR (Blocks of Size $r \times c$)

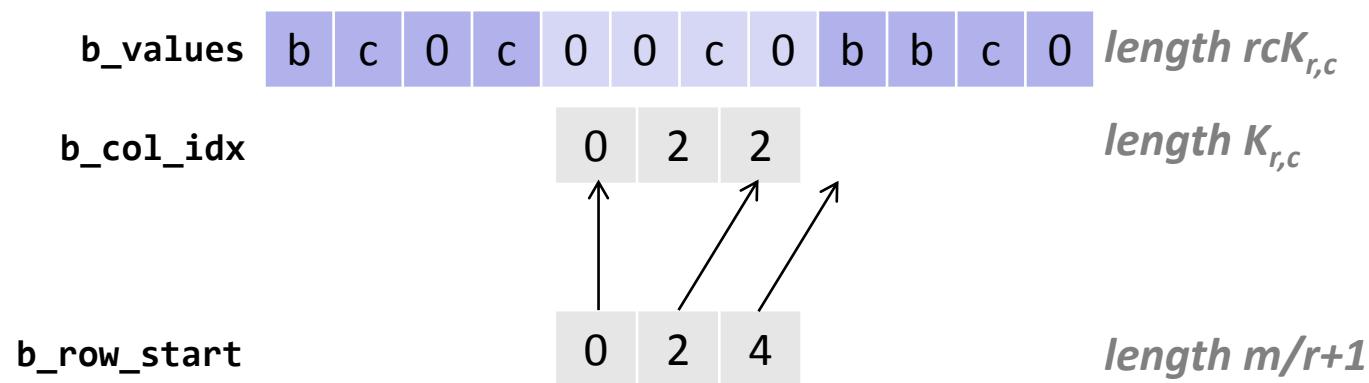
■ Assumptions:

- A is $m \times n$
- Block size $r \times c$
- $K_{r,c}$ nonzero blocks

A as matrix ($r = c = 2$)

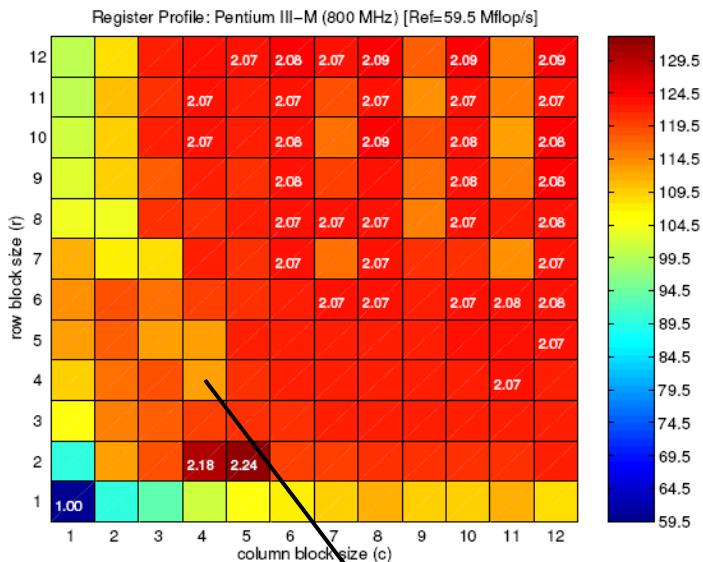
b	c		c
	a		
		b	b
	c		

A in BCSR ($r = c = 2$):



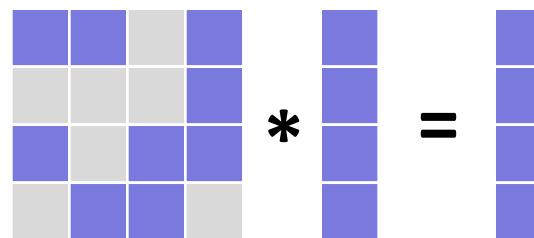
Model: Example

Gain by blocking (dense MVM)



1.4

Overhead (average) by blocking



$$16/9 = 1.77$$

$$1.4/1.77 = 0.79 \text{ (no gain)}$$

Model: Doing that for all r and c and picking best

Typical Result

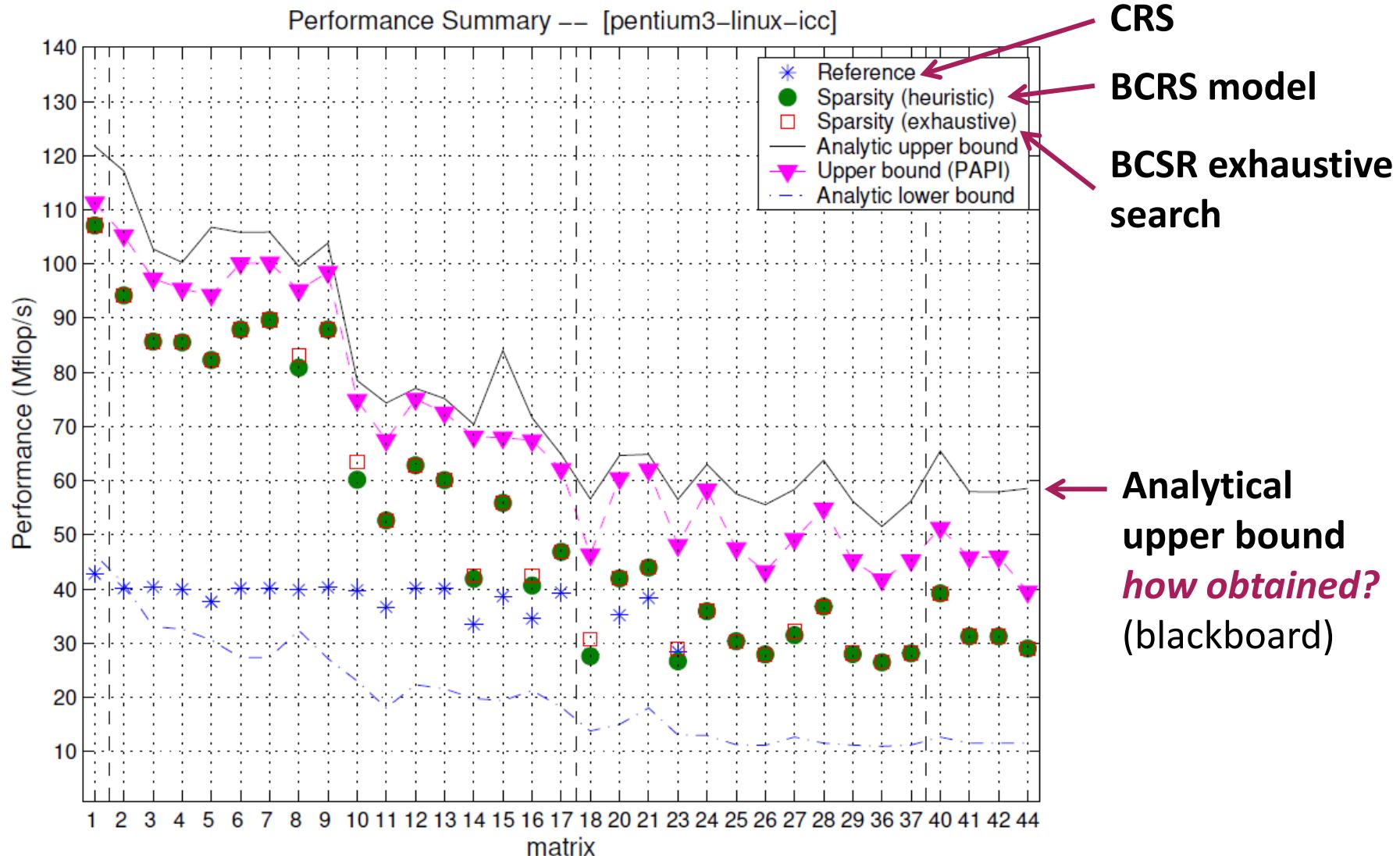


Figure: Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Comp. App., 18(1), pp. 135-158, 2004

Principles in Bebop/Sparsity Optimization

- *SMVM is memory bound*
- **Optimization for memory hierarchy = increasing locality**
 - Blocking for registers (micro-MMMs)
 - *Requires change of data structure for A*
 - Optimizations are *input dependent* (on sparse structure of A)
- **Fast basic blocks for small sizes (micro-MMM):**
 - Unrolled, scalar replacement (enables better compiler optimization)
- **Search for the fastest over a relevant set of algorithm/implementation alternatives (parameters r, c)**
 - *Use of performance model* (versus measuring runtime) to evaluate expected gain

Different from ATLAS

SMVM: Other Ideas

- Value compression
- Index compression
- Pattern-based compression
- Cache blocking
- Special scenario: Multiple inputs

Value Compression

- **Situation:** Matrix A contains many duplicate values
- **Idea:** Store only unique ones plus index information

b	c		c
	a		
		b	b
		c	

A in CSR:

	values	col_idx	row_start
	b c c a b b c	0 1 3 1 2 3 2	0 3 4 6 7

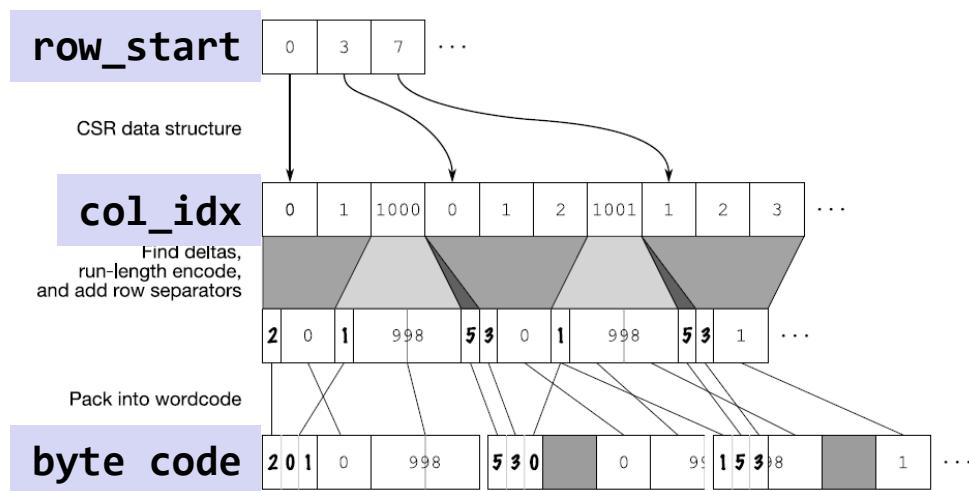
A in CSR-VI:

	values	col_idx	row_start
	a b c	1 2 2 0 1 1 2	0 3 4 6 7

Index Compression

- **Situation:** Matrix A contains sequences of nonzero entries
- **Idea:** Use special byte code to jointly compress `col_idx` and `row_start`

Coding

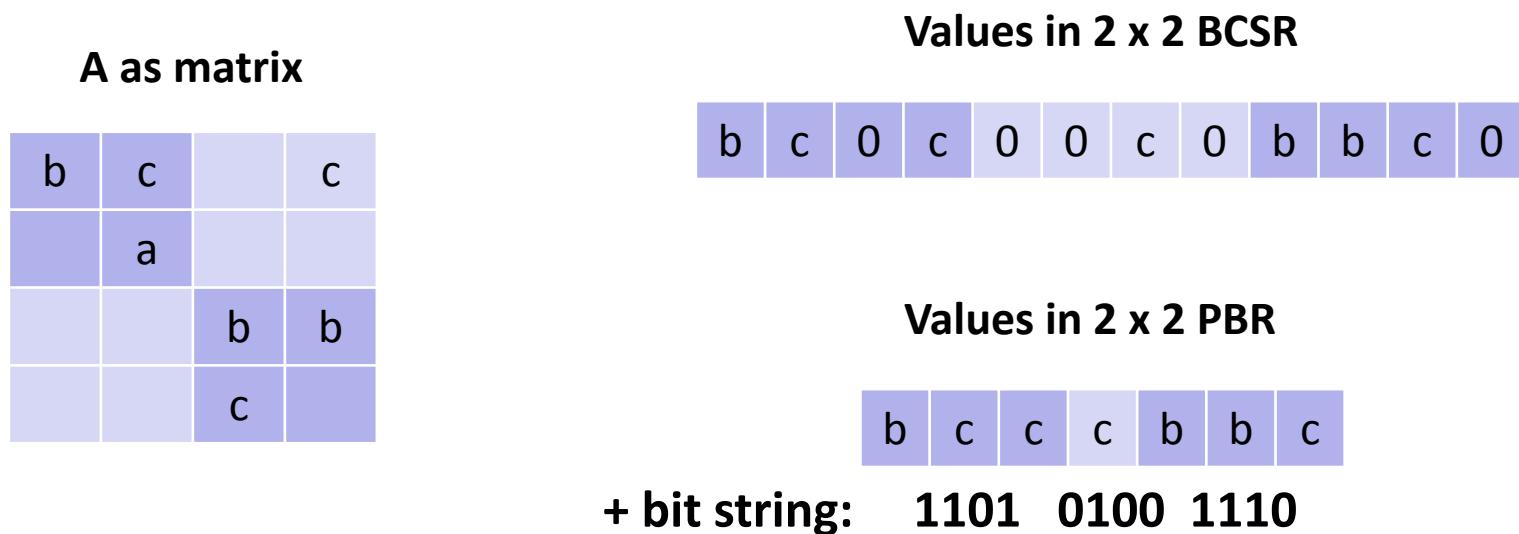


Decoding

```
0: acc = acc * 256 + arg;
1: col = col + acc * 256 + arg; acc = 0;
   emit_element(row, col); col = col + 1;
2: col = col + acc * 256 + arg; acc = 0;
   emit_element(row, col);
   emit_element(row, col + 1); col = col + 2;
3: col = col + acc * 256 + arg; acc = 0;
   emit_element(row, col);
   emit_element(row, col + 1);
   emit_element(row, col + 2); col = col + 3;
4: col = col + acc * 256 + arg; acc = 0;
   emit_element(row, col);
   emit_element(row, col + 1);
   emit_element(row, col + 2);
   emit_element(row, col + 3); col = col + 4;
5: row = row + 1; col = 0;
```

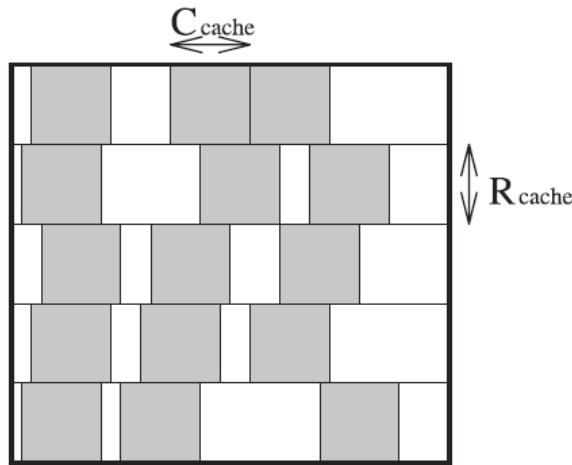
Pattern-Based Compression

- **Situation:** After blocking A, many blocks have the same nonzero pattern
- **Idea:** Use special BCSR format to avoid storing zeros; needs specialized micro-MVM kernel for each pattern



Cache Blocking

- Idea: divide sparse matrix into blocks of sparse matrices



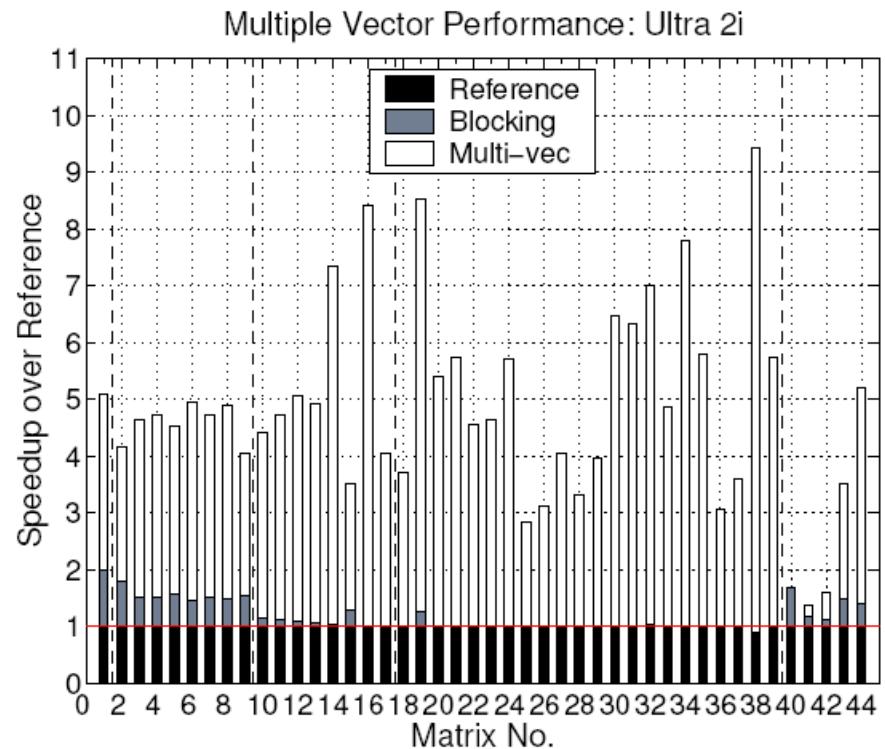
- Experiments:

- Requires very large matrices (x and y do not fit into cache)
- Speed-up up to 2.2x, only for few matrices, with 1×1 BCSR

Figure: Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Comp. App., 18(1), pp. 135-158, 2004

Special scenario: Multiple inputs

- Situation: Compute SMVM $y = y + Ax$ for several independent x
- Blackboard
- Experiments:
up to 9x speedup for 9 vectors



How to Write Fast Numerical Code

Spring 2011

Lecture 17

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SIMD Extensions and SSE

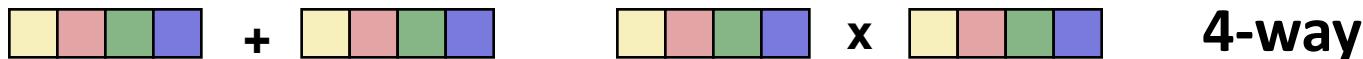
- Overview
- SSE family, floating point, and x87
- SSE intrinsics
- Compiler vectorization

- *This material was developed together with Franz Franchetti,
Carnegie Mellon*

SIMD (Single Instruction Multiple Data) Vector Extensions

■ What is it?

- Extension of the ISA. Data types and instructions for the parallel computation on short (length 2-8) vectors of integers or floats



- Names: MMX, SSE, SSE2, ...

■ Why do they exist?

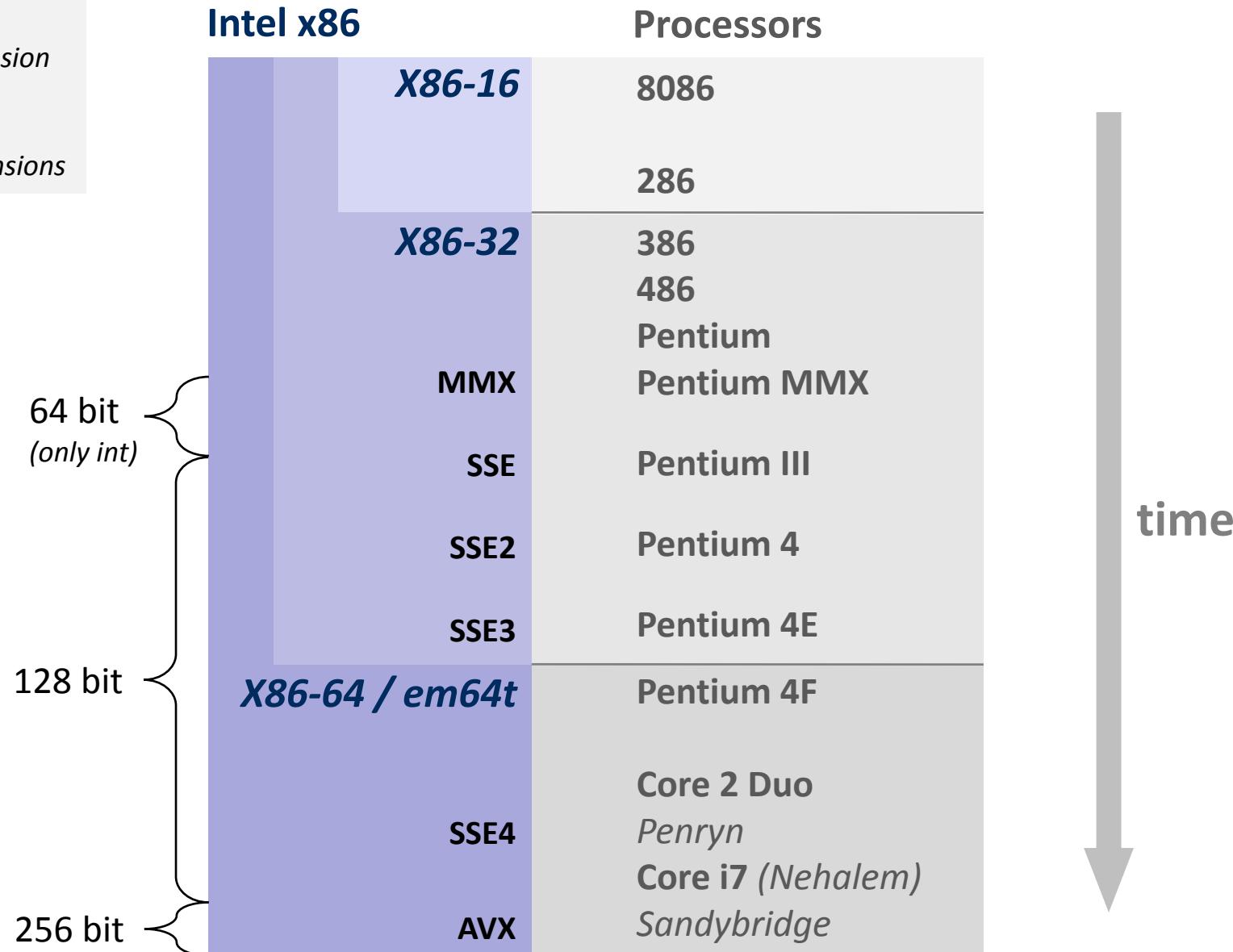
- **Useful:** Many applications have the necessary fine-grain parallelism
Then: speedup by a factor close to vector length
- **Doable:** Chip designers have enough transistors to play with

SSE:

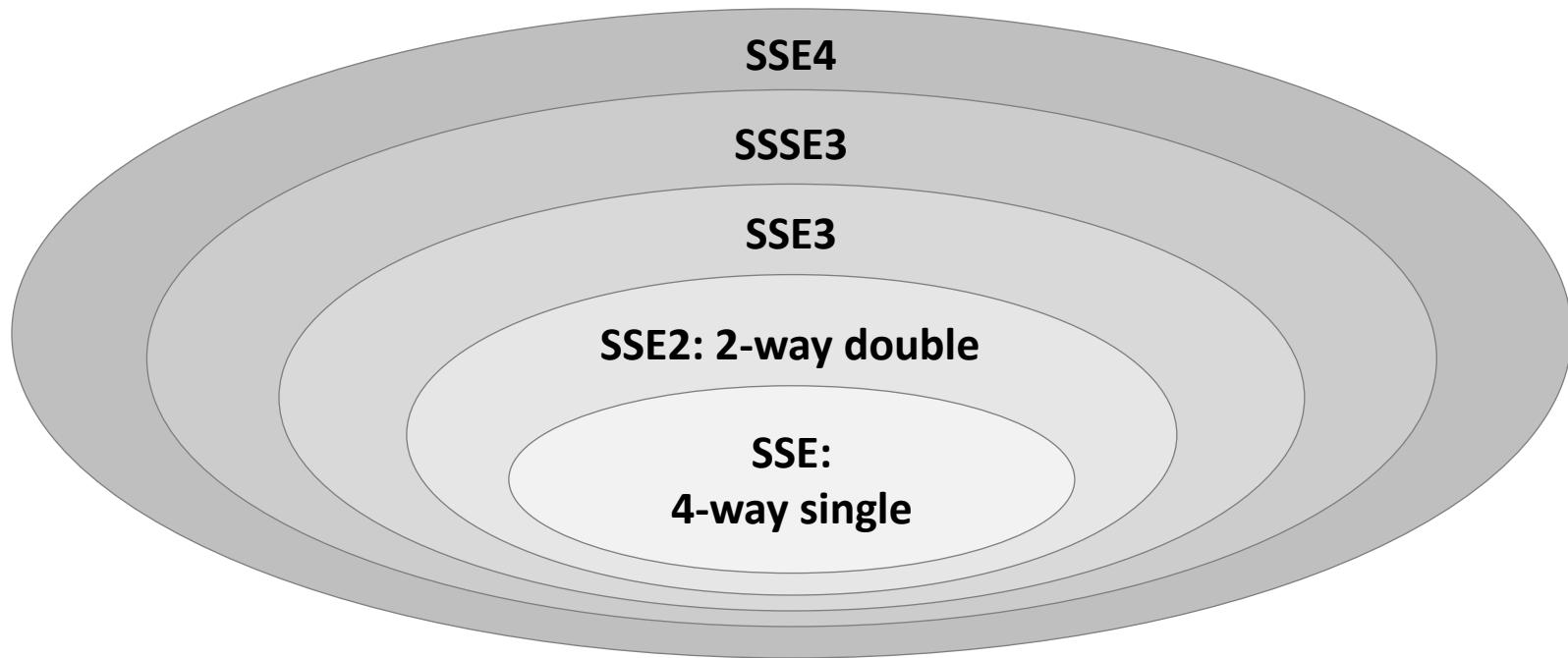
Streaming SIMD extension

AVX:

Advanced vector extensions



SSE Family: Floating Point



- Not drawn to scale
- From SSE3: Only additional instructions
- Every Core 2 has SSE3

Overview Floating-Point Vector ISAs

Vendor	Name	ν -way	Precision	Introduced with
Intel	SSE	4-way + 2-way	single	Pentium III
	SSE2		double	Pentium 4
	SSE3			Pentium 4 (Prescott)
	SSSE3			Core Duo
	SSE4			Core2 Extreme (Penryn)
	AVX		single double	Core i7 (Sandybridge)
Intel	IPF	2-way	single	Itanium
Intel	LRB	16-way	single	Larrabee
		8-way	double	
AMD	3DNow!	2-way	single	K6
	Enhanced 3DNow!			K7
	3DNow! Professional	+ 4-way	single	Athlon XP
	AMD64	+ 2-way	double	Opteron
Motorola	AltiVec	4-way	single	MPC 7400 G4
IBM	VMX	4-way	single	PowerPC 970 G5
	SPU	+ 2-way	double	Cell BE
IBM	Double FPU	2-way	double	PowerPC 440 FP2

Within an extension family, newer generations add features to older ones

Convergence: 3DNow! Professional = 3DNow! + SSE; VMX = AltiVec;

Core 2

- Has SSE3
- 16 SSE registers

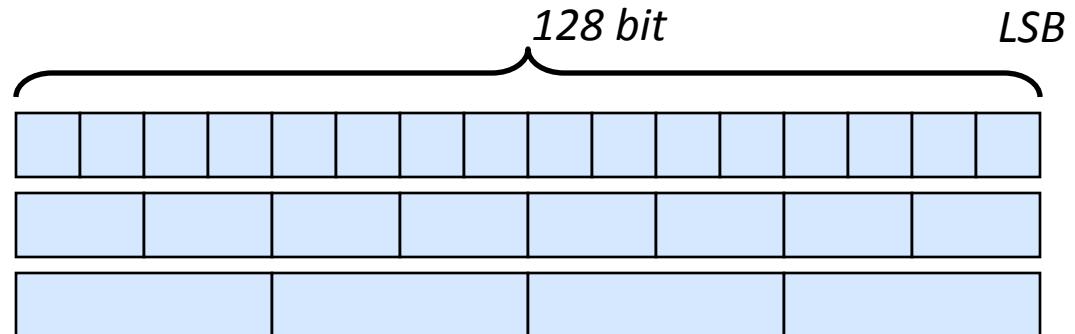


SSE3 Registers

- Different data types and associated instructions

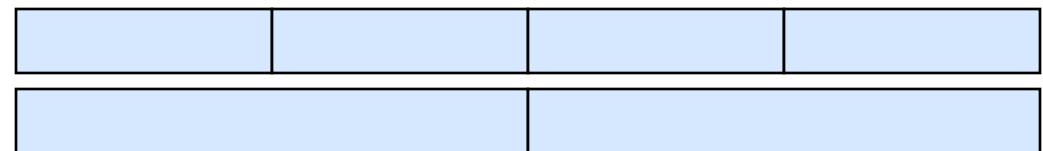
- Integer vectors:

- 16-way byte
- 8-way 2 bytes
- 4-way 4 bytes



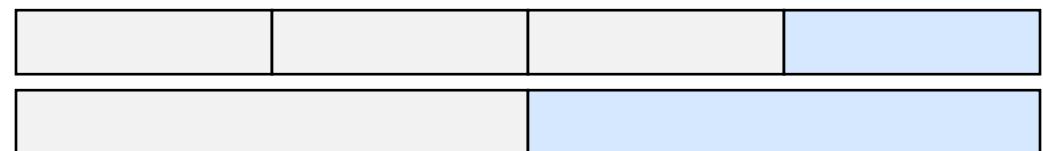
- Floating point vectors:

- 4-way single (since SSE)
- 2-way double (since SSE2)



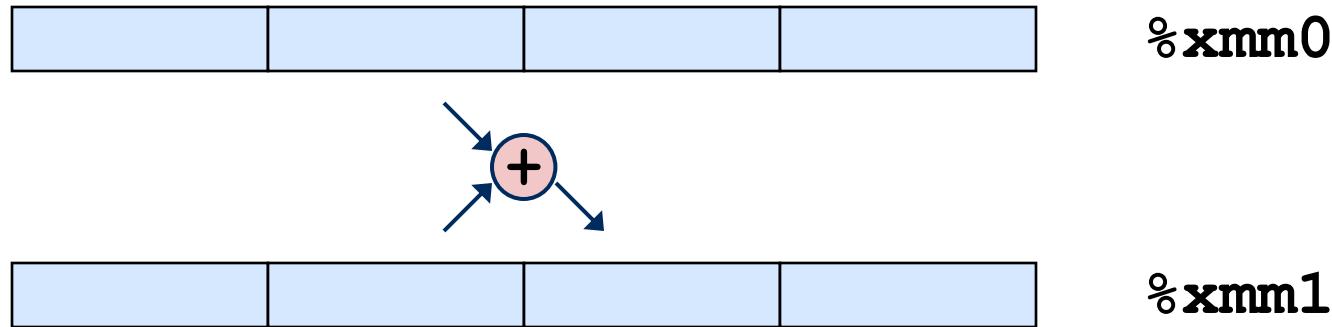
- Floating point scalars:

- single (since SSE)
- double (since SSE2)

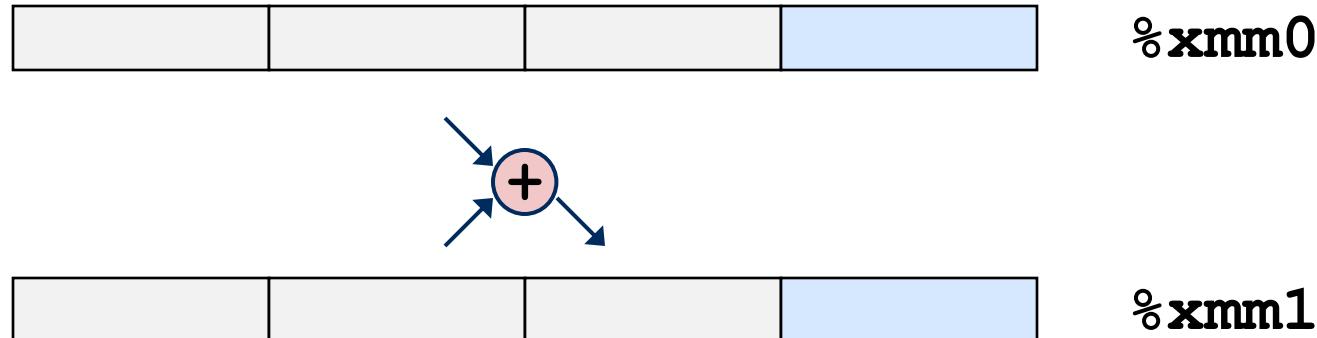


SSE3 Instructions: Examples

- Single precision *4-way vector add*: `addps %xmm0 %xmm1`



- Single precision *scalar add*: `addss %xmm0 %xmm1`



SSE3 Instruction Names

packed (vector)

↓

addps



single precision

addpd



double precision

single slot (scalar)

↓

addss



addsd



Compiler will use this for floating point

- *on x86-64*
- *with proper flags if SSE/SSE2 is available*

x86-64 FP Code Example

- Inner product of two vectors
 - Single precision arithmetic
 - Compiled: uses SSE instructions

```
float ipf (float x[],  
          float y[],  
          int n) {  
    int i;  
    float result = 0.0;  
  
    for (i = 0; i < n; i++)  
        result += x[i]*y[i];  
    return result;  
}
```

```
ipf:  
    xorps  %xmm1, %xmm1           # result = 0.0  
    xorl    %ecx, %ecx            # i = 0  
    jmp     .L8                  # goto middle  
.L10:  
    movslq  %ecx,%rax           # loop:  
    incl    %ecx                # icpy = i  
    movss  (%rsi,%rax,4), %xmm0  # i++  
    mulss  (%rdi,%rax,4), %xmm0  # t = y[icpy]  
    addss  %xmm0, %xmm1          # t *= x[icpy]  
    addss  %xmm0, %xmm1          # result += t  
.L8:  
    cmpl    %edx, %ecx           # middle:  
    jl     .L10                 # i:n  
    movaps  %xmm1, %xmm0          # if < goto loop  
    movaps  %xmm0, %xmm1          # return result  
    ret
```

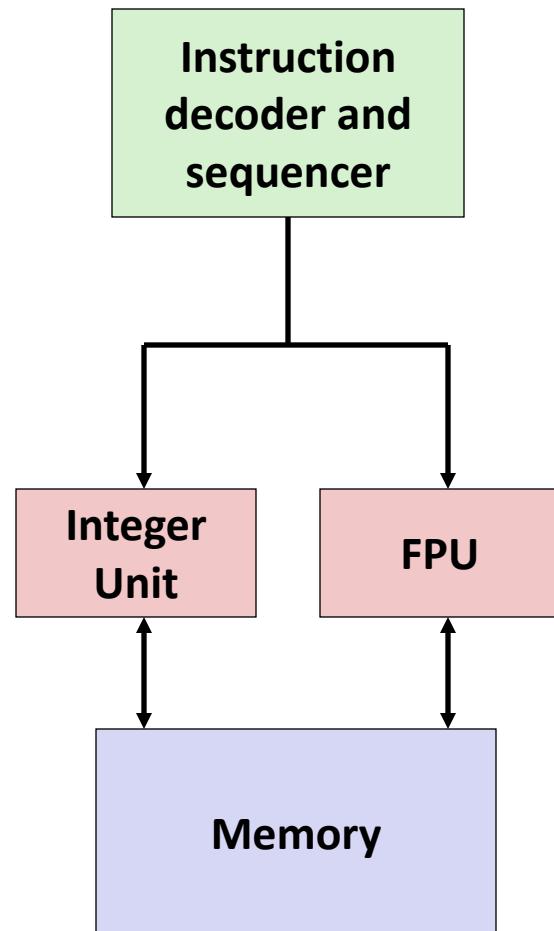
The Other Floating Point (x87)

■ History

- 8086: first computer to implement IEEE FP
(*separate 8087 FPU = floating point unit*)
- Logically stack based
- 486: merged FPU and Integer Unit onto one chip
- Default on x86-32 (since SSE is not guaranteed)
- Became obsolete with x86-64

■ Floating Point Formats

- single precision (C `float`): 32 bits
- double precision (C `double`): 64 bits
- extended precision (C `long double`): 80 bits



x87 FPU Instructions and Register Stack

- Sample instructions:

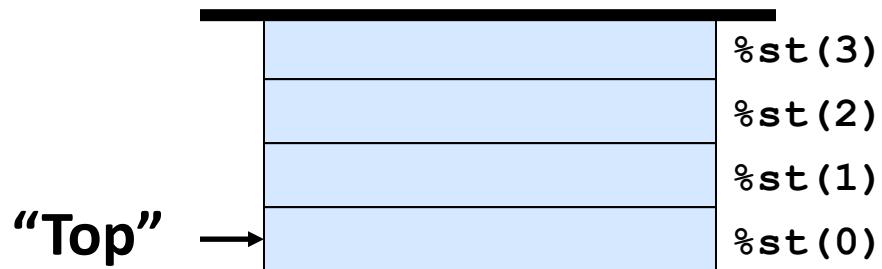
- flds (load single precision)
- fmuls (mult single precision)
- faddp (add and pop)

- 8 registers **%st(0) - %st(7)**

- Logically form stack

- Top: **%st(0)**

- Bottom disappears (drops out) after too many pushes



FP Code Example (x87)

■ Inner product of two vectors

- Single precision arithmetic

```
float ipf (float x[],  
          float y[],  
          int n) {  
  
    int i;  
    float result = 0.0;  
  
    for (i = 0; i < n; i++)  
        result += x[i]*y[i];  
    return result;  
}
```

```
pushl %ebp                      # setup  
movl %esp,%ebp  
pushl %ebx  
  
movl 8(%ebp),%ebx               # %ebx=&x  
movl 12(%ebp),%ecx              # %ecx=&y  
movl 16(%ebp),%edx              # %edx=n  
fldz                           # push +0.0  
xorl %eax,%eax                 # i=0  
cmpl %edx,%eax                 # if i>=n done  
jge .L3:  
  
.L5:  
flds (%ebx,%eax,4)           # push x[i]  
fmuls (%ecx,%eax,4)           # st(0)*=y[i]  
faddp                          # st(1)+=st(0); pop  
incl %eax                        # i++  
cmpl %edx,%eax                  # if i<n repeat  
jl .L5:  
  
.L3:  
movl -4(%ebp),%ebx              # finish  
movl %ebp, %esp  
popl %ebp  
ret                             # st(0) = result
```

From Core 2 Manual

Single-precision (SP) FP MUL Double-precision FP MUL	4, 1 5, 1	4, 1 5, 1	Issue port 0; Writeback port 0
FP MUL (X87) FP Shuffle DIV/SQRT	5, 2 1, 1	5, 2 1, 1	Issue port 0; Writeback port 0 FP shuffle does not handle QW shuffle.

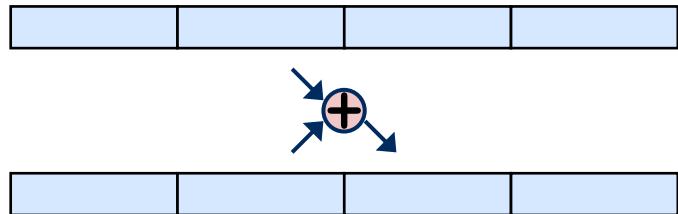
SSE based FP

x87 FP

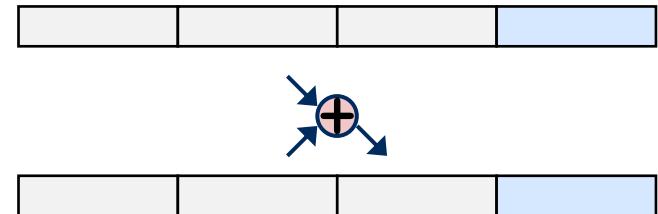
Summary

- **On Core 2 there are two different (unvectorized) floating points**
 - x87: obsolete, is default on x86-32
 - SSE based: uses only one slot, is default on x86-64
- **SIMD vector floating point instructions**
 - 4-way single precision: since SSE
 - 2-way double precision: since SSE2
 - Since on Core 2 add and mult are fully pipelined (1 per cycle): possible gain 4x and 2x, respectively

SSE: How to Take Advantage?



instead of



- Necessary: fine grain parallelism
- Options:
 - Use vectorized libraries (easy, not always available)
 - Write assembly
 - Use intrinsics (focus of this course)
 - Compiler vectorization (this course)
- We will focus on floating point and single precision (4-way)

SIMD Extensions and SSE

- Overview
- SSE family, floating point, and x87
- *SSE intrinsics*
- Compiler vectorization

References:

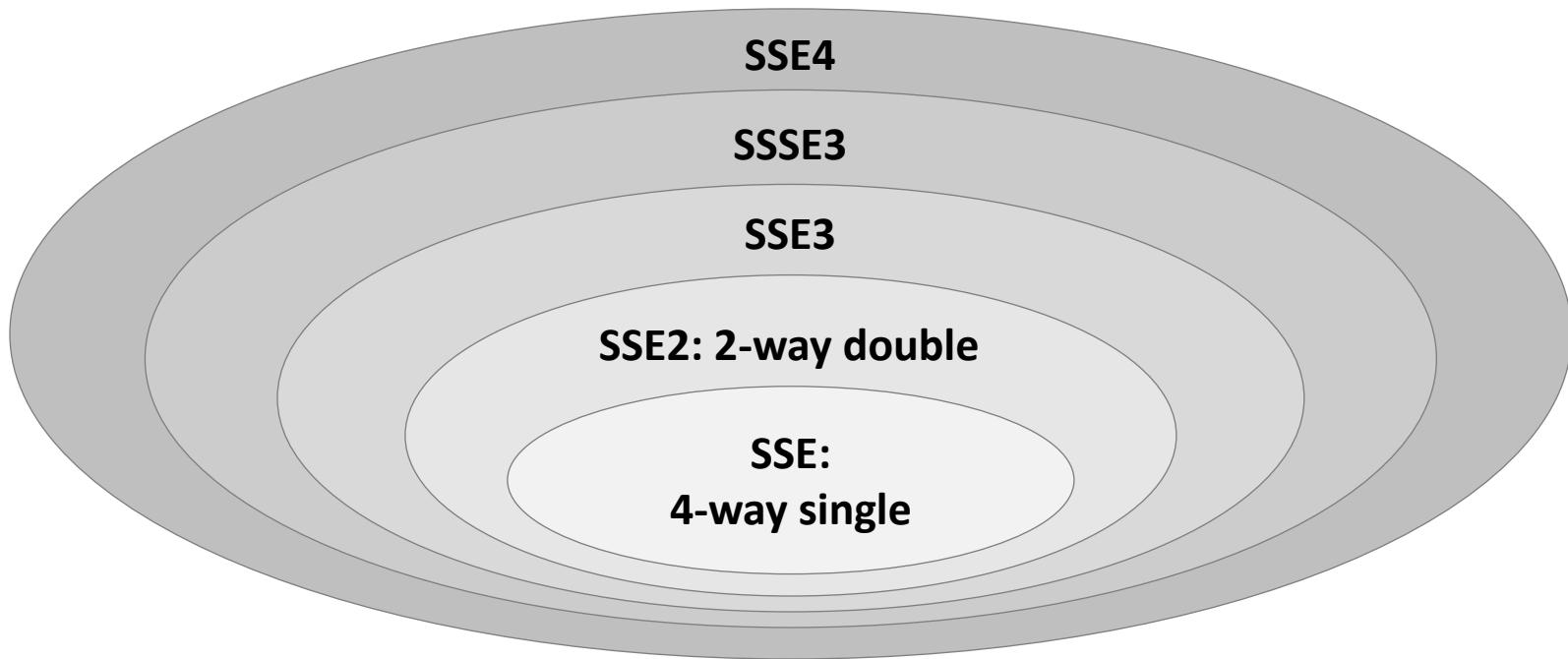
Intel icc manual (currently 12.0) → Intrinsics reference

<http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/lin/index.htm>

Visual Studio Manual (also: paste the intrinsic into Google)

<http://msdn.microsoft.com/de-de/library/26td21ds.aspx>

SSE Family: Floating Point



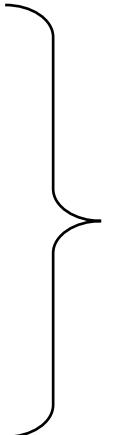
- Not drawn to scale
- From SSE3: Only additional instructions
- Every Core 2 has SSE3

SSE Family Intrinsics

- **Assembly coded C functions**
- **Expanded inline upon compilation: no overhead**
- **Like writing assembly inside C**
- **Floating point:**
 - Intrinsics for math functions: log, sin, ...
 - Intrinsics for SSE
- **Our introduction is based on icc**
 - Most intrinsics work with gcc and Visual Studio (VS)
 - Some language extensions are icc (or even VS) specific

Header files

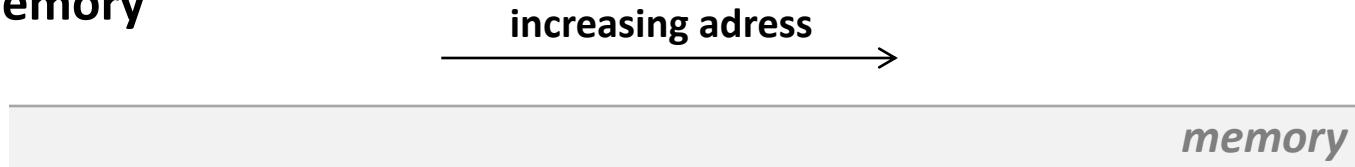
- SSE: `xmmINTRIN.h`
- SSE2: `emmINTRIN.h`
- SSE3: `pmmINTRIN.h`
- SSSE3: `tmmINTRIN.h`
- SSE4: `smmINTRIN.h` and `nmmINTRIN.h`



or `ia32INTRIN.h`

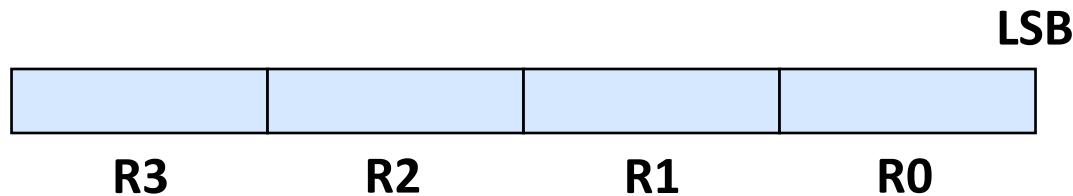
Visual Conventions We Will Use

■ Memory

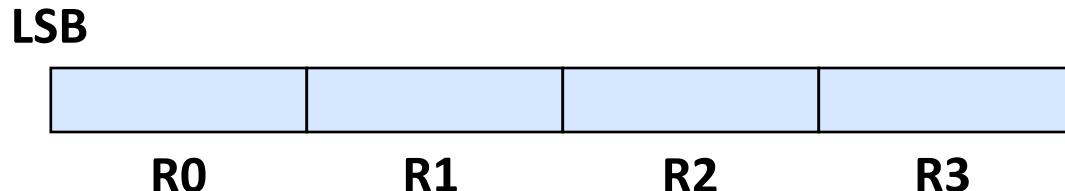


■ Registers

- Before (and common)



- Now we will use



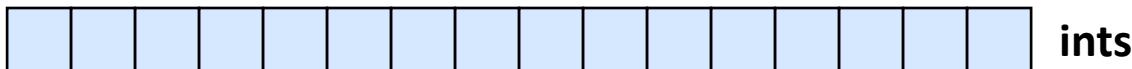
SSE Intrinsics (Focus Floating Point)

■ Data types

```
_m128 f; // = {float f0, f1, f2, f3}
```

```
_m128d d; // = {double d0, d1}
```

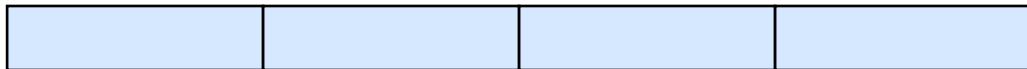
```
_m128i i; // 16 8-bit, 8 16-bit, 4 32-bit, or 2 64-bit ints
```



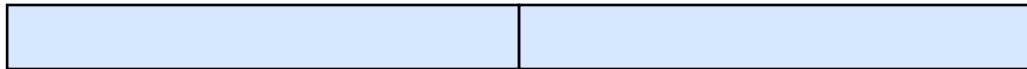
ints



ints



ints or floats



ints or doubles

SSE Intrinsics (Focus Floating Point)

■ Instructions

- Naming convention: `_mm_<intrin_op>_<suffix>`
- Example:

```
// a is 16-byte aligned
float a[4] = {1.0, 2.0, 3.0, 4.0};
__m128 t = _mm_load_ps(a);
```

p: packed
s: single

LSB

1.0	2.0	3.0	4.0
-----	-----	-----	-----

- Same result as

```
__m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0)
```

SSE Intrinsics

- Native instructions (one-to-one with assembly)

- `_mm_load_ps()`
 - `_mm_add_ps()`
 - `_mm_mul_ps()`

- ...

- Multi instructions (map to several assembly instructions)

- `_mm_set_ps()`
 - `_mm_set1_ps()`

- ...

- Macros and helpers

- `_MM_TRANSPOSE4_PS()`
 - `_MM_SHUFFLE()`

- ...

What Are the Main Issues?

- Alignment is important (**128 bit = 16 byte**)
- You need to code explicit loads and stores
(what does that remind you of?)
- Overhead through shuffles

SSE Intrinsics

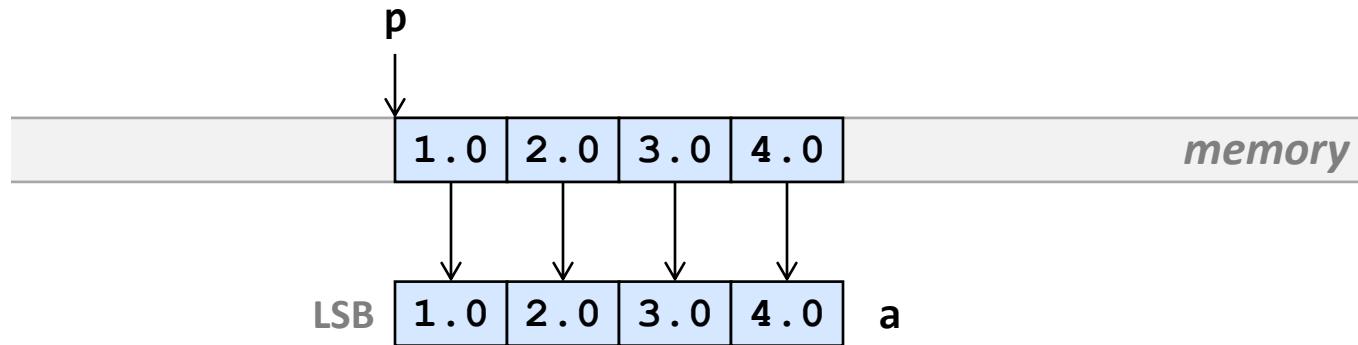
- Load and store
- Constants
- Arithmetic
- Comparison
- Conversion
- Shuffles

Loads and Stores

Intrinsic Name	Operation	Corresponding SSE Instructions
_mm_loadh_pi	Load high	MOVHPS reg, mem
_mm_loadl_pi	Load low	MOVLPS reg, mem
_mm_load_ss	Load the low value and clear the three high values	MOVSS
_mm_load1_ps	Load one value into all four words	MOVSS + Shuffling
_mm_load_ps	Load four values, address aligned	MOVAPS
_mm_loadu_ps	Load four values, address unaligned	MOVUPS
_mm_loadr_ps	Load four values in reverse	MOVAPS + Shuffling

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_set_ss	Set the low value and clear the three high values	Composite
_mm_set1_ps	Set all four words with the same value	Composite
_mm_set_ps	Set four values, address aligned	Composite
_mm_setr_ps	Set four values, in reverse order	Composite
_mm_setzero_ps	Clear all four values	Composite

Loads and Stores



```
a = _mm_load_ps(p); // p 16-byte aligned
```

```
a = _mm_loadu_ps(p); // p not aligned
```

avoid (expensive)

How to Align

- `__m128`, `__m128d`, `__m128i` are 16-byte aligned

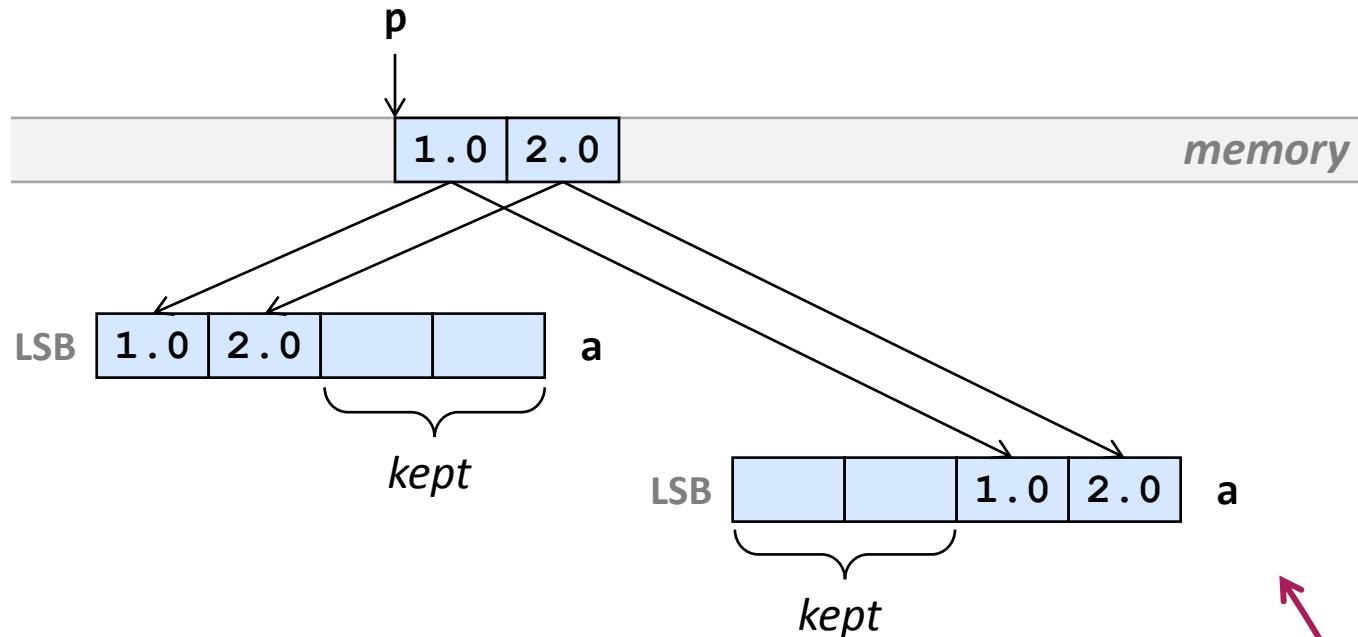
- Arrays:

```
__declspec(align(16)) float g[4];
```

- Dynamic allocation

- `_mm_malloc()` and `_mm_free()`
- Write your own malloc that returns 16-byte aligned addresses
- Some malloc's already guarantee 16-byte alignment

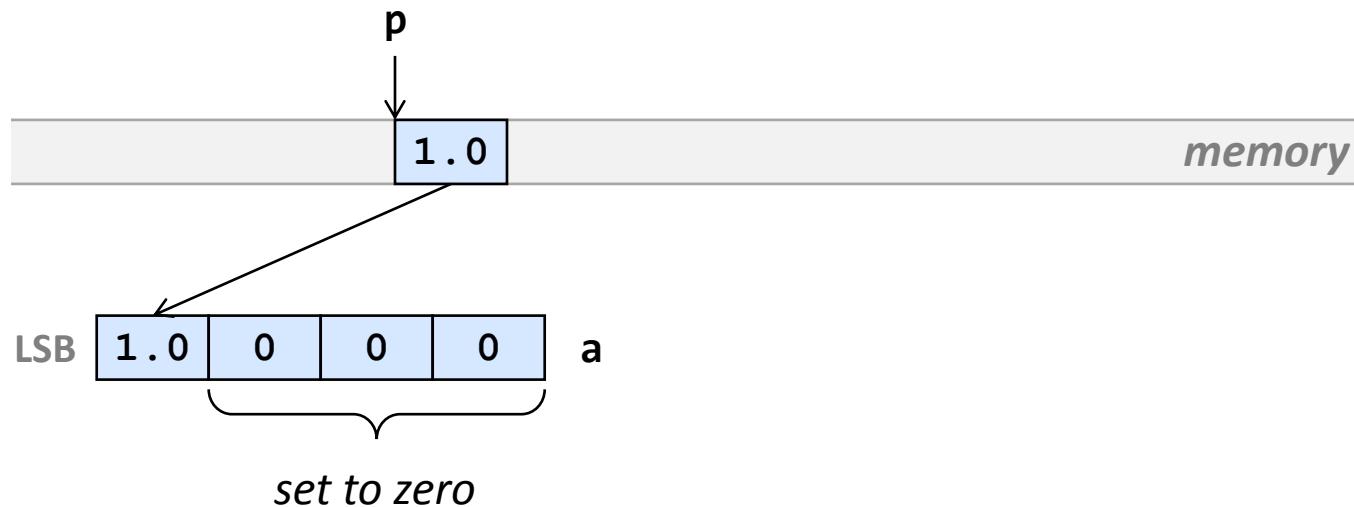
Loads and Stores



```
a = _mm_loadl_pi(a, p); // p 8-byte aligned
```

```
a = _mm_loadh_pi(a, p); // p 8-byte aligned
```

Loads and Stores



```
a = _mm_load_ss(p);
```

Stores Analogous to Loads

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_storeh_pi	Store high	MOVHPS mem, reg
_mm_storl_pi	Store low	MOVLPS mem, reg
_mm_store_ss	Store the low value	MOVSS
_mm_store1_ps	Store the low value across all four words, address aligned	Shuffling + MOVSS
_mm_store_ps	Store four values, address aligned	MOVAPS
_mm_storeu_ps	Store four values, address unaligned	MOVUPS
_mm_storer_ps	Store four values, in reverse order	MOVAPS + Shuffling

Constants

LSB

1.0	2.0	3.0	4.0
-----	-----	-----	-----

 a

```
a = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

LSB

1.0	1.0	1.0	1.0
-----	-----	-----	-----

 b

```
b = _mm_set1_ps(1.0);
```

LSB

1.0	0	0	0
-----	---	---	---

 c

```
c = _mm_set_ss(1.0);
```

LSB

0	0	0	0
---	---	---	---

 d

```
d = _mm_setzero_ps();
```

Arithmetic

SSE

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_add_ss	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	Computes Maximum	MAXPS

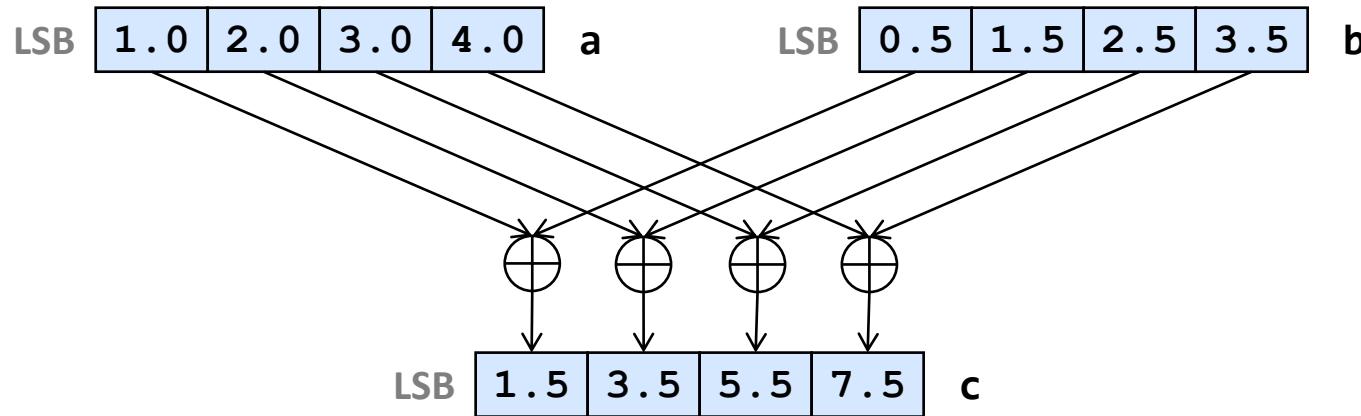
SSE3

Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_addsub_ps	Subtract and add	ADDSUBPS
_mm_hadd_ps	Add	HADDPS
_mm_hsub_ps	Subtracts	HSUBPS

SSE4

Intrinsic	Operation	Corresponding SSE4 Instruction
_mm_dp_ps	Single precision dot product	DPPS

Arithmetic



```
c = _mm_add_ps(a, b);
```

analogous:

```
c = _mm_sub_ps(a, b);
```

```
c = _mm_mul_ps(a, b);
```

Example

```
void addindex(float *x, int n) {
    for (int i = 0; i < n; i++)
        x[i] = x[i] + i;
}
```

```
#include <ia32intrin.h>

// n a multiple of 4, x is 16-byte aligned
void addindex_vec(float *x, int n) {
    __m128 index, x_vec;

    for (int i = 0; i < n/4; i++) {
        x_vec = _mm_load_ps(x+i*4);           // load 4 floats
        index = _mm_set_ps(i*4+3, i*4+2, i*4+1, i*4); // create vector with indexes
        x_vec = _mm_add_ps(x_vec, index);       // add the two
        _mm_store_ps(x+i*4, x_vec);           // store back
    }
}
```

Note how using intrinsics implicitly forces scalar replacement!

Example: Better Solution

```
void addindex(float *x, int n) {
    for (int i = 0; i < n; i++)
        x[i] = x[i] + i;
}
```

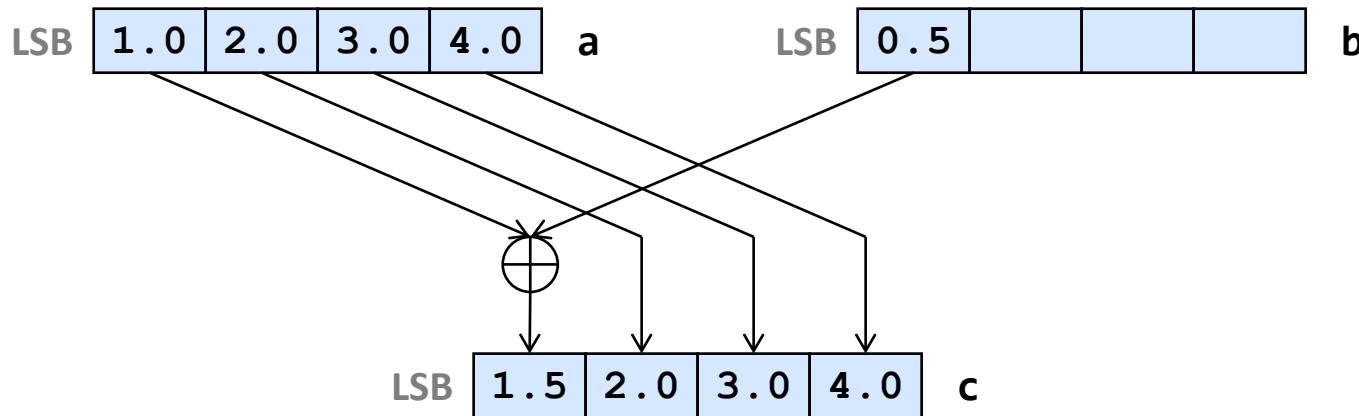
```
#include <ia32intrin.h>

// n a multiple of 4, x is 16-byte aligned
void addindex_vec(float *x, int n) {
    __m128 index, incr, x_vec;

    index = _mm_set_ps(0, 1, 2, 3);
    incr = _mm_set1_ps(4);
    for (int i = 0; i < n/4; i++) {
        x_vec = _mm_load_ps(x+i*4);           // load 4 floats
        x_vec = _mm_add_ps(x_vec, index);      // add index
        _mm_store_ps(x+i*4, x_vec);           // store back
        index = _mm_add_ps(index, incr);       // increment index
    }
}
```

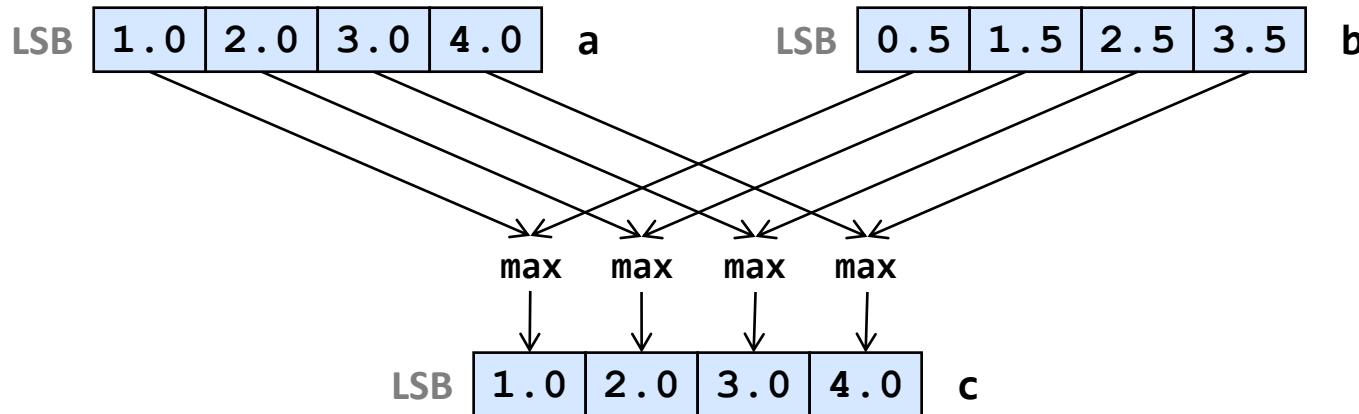
Note how using intrinsics implicitly forces scalar replacement!

Arithmetic



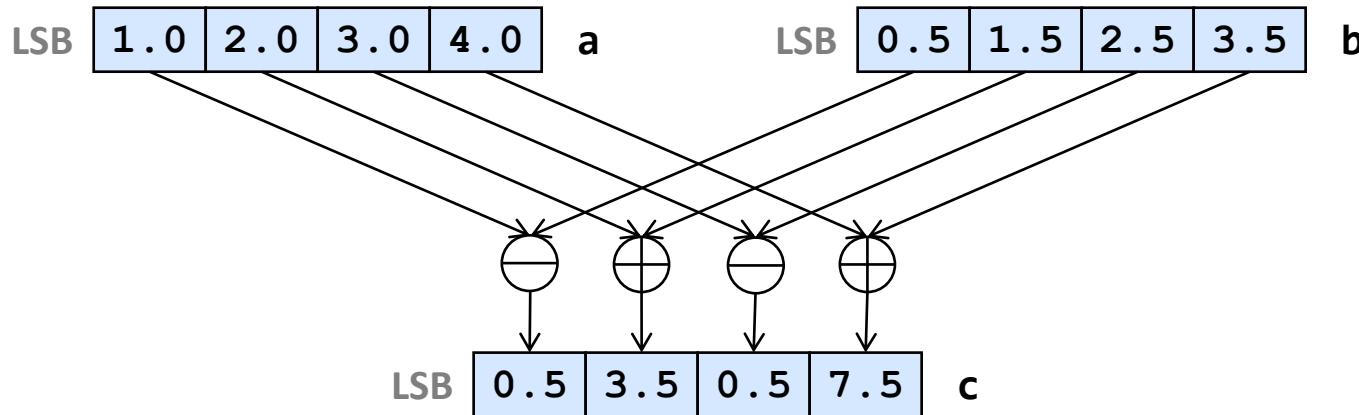
```
c = _mm_add_ss(a, b);
```

Arithmetic



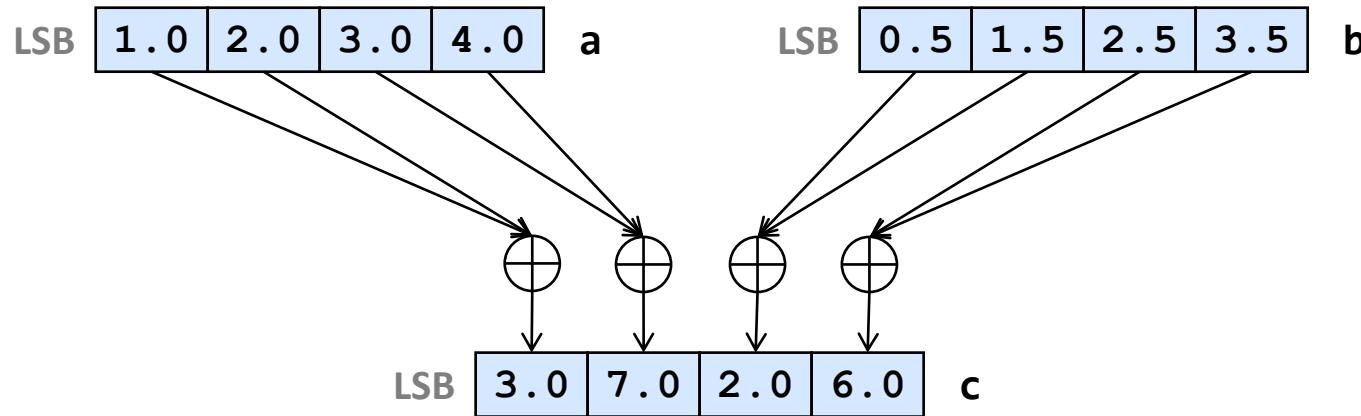
```
c = _mm_max_ps(a, b);
```

Arithmetic



```
c = _mm_addsub_ps(a, b);
```

Arithmetic



```
c = _mm_hadd_ps(a, b);
```

analogous:

```
c = _mm_hsub_ps(a, b);
```

Example

```
// n is even
void lp(float *x, float *y, int n) {
    for (int i = 0; i < n/2; i++)
        y[i] = (x[2*i] + x[2*i+1])/2;
}
```

```
#include <ia32intrin.h>

// n a multiple of 8, x, y are 16-byte aligned
void lp_vec(float *x, int n) {
    __m128 half, v1, v2, avg;

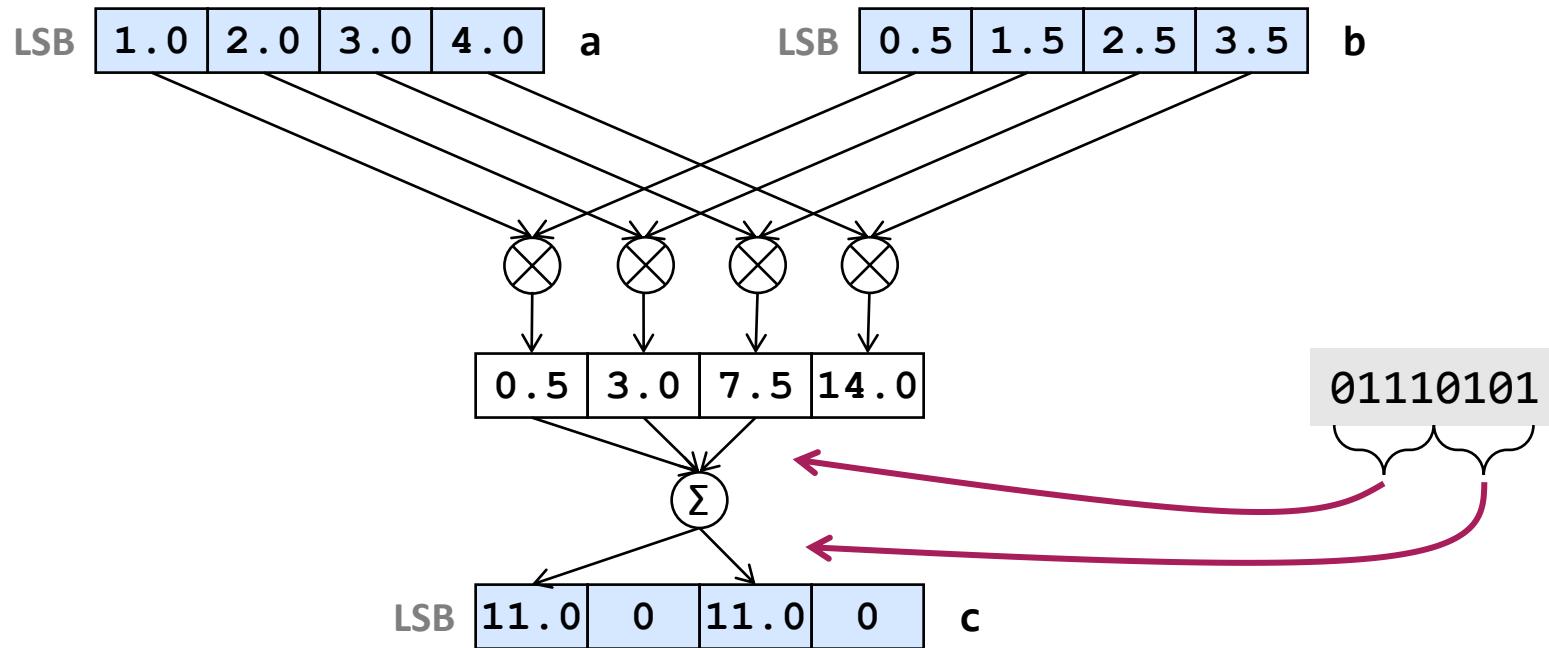
    half = _mm_set1_ps(0.5);           // set vector to all 0.5
    for (int i = 0; i < n/8; i++) {
        v1  = _mm_load_ps(x+i*8);     // load first 4 floats
        v2  = _mm_load_ps(x+4+i*8);   // load next 4 floats
        avg = _mm_hadd_ps(v1, v2);    // add pairs of floats
        avg = _mm_mul_ps(avg, half);   // multiply with 0.5
        _mm_store_ps(y+i*4, avg);     // save result
    }
}
```

Arithmetic

```
__m128 _mm_dp_ps(__m128 a, __m128 b, const int mask)
```

(SSE4) Computes the pointwise product of a and b and writes a selected sum of the resulting numbers into selected elements of c; the others are set to zero. The selections are encoded in the mask.

Example: mask = 117 = 01110101

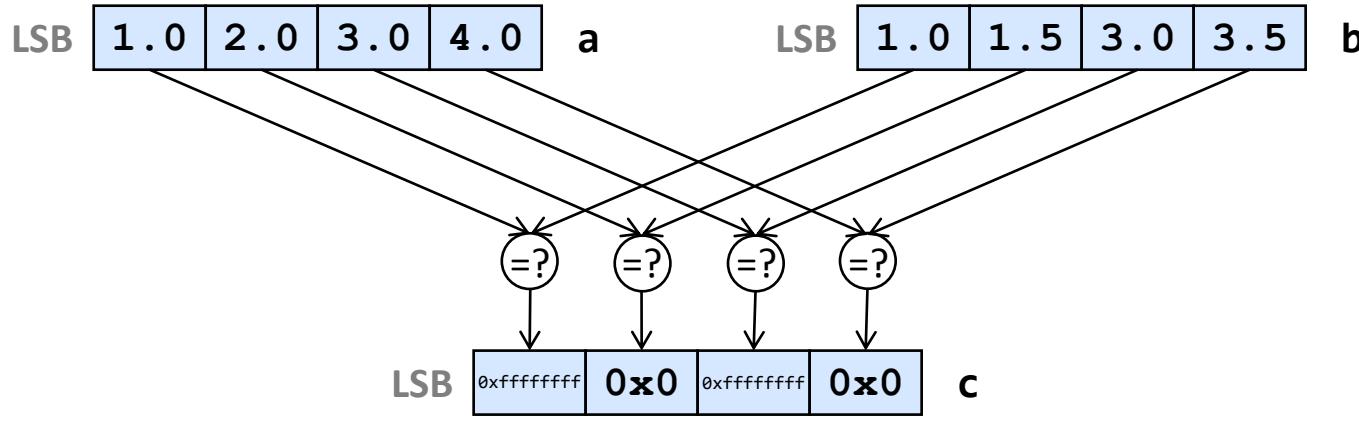


Comparisons

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_cmpeq_ss	Equal	CMPEQSS
_mm_cmpeq_ps	Equal	CMPEQPS
_mm_cmplt_ss	Less Than	CMPLTSS
_mm_cmplt_ps	Less Than	CMPLTPS
_mm_cmple_ss	Less Than or Equal	CMPLESS
_mm_cmple_ps	Less Than or Equal	CMPLEPS
_mm_cmpgt_ss	Greater Than	CMPLTSS
_mm_cmpgt_ps	Greater Than	CMPLTPS
_mm_cmpge_ss	Greater Than or Equal	CMPLESS
_mm_cmpge_ps	Greater Than or Equal	CMPLEPS
_mm_cmpneq_ss	Not Equal	CMPNEQSS
_mm_cmpneq_ps	Not Equal	CMPNEQPS
_mm_cmpnlt_ss	Not Less Than	CMPNLTSS
_mm_cmpnlt_ps	Not Less Than	CMPNLTPS
_mm_cmpnle_ss	Not Less Than or Equal	CMPNLESS
_mm_cmpnle_ps	Not Less Than or Equal	CMPNLEPS
_mm_cmpngt_ss	Not Greater Than	CMPNLTSS
_mm_cmpngt_ps	Not Greater Than	CMPNLTPS
_mm_cmpnge_ss	Not Greater Than or Equal	CMPNLESS
_mm_cmpnge_ps	Not Greater Than or Equal	CMPNLEPS

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_cmpord_ss	Ordered	CMPORDSS
_mm_cmpord_ps	Ordered	CMPORDPS
_mm_cmpunord_ss	Unordered	CMPUNORDSS
_mm_cmpunord_ps	Unordered	CMPUNORDPS
_mm_comieq_ss	Equal	COMISS
_mm_comilt_ss	Less Than	COMISS
_mm_comile_ss	Less Than or Equal	COMISS
_mm_comigt_ss	Greater Than	COMISS
_mm_comigc_ss	Greater Than or Equal	COMISS
_mm_comineq_ss	Not Equal	COMISS
_mm_ucomieq_ss	Equal	UCOMISS
_mm_ucomilt_ss	Less Than	UCOMISS
_mm_ucomile_ss	Less Than or Equal	UCOMISS
_mm_ucomigt_ss	Greater Than	UCOMISS
_mm_ucomigc_ss	Greater Than or Equal	UCOMISS
_mm_ucomineq_ss	Not Equal	UCOMISS

Comparisons



```
c = _mm_cmpeq_ps(a, b);
```

analogous:

```
c = _mm_cmple_ps(a, b);
```

```
c = _mm_cmplt_ps(a, b);
```

```
c = _mm_cmpge_ps(a, b);
```

Each field:

0xffffffff if true

0x0 if false

Return type __m128

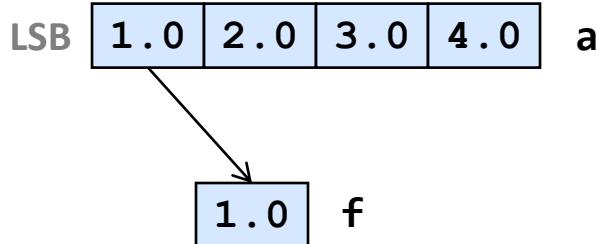
etc.

Conversion

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_cvtss_si32	Convert to 32-bit integer	CVTSS2SI
_mm_cvtss_si64*	Convert to 64-bit integer	CVTSS2SI
_mm_cvtps_pi32	Convert to two 32-bit integers	CVTPS2PI
_mm_cvttss_si32	Convert to 32-bit integer	CVTTSS2SI
_mm_cvttss_si64*	Convert to 64-bit integer	CVTTSS2SI
_mm_cvttps_pi32	Convert to two 32-bit integers	CVTPS2PI
_mm_cvtsi32_ss	Convert from 32-bit integer	CVTSI2SS
_mm_cvtsi64_ss*	Convert from 64-bit integer	CVTSI2SS
_mm_cvtpi32_ps	Convert from two 32-bit integers	CVTTPI2PS
_mm_cvtpi16_ps	Convert from four 16-bit integers	composite
_mm_cvtpu16_ps	Convert from four 16-bit integers	composite
_mm_cvtpi8_ps	Convert from four 8-bit integers	composite
_mm_cvtpu8_ps	Convert from four 8-bit integers	composite
_mm_cvtpi32x2_ps	Convert from four 32-bit integers	composite
_mm_cvtps_pi16	Convert to four 16-bit integers	composite
_mm_cvtps_pi8	Convert to four 8-bit integers	composite
_mm_cvtss_f32	Extract	composite

Conversion

```
float _mm_cvtss_f32(__m128 a)
```



```
float f;  
f = _mm_cvtss_f32(a);
```

Cast



```
_m128i _mm_castps_si128(_m128 a)
```

```
_m128 _mm_castsi128_ps(_m128i a)
```

Reinterprets the four single precision floating point values in `a` as four 32-bit integers, and vice versa.

No conversion is performed.

Makes integer shuffle instructions usable for floating point.

Shuffles

SSE

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_shuffle_ps	Shuffle	SHUFPS
_mm_unpackhi_ps	Unpack High	UNPCKHPS
_mm_unpacklo_ps	Unpack Low	UNPCKLPS
_mm_move_ss	Set low word, pass in three high values	MOVSS
_mm_movehl_ps	Move High to Low	MOVHLPS
_mm_movelh_ps	Move Low to High	MOVLHPS
_mm_movemask_ps	Create four-bit mask	MOVMSKPS

SSE3

Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_movehdup_ps	Duplicates	MOVSHDUP
_mm_movelddup_ps	Duplicates	MOVSLDDUP

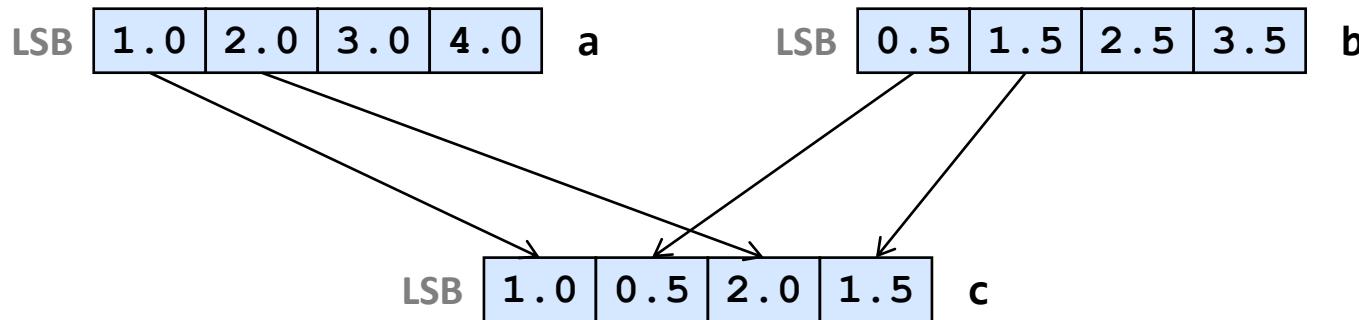
SSSE3

Intrinsic Name	Operation	Corresponding SSSE3 Instruction
_mm_shuffle_epi8	Shuffle	PSHUFB
_mm_alignr_epi8	Shift	PALIGNR

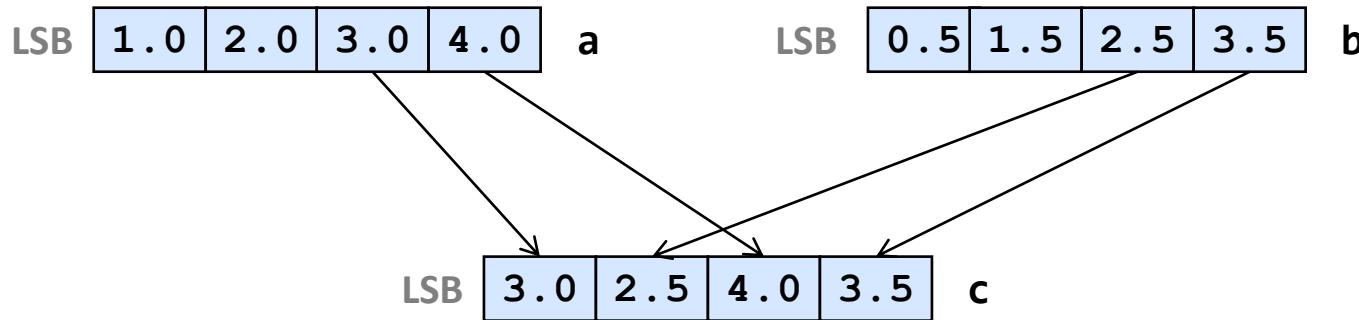
SSE4

Intrinsic Syntax	Operation	Corresponding SSE4 Instruction
__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask)	Selects float single precision data from 2 sources using constant mask	BLENDPS
__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)	Selects float single precision data from 2 sources using variable mask	BLENDVPS
__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)	Insert single precision float into packed single precision array element selected by index.	INSERTPS
int _mm_extract_ps(__m128 src, const int ndx)	Extract single precision float from packed single precision array selected by index.	EXTRACTPS

Shuffles



```
c = _mm_unpacklo_ps(a, b);
```



```
c = _mm_unpackhi_ps(a, b);
```

Shuffles

```
c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(l, k, j, i));
```

helper macro to create mask

LSB

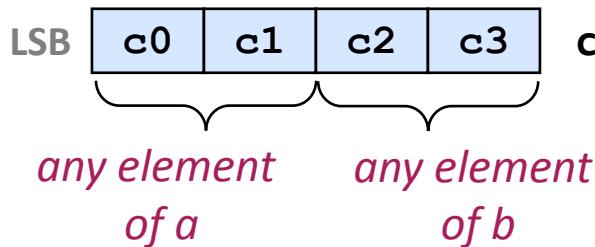
1.0	2.0	3.0	4.0
-----	-----	-----	-----

 a

LSB

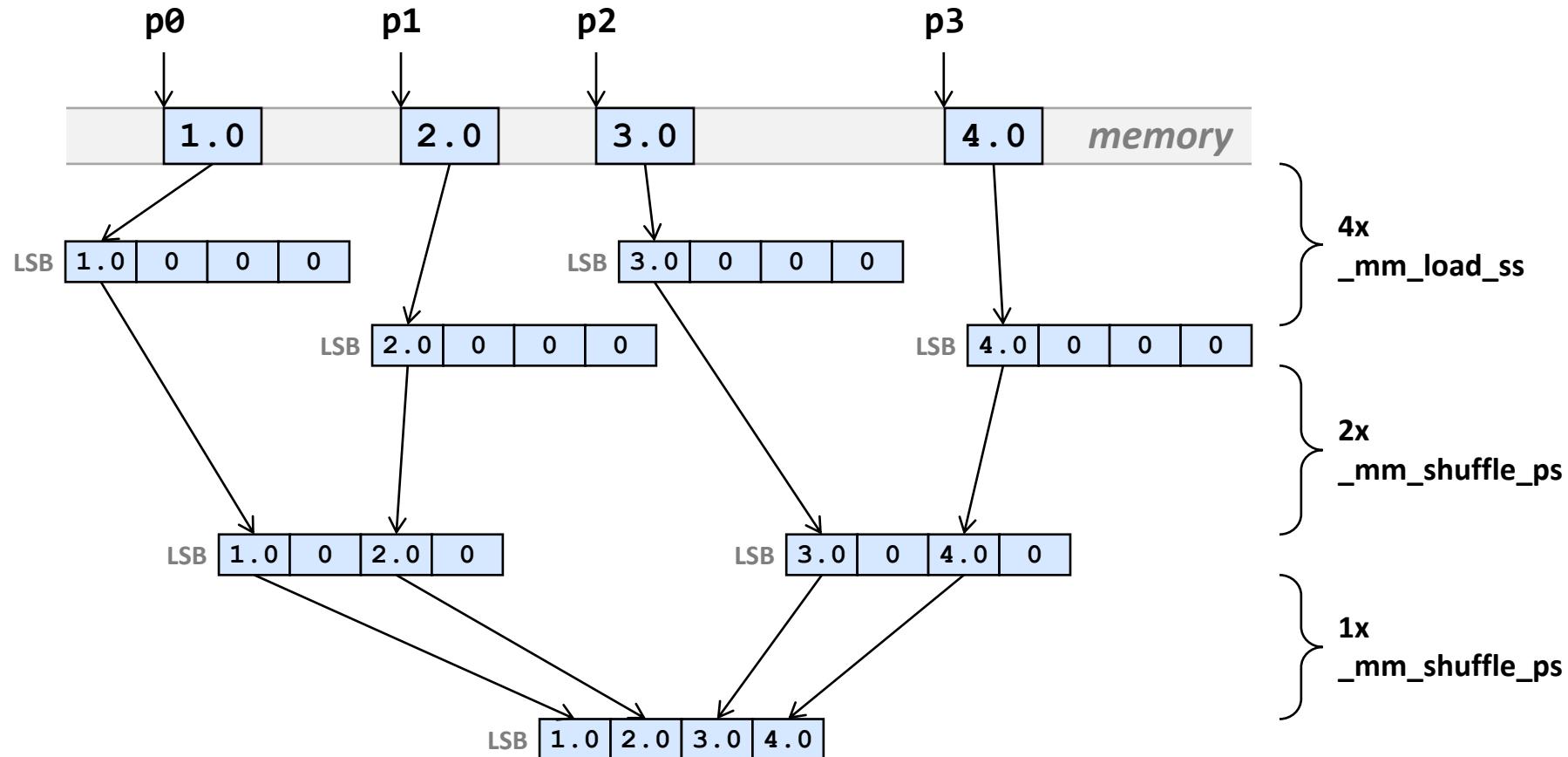
0.5	1.5	2.5	3.5
-----	-----	-----	-----

 b



$c_0 = ai$
 $c_1 = aj$
 $c_2 = bk$
 $c_3 = bl$
 $i, j, k, l \in \{0, 1, 2, 3\}$

Example: Loading 4 Real Numbers from Arbitrary Memory Locations



7 instructions, this is the right way (before SSE4)

Code For Previous Slide

```
#include <ia32intrin.h>

__m128 LoadArbitrary(float *p0, float *p1, float *p2, float *p3) {
    __m128 a, b, c, d, e, f;

    a = _mm_load_ss(p0);
    b = _mm_load_ss(p1);
    c = _mm_load_ss(p2);
    d = _mm_load_ss(p3);
    e = _mm_shuffle_ps(a, b, _MM_SHUFFLE(0,2,0,2));      //only zeros are important
    f = _mm_shuffle_ps(c, d, _MM_SHUFFLE(0,2,0,2));      //only zeros are important
    return _mm_shuffle_ps(e, f, _MM_SHUFFLE(3,1,3,1));
}
```

Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont'd)

- Whenever possible avoid the previous situation
- Restructure algorithm and use the aligned `_mm_load_ps()`
- Other possibility (should also yields 7 instructions, trusting the compiler)

```
__m128 vf;  
  
vf = _mm_set_ps(*p3, *p2, *p1, *p0);
```

- SSE4: `_mm_insert_epi32` together with `_mm_castsi128_ps`
 - Not clear whether better

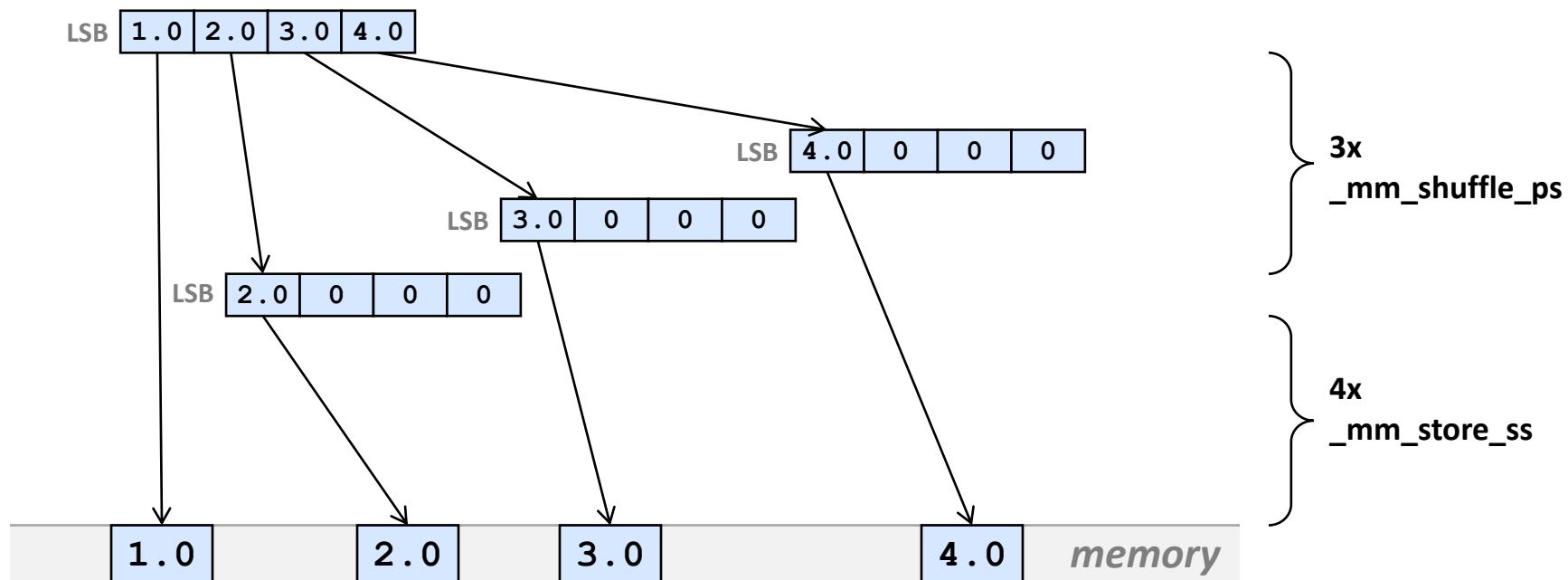
Example: Loading 4 Real Numbers from Arbitrary Memory Locations (cont'd)

- Do not do this (why?):

```
__declspec(align(16)) float g[4];
__m128 vf;

g[0] = *p0;
g[1] = *p1;
g[2] = *p2;
g[3] = *p3;
vf = _mm_load_ps(g);
```

Example: Storing 4 Real Numbers to Arbitrary Memory Locations



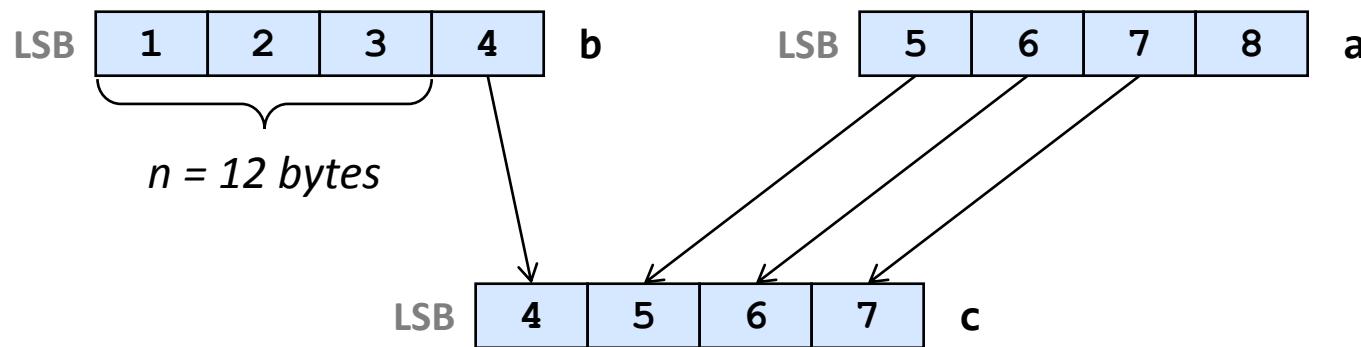
*7 instructions, shorter critical path
(before SSE4)*

Shuffle

```
__m128i _mm_alignr_epi8(__m128i a, __m128i b, const int n)
```

Concatenate a and b and extract byte-aligned result shifted to the right by n bytes

Example: View `__m128i` as 4 32-bit ints; $n = 12$



Use with `_mm_castsi128_ps` to do the same for floating point

Example

```
void shift(float *x, float *y, int n) {
    for (int i = 0; i < n-1; i++)
        y[i] = x[i+1];
    y[n-1] = 0;
}
```

```
#include <ia32intrin.h>

// n a multiple of 4, x, y are 16-byte aligned
void shift_vec(float *x, float *y, int n) {
    __m128 f;
    __m128i i1, i2, i3;

    i1 = _mm_castps_si128(_mm_load_ps(x));           // load first 4 floats and cast to int

    for (int i = 0; i < n-8; i = i + 4) {
        i2 = _mm_castps_si128(_mm_load_ps(x+4+i));   // load next 4 floats and cast to int
        f = _mm_castsi128_ps(_mm_alignr_epi8(i2,i1,4)); // shift and extract and cast back
        _mm_store_ps(y+i,f);                          // store it
        i1 = i2;                                     // make 2nd element 1st
    }

    // we are at the last 4
    i2 = _mm_castps_si128(_mm_setzero_ps());          // set the second vector to 0 and cast to int
    f = _mm_castsi128_ps(_mm_alignr_epi8(i2,i1,4)); // shift and extract and cast back
    _mm_store_ps(y+n-4,f);                          // store it
}
```

Vectorization

=



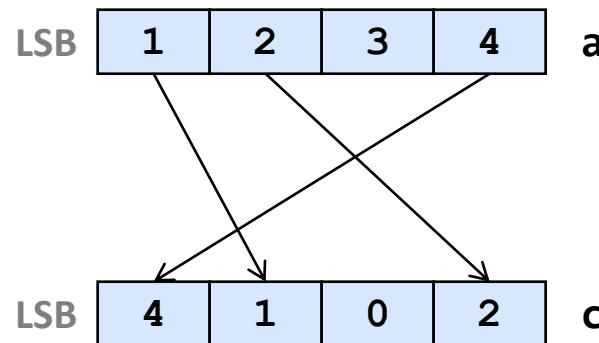
Picture: www.druckundbestell.de

Shuffle

```
__m128i _mm_shuffle_epi8(__m128i a, __m128i mask)
```

Result is filled in each position by any element of a or with 0, as specified by mask

Example: View `__m128i` as 4 32-bit ints



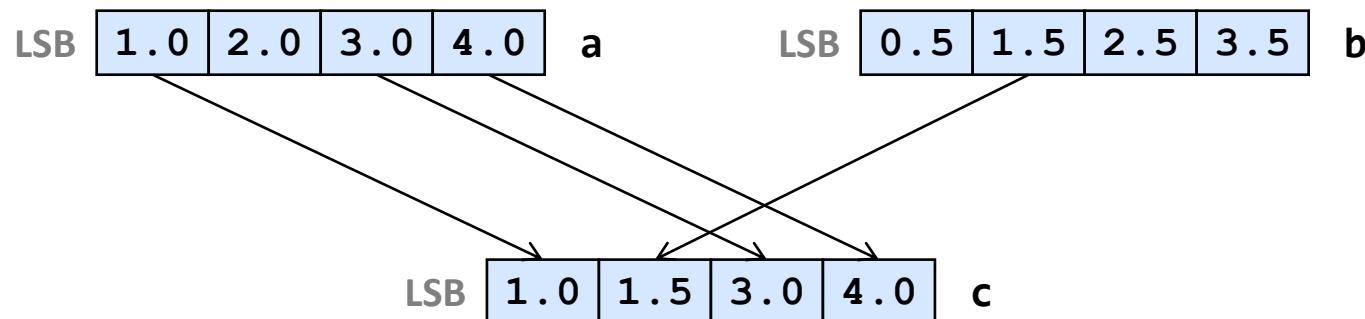
Use with `_mm_castsi128_ps` to do the same for floating point

Shuffle

```
__m128 _mm_blend_ps(__m128 a, __m128 b, const int mask)
```

(SSE4) Result is filled in each position by an element of a or b in the same position as specified by mask

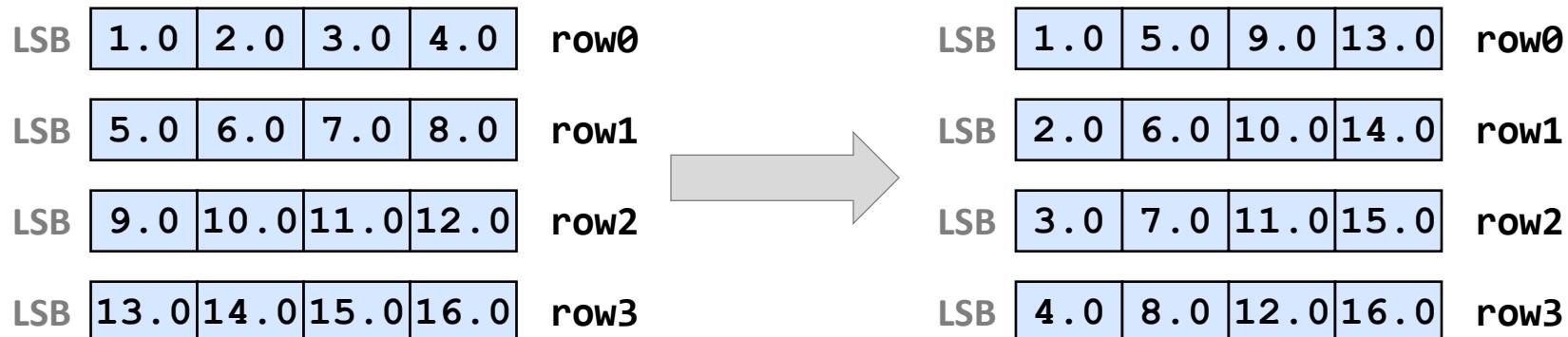
Example: mask = 2 = 0010



Shuffle

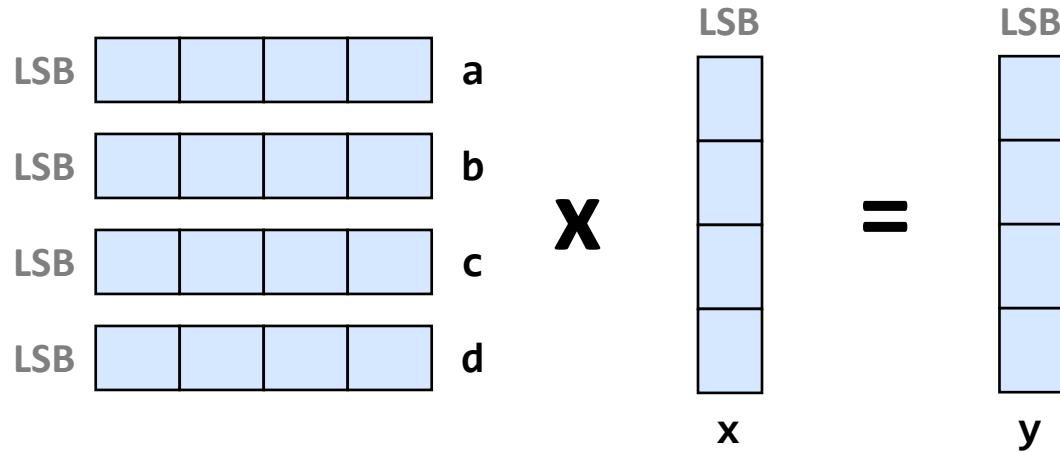
```
_MM_TRANSPOSE4_PS(row0, row1, row2, row3)
```

Macro for 4 x 4 matrix transposition: The arguments row0,..., row3 are __m128 values each containing a row of a 4 x 4 matrix. After execution, row0, .., row 3 contain the columns of that matrix.



In SSE: 8 shuffles (4 `_mm_unpacklo_ps`, 4 `_mm_unpackhi_ps`)

Example: 4 x 4 Matrix-Vector Product



Blackboard

Other Intrinsics

- Logical intrinsics (bitwise and, or, ...)

- Cacheability support intrinsics

- *Prefetch:*

```
void _mm_prefetch(char const *a, int sel)
```

- *Loads that bypass the cache:*

```
void _mm_stream_ps(float *p, __m128 a)
```

- Others

Vectorization With Intrinsics: Key Points

- Use aligned loads and stores
- Minimize overhead (shuffle instructions)
= maximize vectorization efficiency
- ***Definition:*** Vectorization efficiency

$$\frac{\text{Op count of scalar (unvectorized) code}}{\text{Op count (including shuffles) of vectorized code}}$$

- ***Ideally:*** Efficiency = v for v -way vector instructions
(assumes no vector instruction does more than 4 scalar ops)
- Examples (blackboard):
 - Adding two vectors of length 4
 - 4×4 matrix-vector multiplication

How to Write Fast Numerical Code

Spring 2011

Lecture 19

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Schedule

May 2011

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4



Today



Lecture



Project meetings



Project presentations

- 10 minutes each
- random order
- random speaker

**Final project paper and code due:
A week or so (exact date still TBD) after semester end**

SIMD Extensions and SSE

- Overview
- SSE family, floating point, and x87
- SSE intrinsics
- *Compiler vectorization*

References:

Intel icc manual (currently 12.0) → Creating parallel applications

→ *Automatic vectorization*

<http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/lin/index.htm>

Compiler Vectorization

- Compiler flags
- Aliasing
- Proper code style
- Alignment

Compiler Flags (icc 12.0)

Linux* OS and Mac OS* X	Windows* OS	Description
-vec -no-vec	/Qvec /Qvec-	Enables or disables vectorization and transformations enabled for vectorization. Vectorization is enabled by default. To disable, use -no-vec (Linux* and MacOS* X) or /Qvec- (Windows*) option. Supported on IA-32 and Intel® 64 architectures only.
-vec-report	/Qvec-report	Controls the diagnostic messages from the vectorizer. See Vectorization Report .
-simd -no-simd	/Qsimd /Qsimd-	Controls user-mandated (SIMD) vectorization. User-mandated (SIMD) vectorization is enabled by default. Use the -no-simd (Linux* or MacOS* X) or /Qsimd- (Windows*) option to disable SIMD transformations for vectorization.

Architecture flags:

Linux: -xHost ⊃ -mHost

Windows: /QxHost ⊃ /Qarch:Host

Host in {SSE2, SSE3, SSSE3, SSE4.1, SSE4.2}

Default: -mSSE2, /Qarch:SSE2

How Do I Know the Compiler Vectorized?

- vec-report (previous slide)
- Look at assembly: mulps, addps, xxxps
- Generate assembly with source code annotation:
 - Visual Studio + icc: /Fas
 - icc on Linux/Mac: -S

```

void myadd(float *a, float *b, const int n) {
    for (int i = 0; i < n; i++)
        a[i] = a[i] + b[i];
}

```

Example

unvectorized: /Qvec-

```

<more>
;;;     a[i] = a[i] + b[i];
movss     xmm0, DWORD PTR [rcx+rax*4]
addss     xmm0, DWORD PTR [rdx+rax*4]
movss     DWORD PTR [rcx+rax*4], xmm0
<more>

```

vectorized:

```

<more>
;;;     a[i] = a[i] + b[i];
movss     xmm0, DWORD PTR [rcx+r11*4]
addss     xmm0, DWORD PTR [rdx+r11*4]
movss     DWORD PTR [rcx+r11*4], xmm0
...
movups    xmm0, XMMWORD PTR [rdx+r10*4]
movups    xmm1, XMMWORD PTR [16+rdx+r10*4]
addps     xmm0, XMMWORD PTR [rcx+r10*4]
addps     xmm1, XMMWORD PTR [16+rcx+r10*4]
movaps    XMMWORD PTR [rcx+r10*4], xmm0
movaps    XMMWORD PTR [16+rcx+r10*4], xmm1
<more>

```



why this?



why everything twice?
why movups and movaps?

↑
unaligned

↑
aligned

Aliasing

```
for (i = 0; i < n; i++)
    a[i] = a[i] + b[i];
```

Cannot be vectorized in a straightforward way due to potential aliasing.

However, in this case compiler can insert runtime check:

```
if (a + n < b || b + n < a)
    /* vectorized loop */
    ...
else
    /* serial loop */
    ...
```

Removing Aliasing

- Globally with compiler flag:
 - `-fno-alias`, `/Oa`
 - `-fargument-noalias`, `/Qalias-args-` (function arguments only)
- For one loop: pragma

```
void add(float *a, float *b, int n) {  
    #pragma ivdep  
    for (i = 0; i < n; i++)  
        a[i] = a[i] + b[i];  
}
```

- For specific arrays: restrict (needs compiler flag `-restrict`, `/Qrestrict`)

```
void add(float *restrict a, float *restrict b, int n) {  
    for (i = 0; i < n; i++)  
        a[i] = a[i] + b[i];  
}
```

Proper Code Style

- Use countable loops = number of iterations known at runtime
 - *Number of iterations is a:*
 - constant
 - loop invariant term
 - linear function of outermost loop indices
- Countable or not?

```
for (i = 0; i < n; i++)
    a[i] = a[i] + b[i];
```

```
void vsum(float *a, float *b, float *c) {
    int i = 0;

    while (a[i] > 0.0) {
        a[i] = b[i] * c[i];
        i++;
    }
}
```

Proper Code Style

- Use arrays, structs of arrays, not arrays of structs
- Ideally: unit stride access in innermost loop

```
void mmm1(float *a, float *b, float *c) {  
    int N = 100;  
    int i, j, k;  
  
    for (i = 0; i < N; i++)  
        for (j = 0; j < N; j++)  
            for (k = 0; k < N; k++)  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
}
```

```
void mmm2(float *a, float *b, float *c) {  
    int N = 100;  
    int i, j, k;  
  
    for (i = 0; i < N; i++)  
        for (k = 0; k < N; k++)  
            for (j = 0; j < N; j++)  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
}
```

Alignment

```
float x[1024];
int i;

for (i = 0; i < 1024; i++)
    x[i] = 1;
```

Cannot be vectorized in a straightforward way since x may not be aligned

However, the compiler can peel the loop to extract aligned part:

```
float x[1024];
int i;

peel = x & 0x0f; /* x mod 16 */
if (peel != 0) {
    peel = 16 - peel;
    /* initial segment */
    for (i = 0; i < peel; i++)
        x[i] = 1;
}
/* 16-byte aligned access */
for (i = peel; i < 1024; i++)
    x[i] = 1;
```

Ensuring Alignment

- Align arrays to 16-byte boundaries (see earlier discussion)
- If compiler cannot analyze:

- Use pragma for loops

```
float x[1024];
int i;

#pragma vector aligned
for (i = 0; i < 1024; i++)
    x[i] = 1;
```

- For specific arrays:
`__assume_aligned(a, 16);`

```

void myadd(float *a, float *b, const int n) {
    for (int i = 0; i < n; i++)
        a[i] = a[i] + b[i];
}

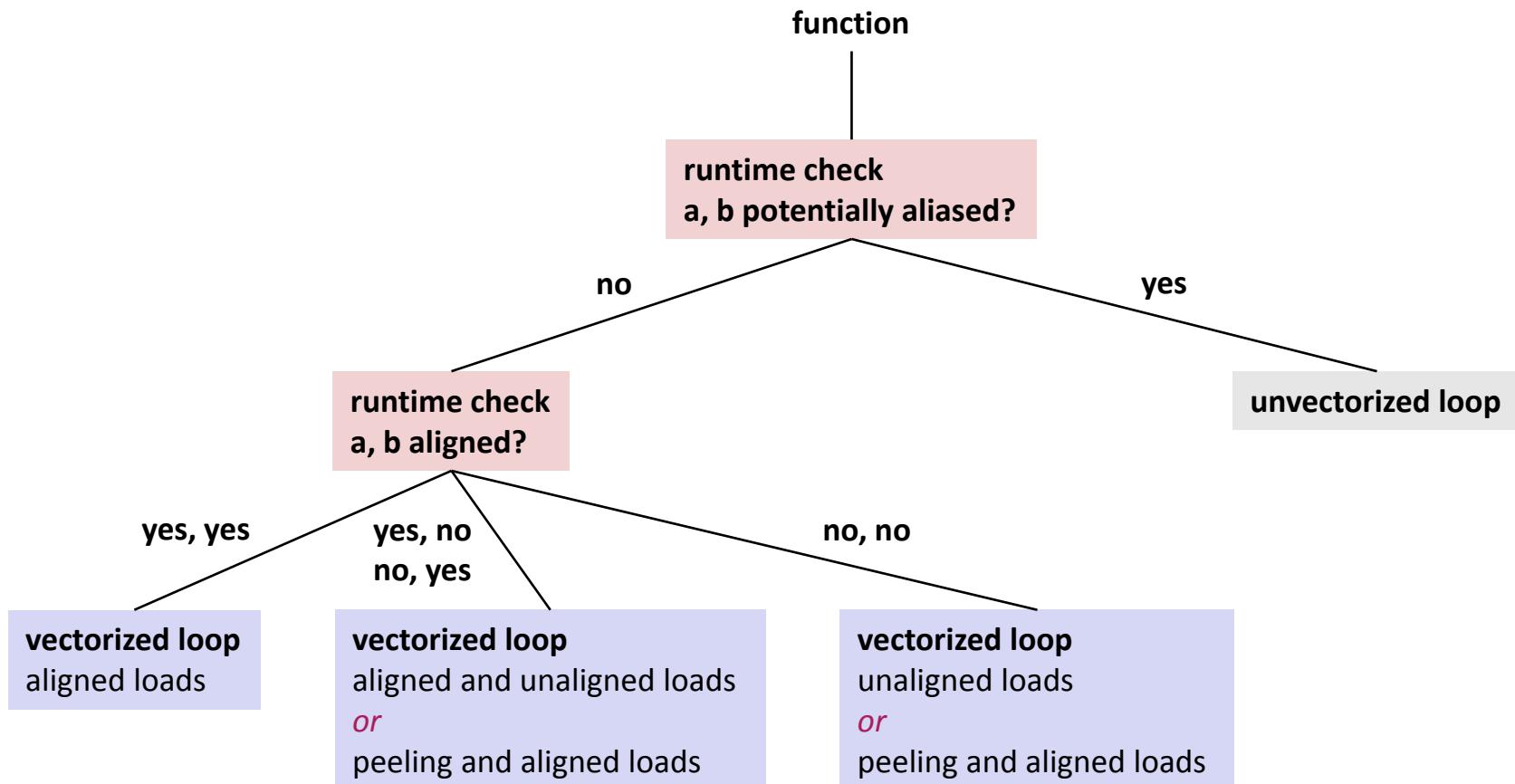
```

Assume:

- No aliasing information
- No alignment information

Can compiler vectorize?

Yes: Through versioning



Compiler Vectorization

- Read manual

Linear Transforms

- Overview
- Discrete Fourier transform
- Fast Fourier transforms
- Optimized implementation and autotuning (FFTW)
- Automatic implementation and optimization (Spiral)

Linear Transforms

- Very important class of functions: signal processing, scientific computing, ...
- *Mathematically:* Change of basis = Multiplication by a fixed matrix T

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = y = Tx \quad \xleftarrow{\hspace{1cm}} \quad T \cdot \quad \xleftarrow{\hspace{1cm}} \quad x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

$T = [t_{k,\ell}]_{0 \leq k,\ell < n}$

Output

Input

- Equivalent definition: Summation form

$$y_k = \sum_{\ell=0}^{n-1} t_{k,\ell} x_\ell, \quad 0 \leq k < n$$

Smallest Relevant Example

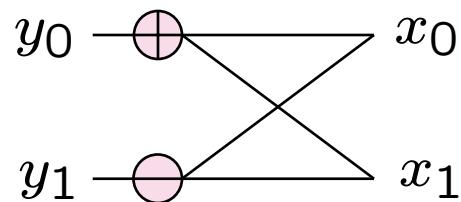
Transform (matrix): $T = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

Computation: $y = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} x$

or

$$\begin{aligned} y_0 &= x_0 + x_1 \\ y_1 &= x_0 - x_1 \end{aligned}$$

As graph (direct acyclic graph or DAG):



called a butterfly



[http://charlottesmartypants.blogspot.com/
2011_02_01_archive.html](http://charlottesmartypants.blogspot.com/2011_02_01_archive.html)

Transforms: Examples

- A few dozen transforms are relevant
- Some examples

$$\mathbf{DFT}_n = [e^{-2k\ell\pi i/n}]_{0 \leq k, \ell < n}$$

$$\mathbf{RDFT}_n = [r_{k\ell}]_{0 \leq k, \ell < n}, \quad r_{k\ell} = \begin{cases} \cos \frac{2\pi k\ell}{n}, & k \leq \lfloor \frac{n}{2} \rfloor \\ -\sin \frac{2\pi k\ell}{n}, & k > \lfloor \frac{n}{2} \rfloor \end{cases}$$

universal tool

$$\mathbf{DHT} = [\cos(2k\ell\pi/n) + \sin(2k\ell\pi/n)]_{0 \leq k, \ell < n}$$

$$\mathbf{WHT}_n = \begin{bmatrix} \mathbf{WHT}_{n/2} & \mathbf{WHT}_{n/2} \\ \mathbf{WHT}_{n/2} & -\mathbf{WHT}_{n/2} \end{bmatrix}, \quad \mathbf{WHT}_2 = \mathbf{DFT}_2$$

$$\mathbf{IMDCT}_n = [\cos((2k+1)(2\ell+1+n)\pi/4n)]_{0 \leq k < 2n, 0 \leq \ell < n}$$

MPEG

$$\mathbf{DCT-2}_n = [\cos(k(2\ell+1)\pi/2n)]_{0 \leq k, \ell < n}$$

JPEG

$$\mathbf{DCT-3}_n = \mathbf{DCT-2}_n^T \quad (\text{transpose})$$

$$\mathbf{DCT-4}_n = [\cos((2k+1)(2\ell+1)\pi/4n)]_{0 \leq k, \ell < n}$$

Blackboard

- Discrete Fourier transform (DFT)
- Transform algorithms
- Fast Fourier transform, size 4

How to Write Fast Numerical Code

Spring 2011

Lecture 20

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Schedule

May 2011

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4



Today



Lecture



Project meetings

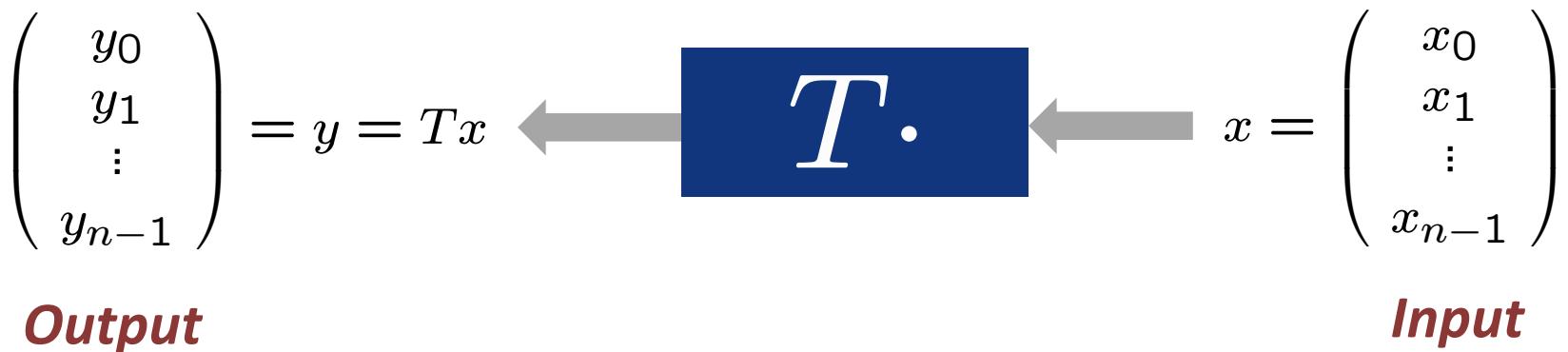


Project presentations

- 10 minutes each
- random order
- random speaker

**Final project paper and code due:
A week or so (exact date still TBD) after semester end**

Linear Transforms



Example: $T = \text{DFT}_n = [e^{-2k\ell\pi i/n}]_{0 \leq k, \ell < n}$

$$= [\omega_n^{k\ell}]_{0 \leq k, \ell < n}, \quad \omega_n = e^{-2\pi i/n}$$

Algorithms: Example FFT, n = 4

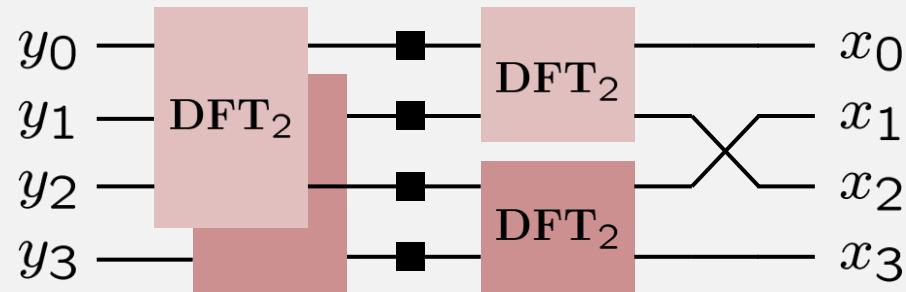
Fast Fourier transform (FFT)

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} x = \begin{bmatrix} 1 & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & 1 \\ 1 & \cdot & -1 & \cdot \\ \cdot & 1 & \cdot & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & i \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdot & \cdot \\ 1 & -1 & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix} x$$

Representation using matrix algebra

$$\text{DFT}_4 = (\text{DFT}_2 \otimes \text{I}_2) \text{ diag}(1, 1, 1, i) (\text{I}_2 \otimes \text{DFT}_2) L_2^4$$

Data flow graph



How to Write Fast Numerical Code

Spring 2011

Lecture 21

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Schedule

May 2011

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4



Today



Lecture



Project presentations

- 10 minutes each
- random order
- random speaker

Final project paper and code due:

Friday, June 10th

FFT References

- **Complexity:** Bürgisser, Clausen, Shokrollahi, *Algebraic Complexity Theory*, Springer, 1997
- **History:** Heideman, Johnson, Burrus: *Gauss and the History of the Fast Fourier Transform*, Arch. Hist. Sc. 34(3) 1985
- **FFTs:**
 - Cooley and Tukey, *An algorithm for the machine calculation of complex Fourier series,*" Math. of Computation, vol. 19, pp. 297–301, 1965
 - Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, 2nd ed., Springer, 1982
 - van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, 1992
 - Tolimieri, An, Lu, *Algorithms for Discrete Fourier Transforms and Convolution*, Springer, 2nd edition, 1997
 - Franchetti, Püschel, Voronenko, Chellappa and Moura, *Discrete Fourier Transform on Multicore*, IEEE Signal Processing Magazine, special issue on ``Signal Processing on Platforms with Multiple Cores'', Vol. 26, No. 6, pp. 90-102, 2009
- **FFTW:** www.fftw.org
 - Frigo and Johnson, *FFTW: An Adaptive Software Architecture for the FFT*, Proc. ICASSP, vol. 3, pp. 1381-1384
 - M. Frigo, *A fast Fourier transform compiler*, in Proc. PLDI, 1999

Discrete Fourier Transform

- Defined for all sizes n:

$$y = \mathbf{DFT}_n x$$

$$\mathbf{DFT}_n = [\omega_n^{k\ell}]_{0 \leq k, \ell < n}, \quad \omega_n = e^{-2\pi i / n}$$

Complexity of the DFT

■ Measure: L_c , $2 \leq c$

- Complex adds count 1
- Complex mult by a constant a with $|a| < c$ counts 1
- L_2 is strictest, L_∞ the loosest (and most natural)

■ Upper bounds:

- $n = 2^k$: $L_2(DFT_n) \leq 3/2 n \log_2(n)$ *(using Cooley-Tukey FFT)*
- General n : $L_2(DFT_n) \leq 8 n \log_2(n)$ *(needs Bluestein FFT)*

■ Lower bound:

- Theorem by Morgenstern: If $c < \infty$, then $L_c(DFT_n) \geq \frac{1}{2} n \log_c(n)$
- Implies: in the measure L_c , the DFT is $\Theta(n \log(n))$

History of FFTs

- The advent of digital signal processing is often attributed to the FFT
(Cooley-Tukey 1965)
- History:
 - Around 1805: FFT discovered by Gauss [1]
(Fourier publishes the concept of Fourier analysis in 1807!)
 - 1965: Rediscovered by Cooley-Tukey

[1]: Heideman, Johnson, Burrus: "Gauss and the History of the Fast Fourier Transform" Arch. Hist. Sc. 34(3) 1985

Carl-Friedrich Gauss



1777 - 1855

- **Contender for the greatest mathematician of all times**
- **Some contributions:** Modular arithmetic, least square analysis, normal distribution, fundamental theorem of algebra, Gauss elimination, Gauss quadrature, Gauss-Seidel, non-euclidean geometry, ...

Example FFT, n = 4

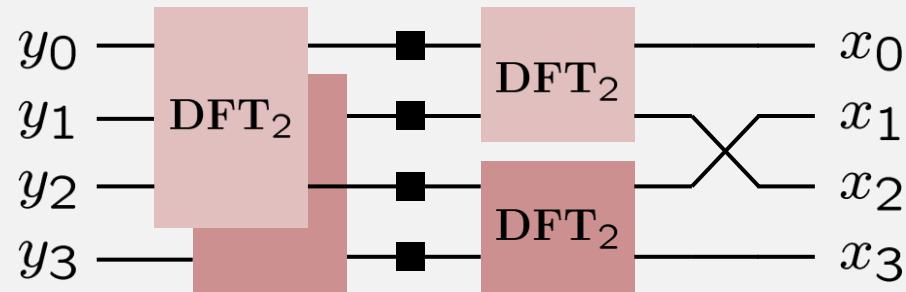
Fast Fourier transform (FFT)

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} x = \begin{bmatrix} 1 & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & 1 \\ 1 & \cdot & -1 & \cdot \\ \cdot & 1 & \cdot & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & i \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdot & \cdot \\ 1 & -1 & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix} x$$

Representation using matrix algebra

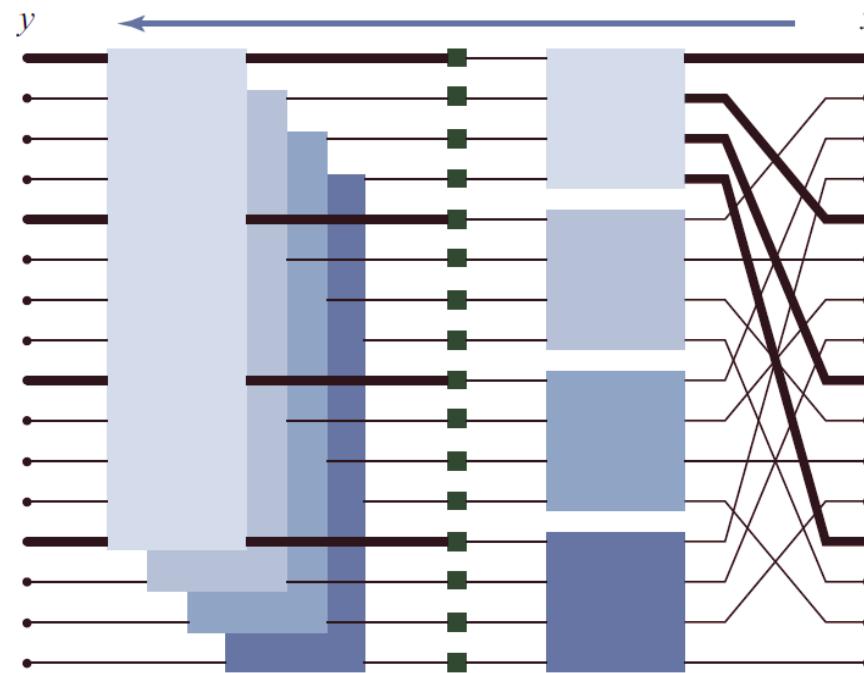
$$\text{DFT}_4 = (\text{DFT}_2 \otimes \text{I}_2) \text{ diag}(1, 1, 1, i) (\text{I}_2 \otimes \text{DFT}_2) L_2^4$$

Data flow graph



Example FFT, n = 16 (*Recursive, Radix 4*)

$$\text{DFT}_{16} = \begin{matrix} & \text{DFT}_4 \otimes I_4 & T_4^{16} & I_4 \otimes \text{DFT}_4 & L_4^{16} \end{matrix}$$



FFTs

- Recursive, general radix, decimation-in-time/decimation-in-frequency
 $\text{radix} = k$

$$\mathbf{DFT}_{km} = (\mathbf{DFT}_k \quad \mathbf{I}_m) T_m^{km} (\mathbf{I}_k \quad \mathbf{DFT}_m) L_k^{km}$$

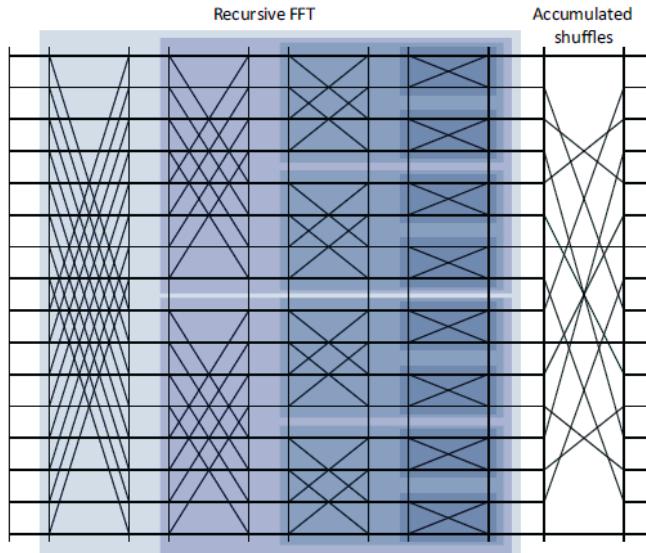
$$\mathbf{DFT}_{km} = L_m^{km} (\mathbf{I}_k \quad \mathbf{DFT}_m) T_m^{km} (\mathbf{DFT}_k \quad \mathbf{I}_m)$$

- Iterative, radix 2, decimation-in-time/decimation-in-frequency

$$\mathbf{DFT}_{2^t} = \left(\prod_{j=1}^t (\mathbf{I}_{2^{j-1}} \quad \mathbf{DFT}_2 \quad \mathbf{I}_{2^{t-j}}) \cdot (\mathbf{I}_{2^{j-1}} \quad T_{2^{t-j}}^{2^{t-j+1}}) \right) \cdot R_{2^t}$$

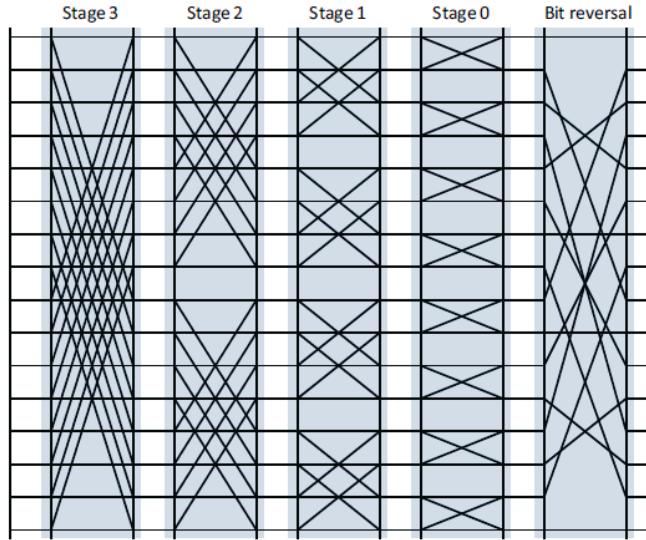
$$\mathbf{DFT}_{2^t} = R_{2^t} \cdot \left(\prod_{j=1}^t (\mathbf{I}_{2^{t-j}} \quad T_{2^{j-1}}^{2^j}) \cdot (\mathbf{I}_{2^{t-j}} \quad \mathbf{DFT}_2 \quad \mathbf{I}_{2^{j-1}}) \right)$$

Radix 2, recursive



$$(\text{DFT}_2 \otimes I_8) T_8^{16} \left(I_2 \otimes \left((\text{DFT}_2 \otimes I_4) T_4^8 \left(I_2 \otimes ((\text{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \text{DFT}_2) L_2^4) \right) L_2^8 \right) \right) L_2^{16}$$

Radix 2, iterative



$$\left((I_1 \otimes \text{DFT}_2 \otimes I_8) D_0^{16} \right) \left((I_2 \otimes \text{DFT}_2 \otimes I_4) D_1^{16} \right) \left((I_4 \otimes \text{DFT}_2 \otimes I_2) D_2^{16} \right) \left((I_8 \otimes \text{DFT}_2 \otimes I_1) D_3^{16} \right) R_2^{16}$$

Recursive vs. Iterative

- Iterative FFT computes in stages of butterflies = $\log_2(n)$ passes through the data
- Recursive FFT reduces passes through data = better locality
- Same computation graph but different topological sorting
- Rough analogy:

MMM	DFT
Triple loop	Iterative FFT
Blocked	Recursive FFT

Fast Implementation (\approx FFTW 2.x)

- Choice of algorithm
- Locality optimization
- Constants
- Fast basic blocks
- Adaptivity

- Blackboard

How to Write Fast Numerical Code

Spring 2011

Lecture 22

Instructor: Markus Püschel

TA: Georg Ofenbeck



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Schedule

May 2011

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
			10		<i>Final code and paper due</i>	



Today



Lecture

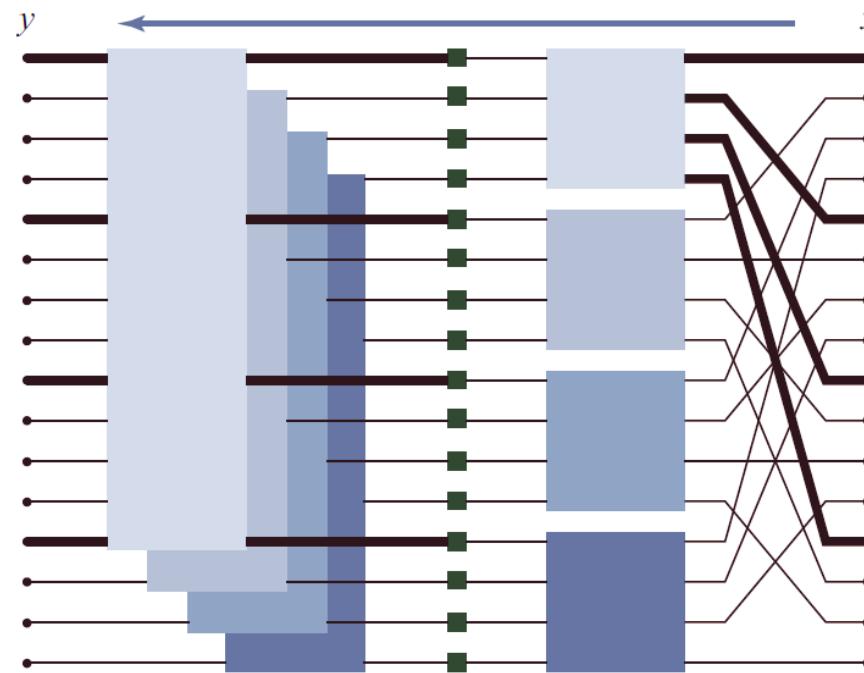


Project presentations

- 10 minutes each
- random order
- random speaker

Example FFT, $n = 16$ (Recursive, Radix 4)

$$\text{DFT}_{16} = \begin{matrix} & \text{DFT}_4 \otimes I_4 & T_4^{16} & I_4 \otimes \text{DFT}_4 & L_4^{16} \\ \text{DFT}_{16} & = & \begin{array}{|c|c|c|c|} \hline & \text{DFT}_4 \otimes I_4 & T_4^{16} & I_4 \otimes \text{DFT}_4 & L_4^{16} \\ \hline \end{array} & \begin{array}{|c|c|c|c|} \hline & \text{DFT}_4 \otimes I_4 & T_4^{16} & I_4 \otimes \text{DFT}_4 & L_4^{16} \\ \hline \end{array} & \begin{array}{|c|c|c|c|} \hline & \text{DFT}_4 \otimes I_4 & T_4^{16} & I_4 \otimes \text{DFT}_4 & L_4^{16} \\ \hline \end{array} \\ \end{matrix}$$



Fast Implementation (\approx FFTW 2.x)

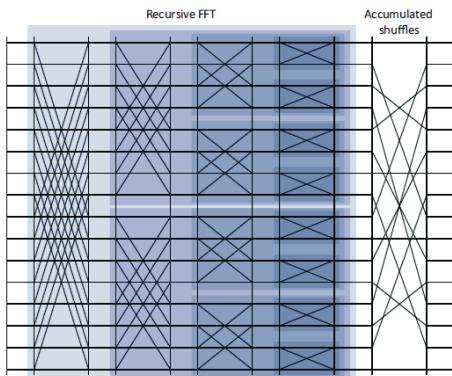
- Choice of algorithm
- Locality optimization
- Constants
- Fast basic blocks
- Adaptivity

1: Choice of Algorithm

- Choose recursive, not iterative

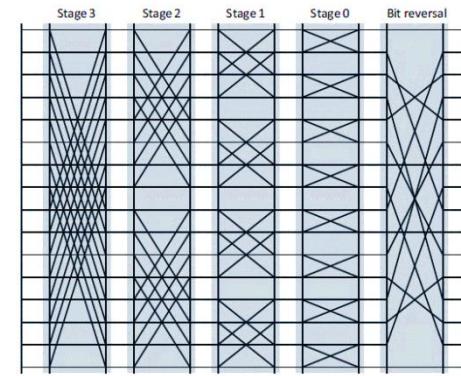
$$\mathbf{DFT}_{km} = (\mathbf{DFT}_k \quad \mathbf{I}_m) T_m^{km} (\mathbf{I}_k \quad \mathbf{DFT}_m) L_k^{km}$$

Radix 2, recursive



$$(\mathbf{DFT}_2 \otimes \mathbf{I}_8) T_8^{16} \left(I_2 \otimes \left((\mathbf{DFT}_2 \otimes \mathbf{I}_4) T_4^8 \left(I_2 \otimes \left((\mathbf{DFT}_2 \otimes \mathbf{I}_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4 \right) L_2^8 \right) \right) L_2^{16} \right)$$

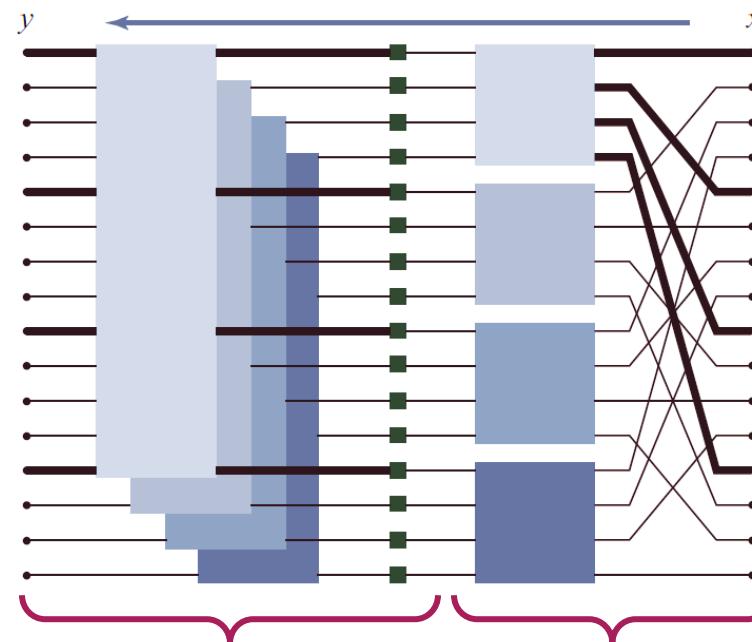
Radix 2, iterative



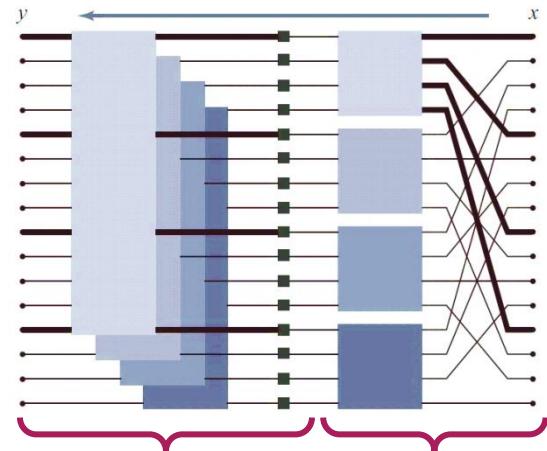
$$\left((I_1 \otimes \mathbf{DFT}_2 \otimes \mathbf{I}_8) D_0^{16} \right) \left((I_2 \otimes \mathbf{DFT}_2 \otimes \mathbf{I}_4) D_1^{16} \right) \left((I_4 \otimes \mathbf{DFT}_2 \otimes \mathbf{I}_2) D_2^{16} \right) \left((I_8 \otimes \mathbf{DFT}_2 \otimes \mathbf{I}_1) D_3^{16} \right) R_2^{16}$$

2: Locality Improvement: Fuse Stages

$$\text{DFT}_{16} = \underbrace{\begin{array}{c} \text{DFT}_4 \otimes I_4 \\ \text{T}_4^{16} \\ I_4 \otimes \text{DFT}_4 \\ L_4^{16} \end{array}}_{\text{Four stages}}$$



$$\mathbf{DFT}_{km} = \underbrace{(\mathbf{DFT}_k \quad \mathbf{I}_m) T_m^{km}}_{\text{left part}} (\mathbf{I}_k \quad \mathbf{DFT}_m) L_k^{km}$$



```
// code sketch
void DFT(int n, cpx *y, cpx *x) {
    int k = choose_dft_radix(n); // ensure k <= 32

    if (use_base_case(n))
        DFTbc(n, y, x); // use base case
    else {
        for (int i=0; i < k; ++i)
            DFTrec(m, y + m*i, x + i, k, 1); // implemented as DFT(..) is
        for (int j=0; j < m; ++j)
            DFTscaled(k, y + j, t[j], m); // always a base case
    }
}
```

3: Constants

- FFT incurs multiplications by roots of unity
- In real arithmetic: Multiplications by sines and cosines, e.g.,

```
y[i] = sin(i·pi/128)*x[i];
```

- Very expensive!
- Solution: Precompute once and use many times

```
d = DFT_init(1024); // init function computes constant table  
d(y, x);           // use many times
```

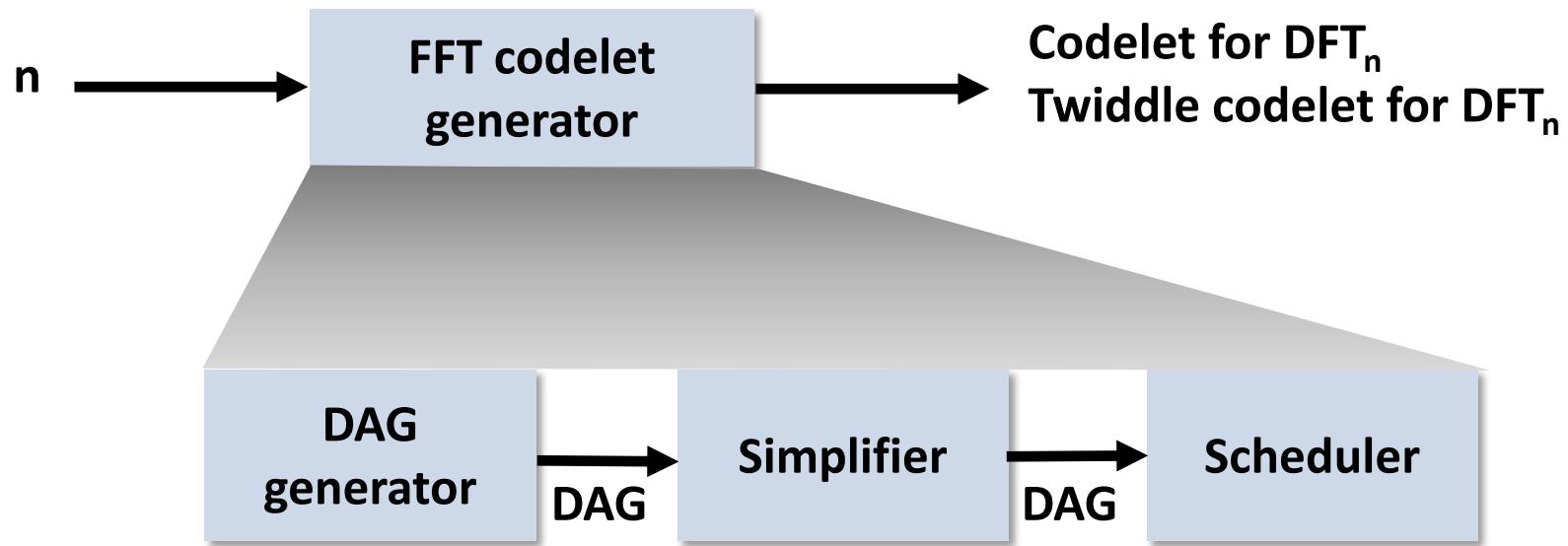
4: Optimized Basic Blocks

```
// code sketch
void DFT(int n, cpx *y, cpx *x) {
    int k = choose_dft_radix(n); // ensure k <= 32

    if (use_base_case(n))
        DFTbc(n, y, x); // use base case
    else {
        for (int i=0; i < k; ++i)
            DFTrec(m, y + m*i, x + i, k, 1); // implemented as DFT(..) is
        for (int j=0; j < m; ++j)
            DFTscaled(k, y + j, t[j], m); // always a base case
    }
}
```

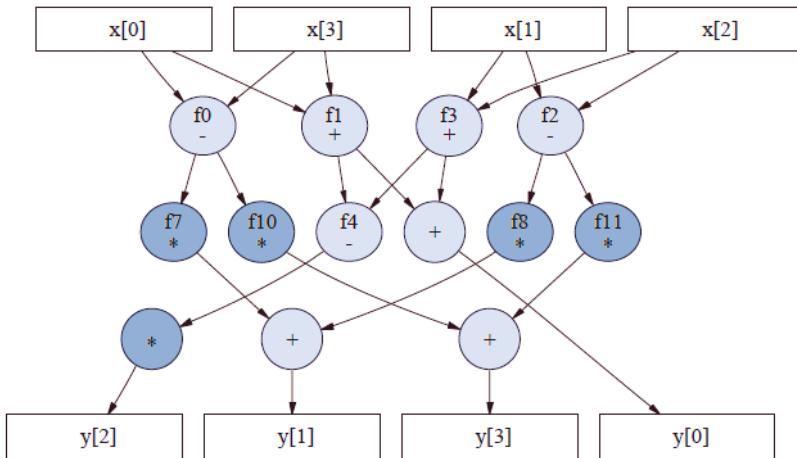
- Empirical study: Base cases for sizes $n \leq 32$ useful (scalar code)
- Needs 62 base case or “codelets” (why?)
 - DFTrec, sizes 2–32
 - DFTscaled, sizes 2–32
- Solution: Codelet generator

FFTW Codelet Generator



Small Example DAG

DAG:



One possible unparsing:

```
f0 = x[0] - x[3];
f1 = x[0] + x[3];
f2 = x[1] - x[2];
f3 = x[1] + x[2];
f4 = f1 - f3;
y[0] = f1 + f3;
y[2] = 0.7071067811865476 * f4;
f7 = 0.9238795325112867 * f0;
f8 = 0.3826834323650898 * f2;
y[1] = f7 + f8;
f10 = 0.3826834323650898 * f0;
f11 = (-0.9238795325112867) * f2;
y[3] = f10 + f11;
```

DAG Generator



- Knows FFTs: Cooley-Tukey, split-radix, Good-Thomas, Rader, represented in sum notation

$$y_{n_2j_1+j_2} = \sum_{k_1=0}^{n_1-1} \left(\omega_n^{j_2 k_1} \right) \left(\sum_{k_2=0}^{n_2-1} x_{n_1 k_2 + k_1} \omega_{n_2}^{j_2 k_2} \right) \omega_{n_1}^{j_1 k_1}$$

- For given n, suitable FFTs are recursively applied to yield n (real) expression trees for outputs y_0, \dots, y_{n-1}
- Trees are fused to an (unoptimized) DAG

Simplifier



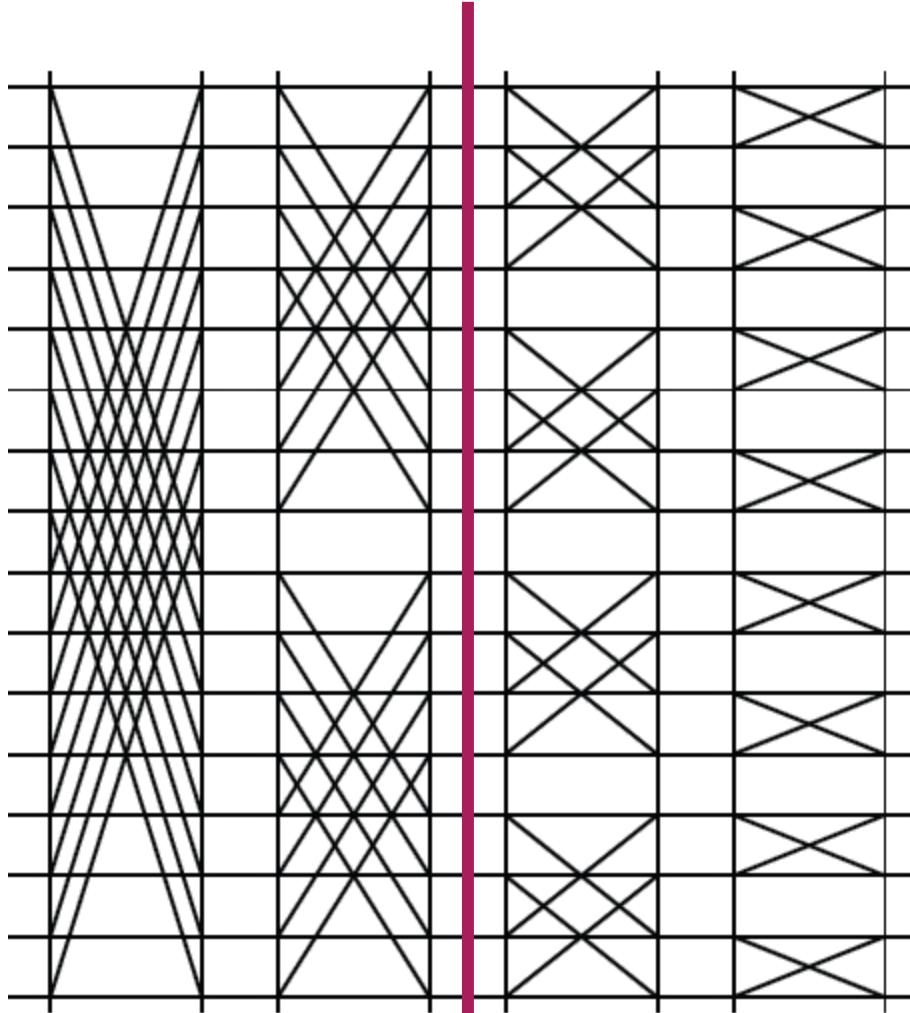
- **Applies:**
 - Algebraic transformations
 - Common subexpression elimination (CSE)
 - DFT-specific optimizations
- **Algebraic transformations**
 - Simplify mults by 0, 1, -1
 - Distributivity law: $kx + ky = k(x + y)$, $kx + lx = (k + l)x$
Canonicalization: $(x-y)$, $(y-x)$ to $(x-y)$, $-(x-y)$
- **CSE: standard**
 - E.g., two occurrences of $2x+y$: assign new temporary variable
- **DFT specific optimizations**
 - All numeric constants are made positive (reduces register pressure)
 - CSE also on transposed DAG

Scheduler



- Determines in which sequence the DAG is unparsed to C (topological sort of the DAG)
Goal: minimizer register spills
- If R registers are available, then a 2-power FFT needs at least $\Omega(n \log(n)/R)$ register spills [1]
Same holds for a fully associative cache
- FFTW's scheduler achieves this (asymptotic) bound **independent** of R
- Blackboard

[1] Hong and Kung: "I/O Complexity: The red-blue pebbling game"



First cut

Codelet Examples

- [Notwiddle 2](#)
- [Notwiddle 3](#)
- [Twiddle 3](#)
- [Notwiddle 32](#)

- **Code style:**
 - Single static assignment (SSA)
 - Scoping (limited scope where variables are defined)

5: Adaptivity

```
// code sketch
void DFT(int n, cpx *y, cpx *x) {
    int k = choose_dft_radix(n); // ensure k <= 32

    if (use_base_case(n))
        DFTbc(n, y, x); // use base case
    else {
        for (int i=0; i < k; ++i)
            DFTrec(m, y + m*i, x + i, k, 1); // implemented as DFT
        for (int j=0; j < m; ++j)
            DFTscaled(k, y + j, t[j], m); // always a base case
    }
}
```

Choices used for platform adaptation

```
d = DFT_init(1024); // compute constant table; search for best recursion
d(y, x);           // use many times
```

- Search strategy: Dynamic programming
- Blackboard

	MMM Atlas	Sparse MVM Sparsity/Bebop	DFT FFTW
Cache optimization	Blocking	Blocking (rarely useful)	Recursive FFT, fusion of steps
Register optimization	Blocking	Blocking (sparse format)	Scheduling of small FFTs
Optimized basic blocks	Unrolling, scalar replacement and SSA, scheduling, simplifications (for FFT)		
Other optimizations	—	—	Precomputation of constants
Adaptivity	Search: blocking parameters	Search: register blocking size	Search: recursion strategy

Spiral

Computer Synthesis of Computational Programs

Joint work with ...

Franz Franchetti
Yevgen Voronenko
Srinivas Chellappa
Frédéric de Mesmay
Daniel McFarlin
José Moura
James Hoe

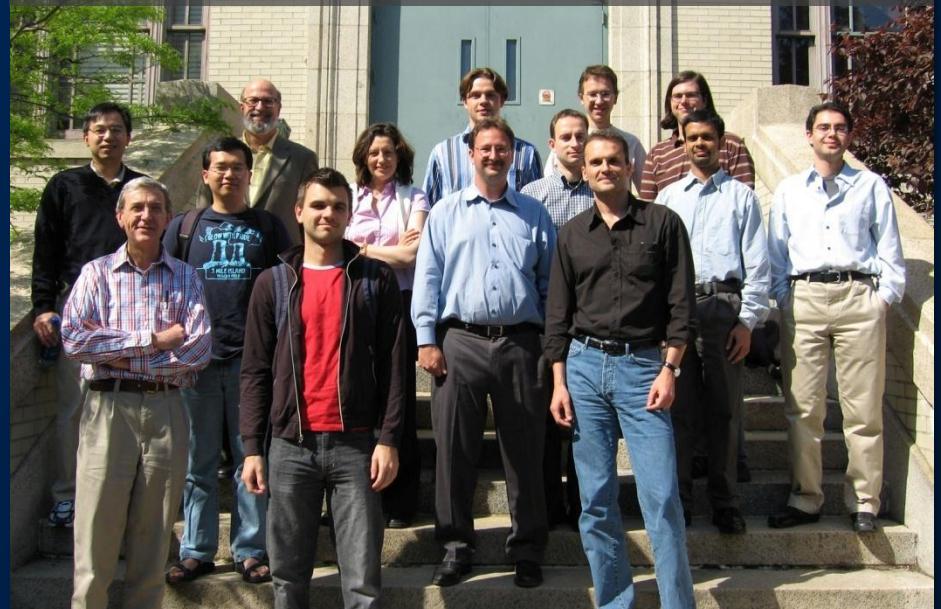
...

Markus Püschel
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

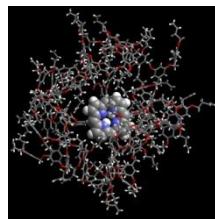
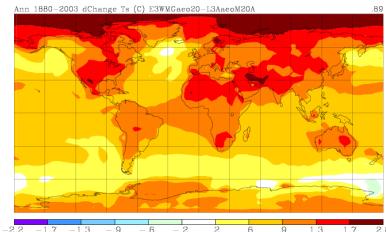
SPIRAL 
www.spiral.net

...and the Spiral team (only part shown)



Supported by DARPA, ONR, NSF, Intel, Mercury

Scientific Computing



Physics/biology simulations

Computing

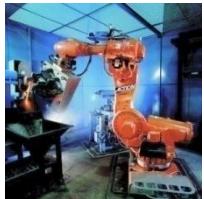
- Unlimited need for performance
- Large set of applications, but ...
- Relatively small set of critical components (100s to 1000s)
 - Matrix multiplication
 - Discrete Fourier transform
 - Viterbi decoder
 - Filter/stencil
 -

Consumer Computing



Audio/image/video processing

Embedded Computing

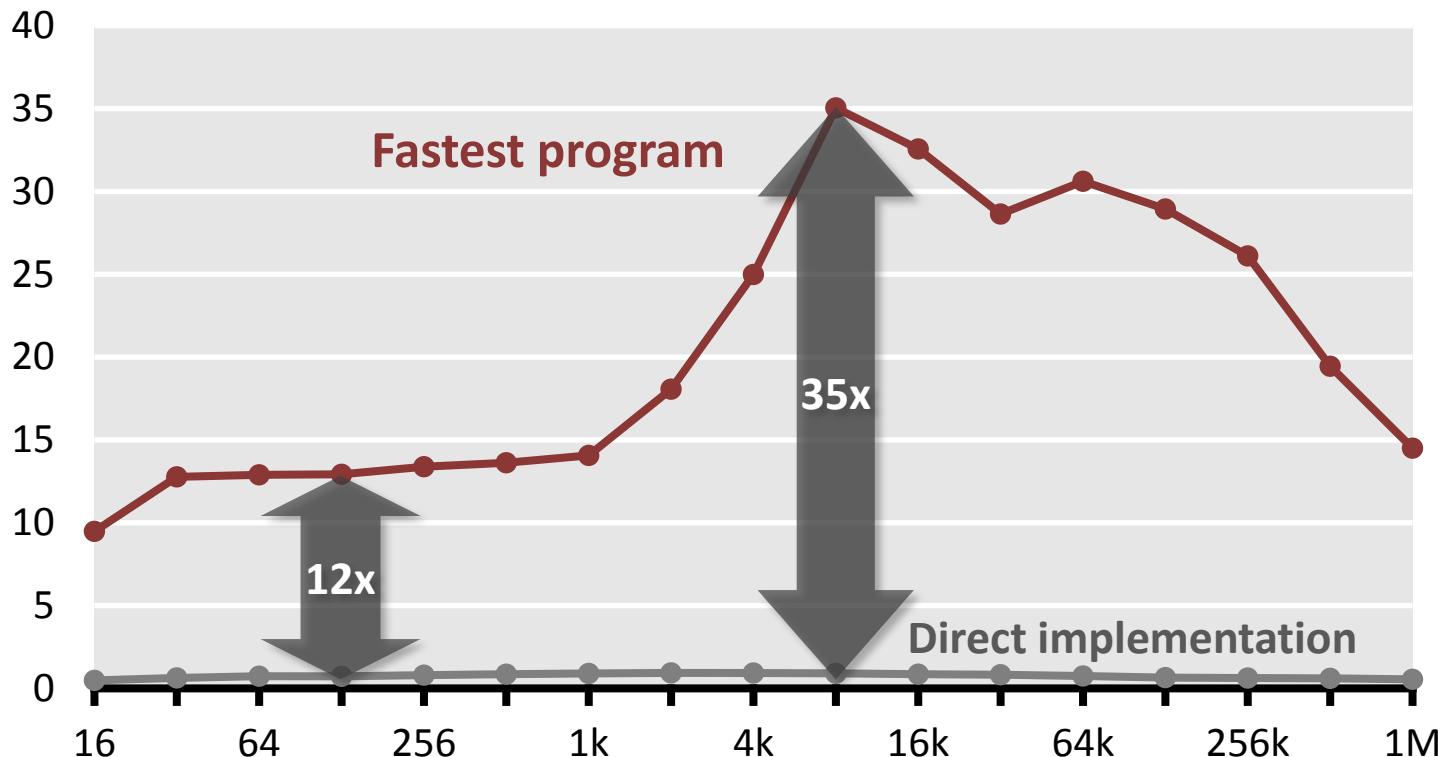


Signal processing, communication, control

The Problem: Example DFT

DFT on Intel Core i7 (4 Cores, 2.66 GHz)

Performance [Gflop/s]

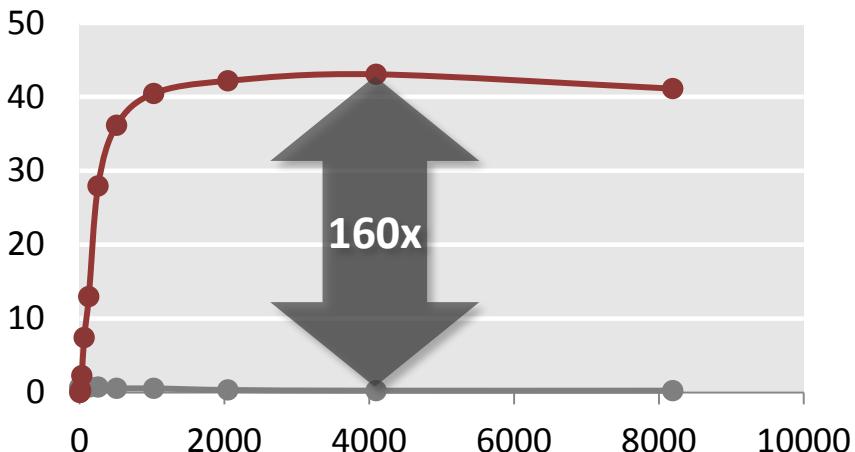


- Same number of operations
- Best compiler

The Problem Is Everywhere

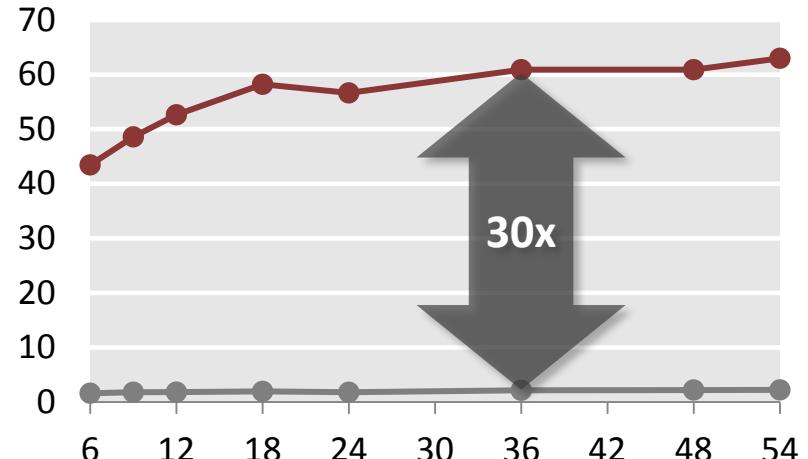
Matrix multiplication

Performance [Gflop/s]



WiFi Receiver

Performance [Mbit/s]

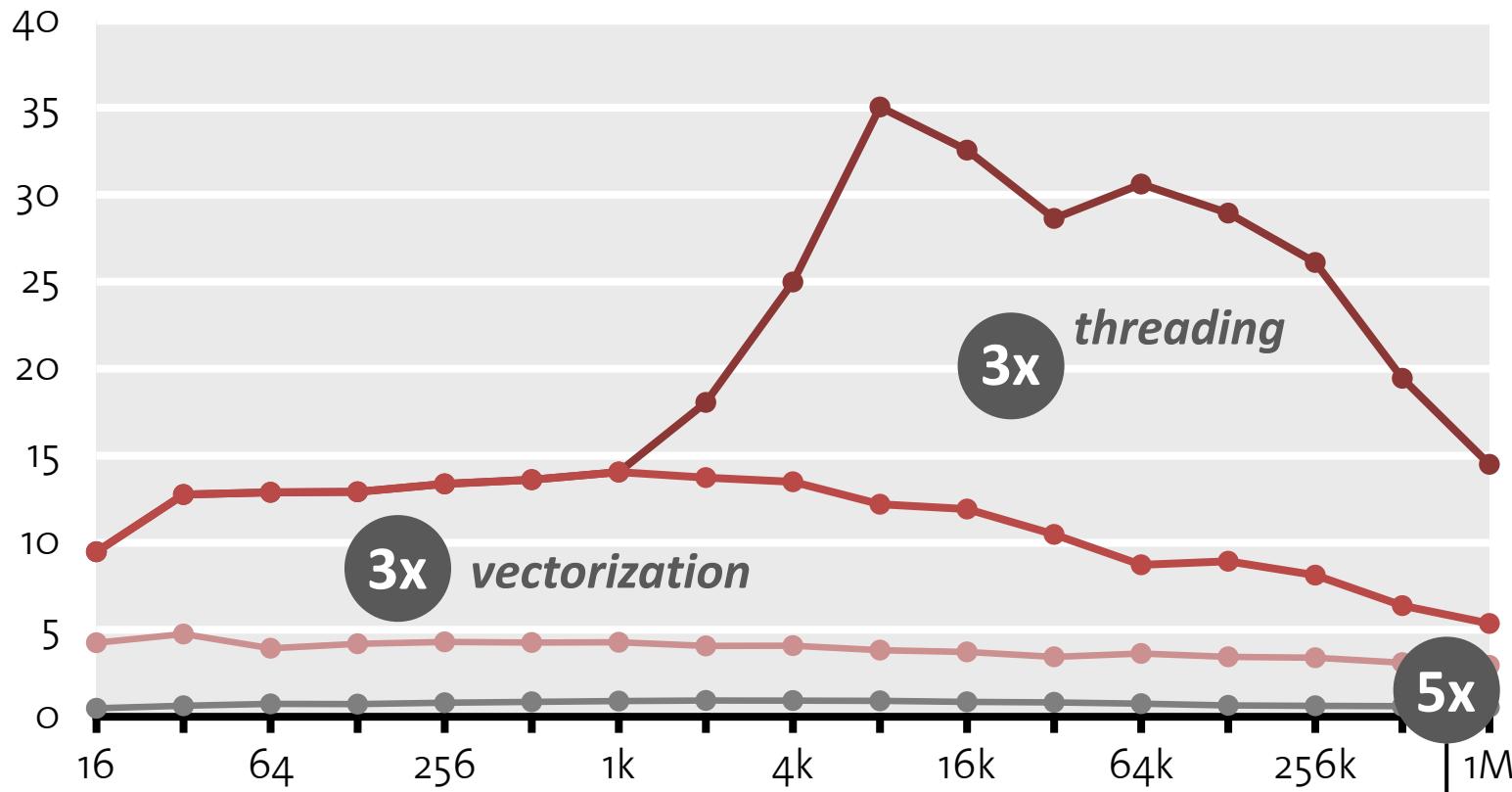


Why is that?

DFT: Analysis

DFT (single precision) on Intel Core i7 (4 cores, 2.66 GHz)

Performance [Gflop/s]

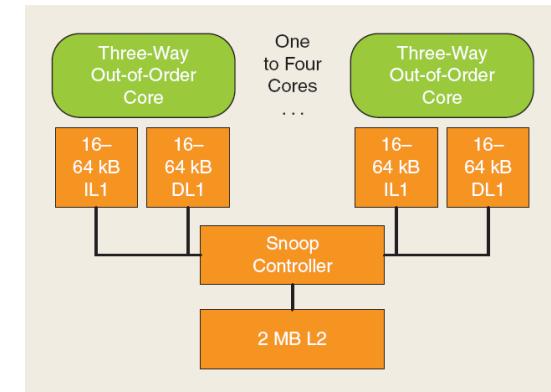


- Compiler doesn't do it
- Doing by hand: Very tough

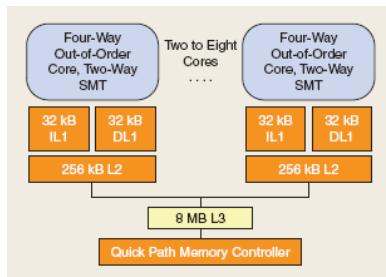
locality optimization

And There Is Not Only Intel ...

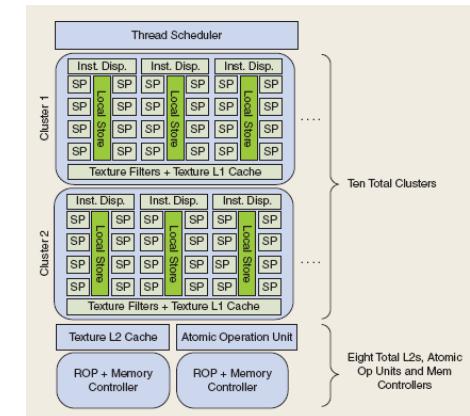
Arm Cortex A9



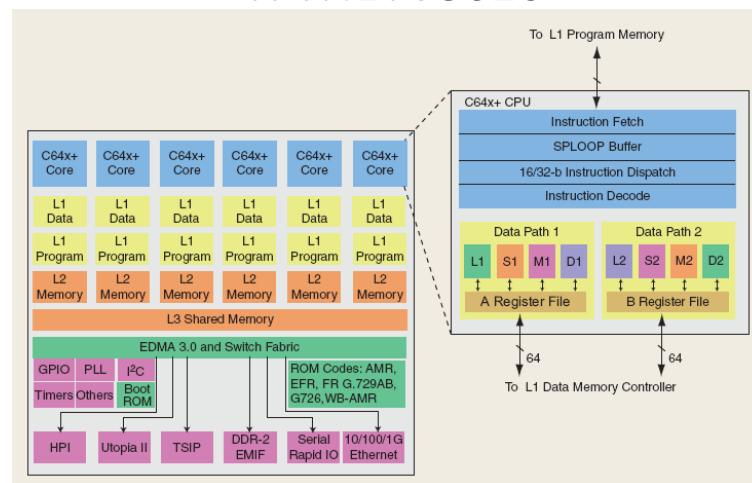
Core i7



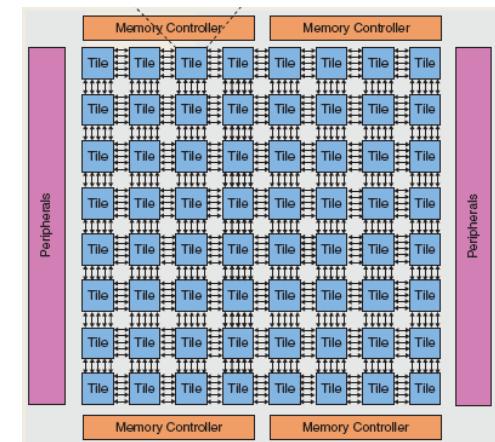
Nvidia G200



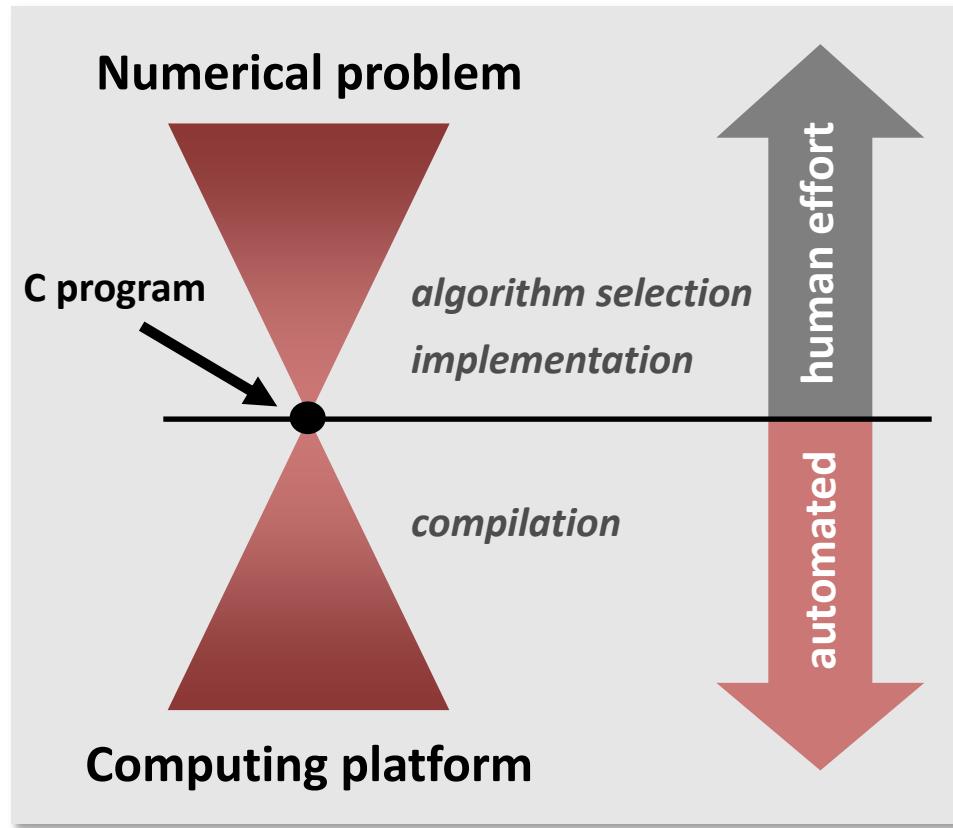
TI TNETV3020



Tilera Tile64



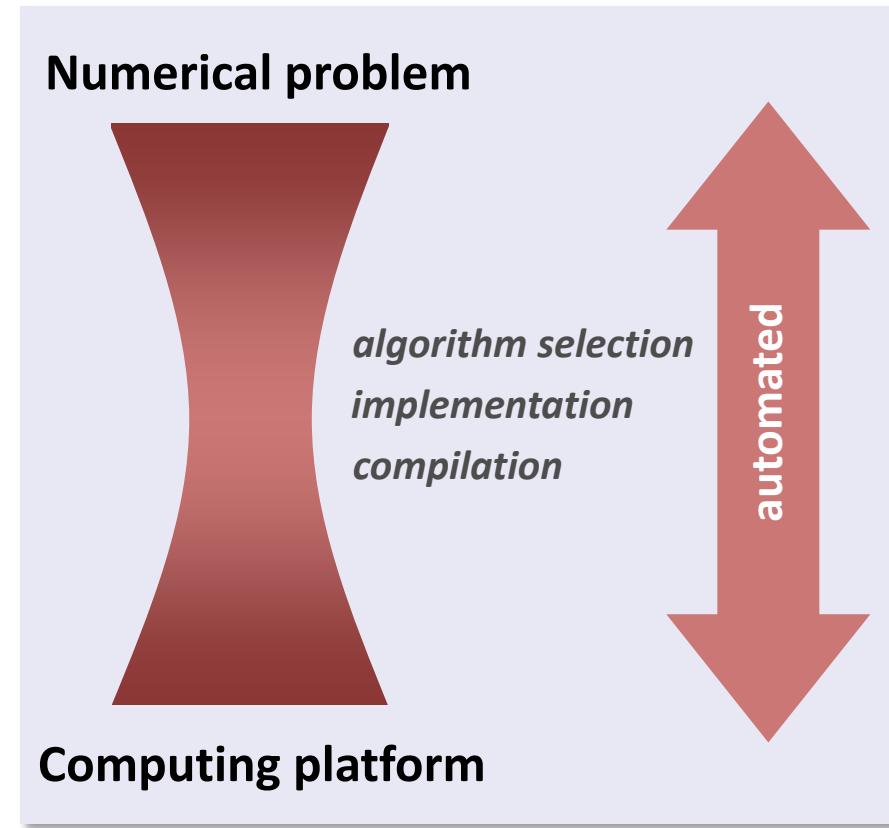
Current



C code a singularity:

- Compiler has no access to high level information
- No structural optimization
- No evaluation of choices

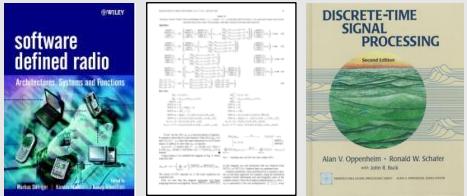
Future



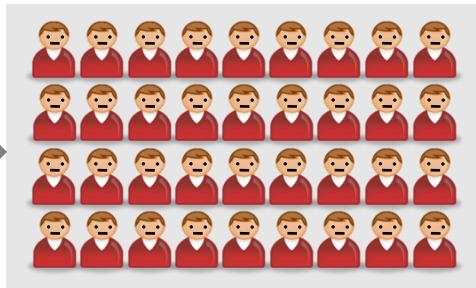
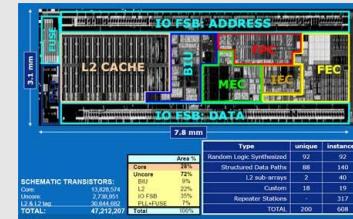
Challenge: conquer the high abstraction level for *complete automation*

Current Solution

Algorithm knowledge



Processor knowledge

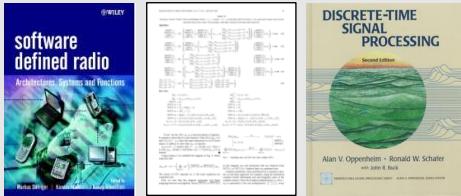


Optimal program

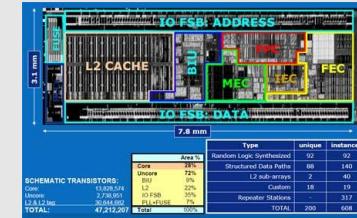
(Repeated for every processor)

Our Research: *The Computer Writes the Program*

Algorithm knowledge



Processor knowledge



Automation:
Spiral

Optimal program

(Regenerated for every processor)

“Computer Writes the Program”

Select transform

Transform

DCT2 ▾

transform

Size

2 ▾

number of samples

Pruning

Input

unpruned ▾

number of non-zero input samples

Output

unpruned ▾

number of non-zero output samples

Select implementation options

Data type

double precision ▾

data type

Scaling

-unscaled- ▾

output scaling

Generate Code

Reset



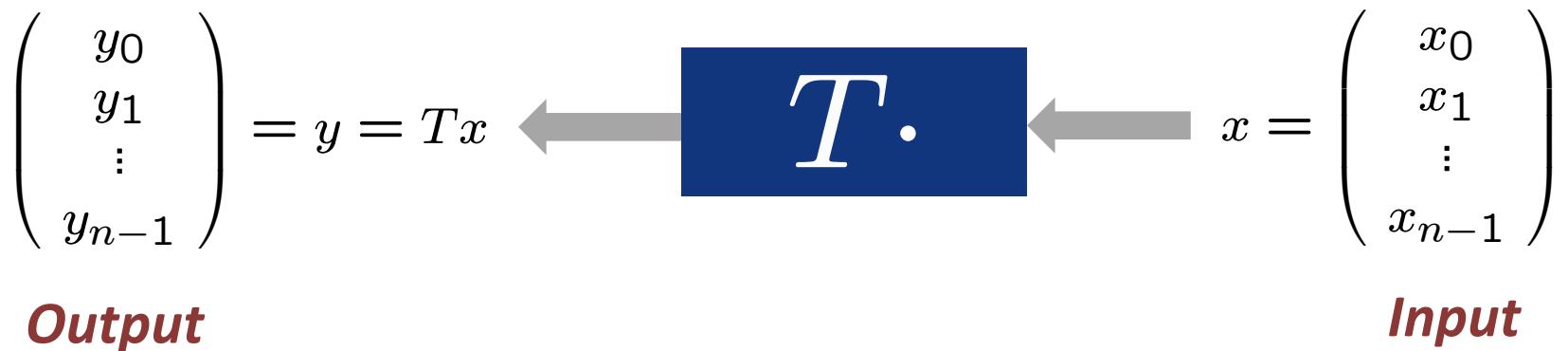
“click”

- Computer writes close to optimal code
- Vectorized, parallelized, etc.

Organization

- Spiral: Basic system
- Parallelism
- General input size
- Results
- Final remarks

Linear Transforms



Example: $T = \text{DFT}_n = [e^{-2k\ell\pi i/n}]_{0 \leq k, \ell < n}$

Algorithms: Example FFT, n = 4

Fast Fourier transform (FFT)

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} x = \begin{bmatrix} 1 & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & 1 \\ 1 & \cdot & -1 & \cdot \\ \cdot & 1 & \cdot & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & i \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdot & \cdot \\ 1 & -1 & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix} x$$

Representation using matrix algebra

$$\text{DFT}_4 = (\text{DFT}_2 \otimes \text{I}_2) \text{T}_2^4 (\text{I}_2 \otimes \text{DFT}_2) \text{L}_2^4$$

- *SPL (Signal processing language): Mathematical, declarative, point-free*
- Divide-and-conquer algorithms = breakdown rules in SPL

Decomposition Rules (>200 for >40 Transforms)

$$\begin{aligned}
& \text{DFT}_n \rightarrow P_{k/2,2m}^\top \left(\text{DFT}_{2m} \oplus \left(I_{k/2-1} \quad i C_{2m} \text{rDFT}_{2m}(i/k) \right) \right) \left(\text{RDFT}'_k \quad I_m \right), \quad k \text{ even}, \\
& \begin{vmatrix} \text{RDFT}_n \\ \text{RDFT}'_n \\ \text{DHT}_n \\ \text{DHT}'_n \end{vmatrix} \rightarrow (P_{k/2,m}^\top \quad I_2) \left(\begin{vmatrix} \text{RDFT}_{2m} \\ \text{RDFT}'_{2m} \\ \text{DHT}_{2m} \\ \text{DHT}'_{2m} \end{vmatrix} \oplus \begin{pmatrix} I_{k/2-1} & i D_{2m} \\ \text{rDFT}_{2m}(i/k) & \text{rDFT}_{2m}(i/k) \\ \text{rDHT}_{2m}(i/k) & \text{rDHT}_{2m}(i/k) \end{pmatrix} \right) \left(\begin{vmatrix} \text{RDFT}'_k \\ \text{RDFT}'_k \\ \text{DHT}'_k \\ \text{DHT}'_k \end{vmatrix} \quad I_m \right), \quad k \text{ even}, \\
& \begin{vmatrix} \text{rDFT}_{2n}(u) \\ \text{rDHT}_{2n}(u) \end{vmatrix} \rightarrow L_m^{2n} \left(I_k \quad i \begin{vmatrix} \text{rDFT}_{2m}((i+u)/k) \\ \text{rDHT}_{2m}((i+u)/k) \end{vmatrix} \right) \left(\begin{vmatrix} \text{rDFT}_{2k}(u) \\ \text{rDHT}_{2k}(u) \end{vmatrix} \quad I_m \right), \\
& \text{RDFT-3}_n \rightarrow (Q_{k/2,m}^\top \quad I_2) (I_k \quad i \text{rDFT}_{2m})(i+1/2)/k)) (\text{RDFT-3}_k \quad I_m), \quad k \text{ even}, \\
& \text{DCT-2}_n \rightarrow P_{k/2,2m}^\top \left(\text{DCT-2}_{2m} K_2^{2m} \oplus \left(I_{k/2-1} \quad N_{2m} \text{RDFT-3}_{2m}^\top \right) \right) B_n(L_{k/2}^{n/2} \quad I_2) (I_m \quad \text{RDFT}'_k) Q_{m/2,k},
\end{aligned}$$

$$\text{DCT-3}_n \rightarrow \text{DCT-2}_n^\top,$$

**Decomposition rules = Algorithm knowledge in Spiral
(from ≈ 100 publications)**

$$\begin{aligned}
& \text{DFT}_n \rightarrow \frac{C}{(2\pi)^{n/2}} \left(I_{n/2} \quad \text{N}_n \text{RDFT-3}_{2m}^\top \right) \text{U}'_m(L_{n/2}^{n/2} \quad I_m \quad \text{RDFT-3}_m) Q_n, \quad n = km \\
& \text{DFT}_n \rightarrow P_n(\text{DFT}_k \quad \text{DFT}_m) Q_n, \quad n = km, \quad \gcd(k, m) = 1
\end{aligned}$$

$$(\text{I}_1 \oplus \text{DFT}_{p-1}) R_p, \quad p \text{ prime}$$

$$\begin{aligned}
& \text{DCT-3}_n \rightarrow (\text{I}_m \oplus \text{J}_m) \text{U}_m^n (\text{DCT-3}_m(1/4) \oplus \text{DCT-3}_m(3/4)) \\
& \cdot (\mathbb{F}_2 \quad \text{I}_m) \begin{bmatrix} \text{I}_m & 0 \oplus -\text{J}_{m-1} \\ 0 & \frac{1}{\sqrt{2}}(\text{I}_1 \oplus 2\text{I}_m) \end{bmatrix}, \quad n = 2m
\end{aligned}$$

$$\text{DCT-4}_n \rightarrow S_n \text{DCT-2}_n \text{diag}_{0 \leq k < n} (1/(2 \cos((2k+1)\pi/4n)))$$

$$\text{IMDCT}_{2m} \rightarrow (\text{J}_m \oplus \text{I}_m \oplus \text{I}_m \oplus \text{J}_m) \left(\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \text{I}_m \right) \oplus \left(\begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad \text{I}_m \right) \right) \text{J}_{2m} \text{DCT-4}_{2m}$$

$$\text{WHT}_{2^k} \rightarrow \prod_{i=1}^t (\text{I}_{2^{k_1+\dots+k_{i-1}}} \quad \text{WHT}_{2^{k_i}} \quad \text{I}_{2^{k_i+1+\dots+k_t}}), \quad k = k_1 + \dots + k_t$$

$$\text{DFT}_2 \rightarrow \mathbb{F}_2$$

$$\text{DCT-2}_2 \rightarrow \text{diag}(1, 1/\sqrt{2}) \mathbb{F}_2$$

$$\text{DCT-4}_2 \rightarrow \text{J}_2 \text{R}_{13\pi/8}$$

Combining these rules yields many algorithms for every given transform

SPL to Code

SPL S Pseudo code for $y = Sx$

$A_n B_n$ <code for: $t = Bx$ >
 <code for: $y = At$ >

$I_m \otimes A_n$ for ($i=0$; $i < m$; $i++$)
 <code for:
 $y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1])$ >

$A_m \otimes I_n$ for ($i=0$; $i < n$; $i++$)
 <code for:
 $y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n])$ >

D_n for ($i=0$; $i < n$; $i++$)
 $y[i] = D[i]*x[i];$

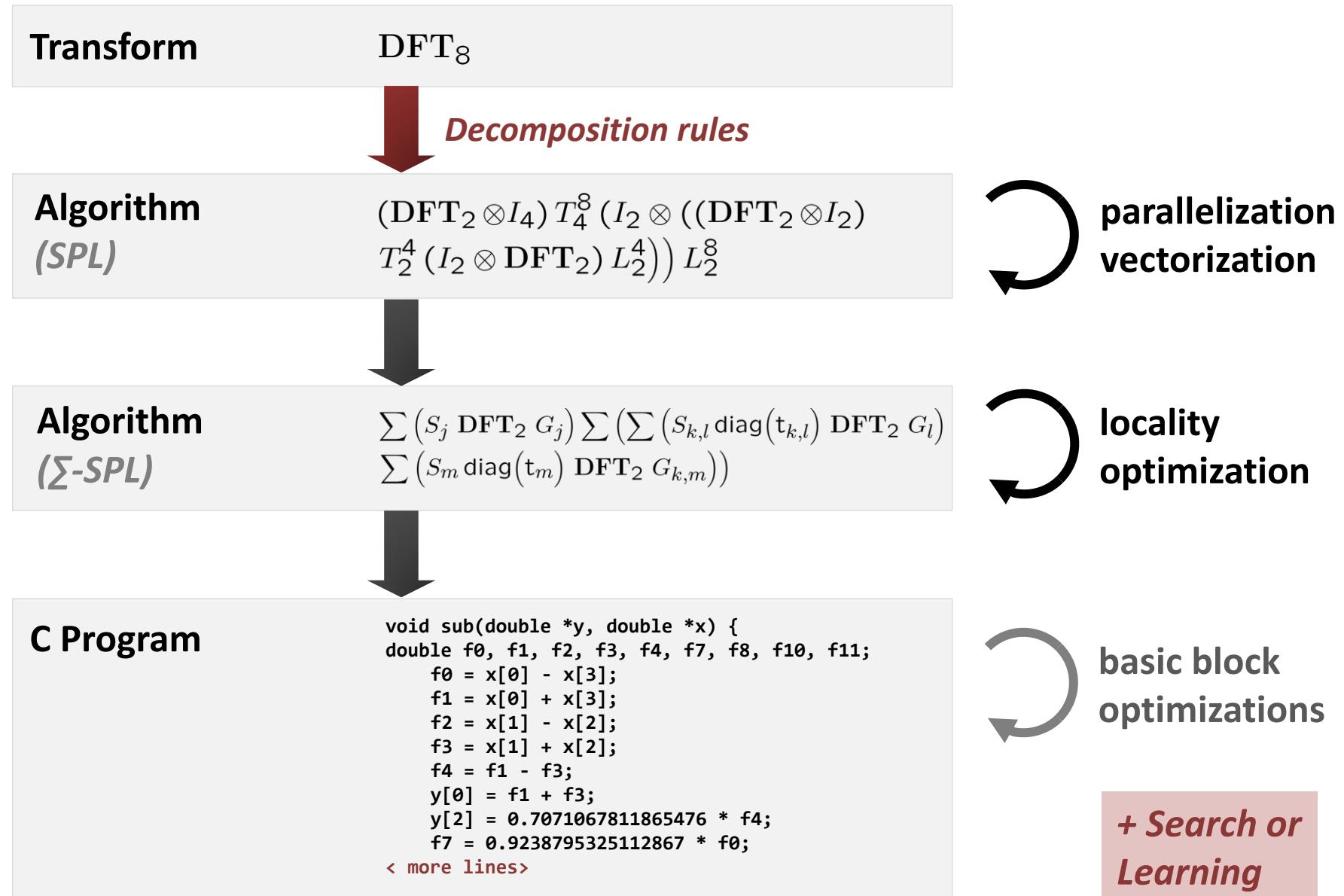
L_k^{km} for ($i=0$; $i < k$; $i++$)
 for ($j=0$; $j < m$; $j++$)
 $y[i*m+j] = x[j*k+i];$

F_2 $y[0] = x[0] + x[1];$
 $y[1] = x[0] - x[1];$

$$I_m \otimes A_n = \begin{bmatrix} A_n & & \\ & \ddots & \\ & & A_n \end{bmatrix}$$

Correct code: easy fast code: very difficult

Program Generation in Spiral



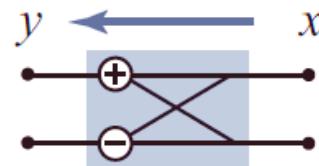
Organization

- Spiral: Basic system
- **Parallelism**
- General input size
- Results
- Final remarks

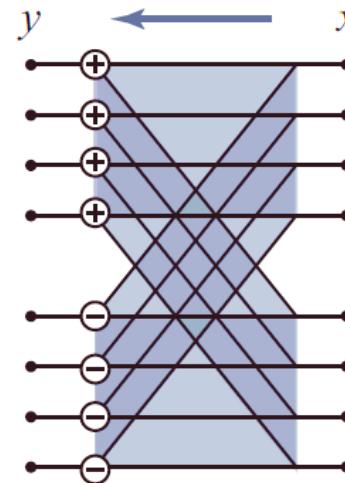
Example: Vectorization in Spiral

- Relationship SPL expressions \leftrightarrow vectorization?

$$y = \text{DFT}_2 x$$



$$y = (\text{DFT}_2 \quad I_4)x$$



one addition
one subtraction

one (4-way) vector addition
one (4-way) vector subtraction

Step 1: Identify “Good” Vector Constructs

- Vector length: ν
- Good (= easily vectorizable) SPL constructs:

$A \quad I_\nu$

$L_\nu^{\nu^2}, L_2^{2\nu}, L_\nu^{2\nu}$ *base cases*

SPL expressions recursively built from those

- **Idea:** Convert a given SPL expression into a “good” SPL expression through rewriting (structural manipulation)

Step 2: Find Manipulation Rules

$$\mathsf{L}_n^{n\nu} \rightarrow \left(\mathbf{I}_{n/\nu} \quad \mathsf{L}_\nu^{\nu^2} \right) \left(\mathsf{L}_{n/\nu}^n \quad \mathbf{I}_\nu \right)$$

$$\mathsf{L}_\nu^{n\nu} \rightarrow \left(\mathsf{L}_\nu^n \quad \mathbf{I}_\nu \right) \left(\mathbf{I}_{n/\nu} \quad \mathsf{L}_\nu^{\nu^2} \right)$$

$$\mathsf{L}_m^{mn} \rightarrow \left(\mathsf{L}_m^{mn/\nu} \quad \mathbf{I}_\nu \right) \left(\mathbf{I}_{mn/\nu^2} \quad \mathsf{L}_\nu^{\nu^2} \right) \left(\mathbf{I}_{n/\nu} \quad \mathsf{L}_{m/\nu}^m \quad \mathbf{I}_\nu \right)$$

$$\mathbf{I}_l \quad \mathsf{L}_n^{kmn} \quad \mathbf{I}_r \rightarrow \left\{ \mathbf{I}_l \quad \mathsf{L}_n^{kn} \quad \mathbf{I}_{mr} \right\} \left\{ \mathbf{I}_{kl} \quad \mathsf{L}_n^{mn} \quad \mathbf{I}_r \right\}$$

$$\mathbf{I}_l \quad \mathsf{L}_n^{kmn} \quad \mathbf{I}_r \rightarrow \left\{ \mathbf{I}_l \quad \mathsf{L}_{kn}^{kmn} \quad \mathbf{I}_r \right\} \left\{ \mathbf{I}_l \quad \mathsf{L}_{mn}^{kmn} \quad \mathbf{I}_r \right\}$$

$$\mathbf{I}_l \quad \mathsf{L}_{km}^{kmn} \quad \mathbf{I}_r \rightarrow \left\{ \mathbf{I}_{kl} \quad \mathsf{L}_m^{mn} \quad \mathbf{I}_r \right\} \left\{ \mathbf{I}_l \quad \mathsf{L}_k^{kn} \quad \mathbf{I}_{mr} \right\}$$

$$\mathbf{I}_l \quad \mathsf{L}_k^{kmn} \quad \mathbf{I}_r \rightarrow \left\{ \mathbf{I}_l \quad \mathsf{L}_k^{kmn} \quad \mathbf{I}_r \right\} \left\{ \mathbf{I}_l \quad \mathsf{L}_n^{kn} \quad \mathbf{I}_r \right\}$$

Manipulation rules = Processor knowledge in Spiral

$$\left(\mathbf{I}_m \quad A^{n \times n} \right) \mathsf{L}_m^{mn} \rightarrow \left(\mathbf{I}_{m/\nu} \quad \mathsf{L}_\nu^{n\nu} \left(A^{n \times n} \quad \mathbf{I}_\nu \right) \right) \left(\mathsf{L}_{m/\nu}^{mn/\nu} \quad \mathbf{I}_\nu \right)$$

$$\mathsf{L}_n^{mn} \left(\mathbf{I}_m \quad A^{n \times n} \right) \rightarrow \left(\mathsf{L}_n^{mn/\nu} \quad \mathbf{I}_\nu \right) \left(\mathbf{I}_{m/\nu} \quad \left(A^{n \times n} \quad \mathbf{I}_\nu \right) \mathsf{L}_n^{n\nu} \right)$$

$$\left(\mathbf{I}_k \quad \left(\mathbf{I}_m \quad A^{n \times n} \right) \mathsf{L}_m^{mn} \right) \mathsf{L}_k^{kmn} \rightarrow \left(\mathsf{L}_k^{km} \quad \mathbf{I}_n \right) \left(\mathbf{I}_m \quad \left(\mathbf{I}_k \quad A^{n \times n} \right) \mathsf{L}_k^{kn} \right) \left(\mathsf{L}_m^{mn} \quad \mathbf{I}_k \right)$$

$$\mathsf{L}_{mn}^{kmn} \left(\mathbf{I}_k \quad \mathsf{L}_n^{mn} \left(\mathbf{I}_m \quad A^{n \times n} \right) \right) \rightarrow \left(\mathsf{L}_n^{mn} \quad \mathbf{I}_k \right) \left(\mathbf{I}_m \quad \mathsf{L}_n^{kn} \left(\mathbf{I}_k \quad A^{n \times n} \right) \right) \left(\mathsf{L}_m^{km} \quad \mathbf{I}_n \right)$$

$$\overline{AB} \rightarrow \overline{A}\overline{B}$$

$$\overline{A^{m \times m}} \quad \mathbf{I}_\nu \rightarrow \left(\mathbf{I}_m \quad \mathsf{L}_\nu^{2\nu} \right) \left(\overline{A^{m \times m}} \quad \mathbf{I}_\nu \right) \left(\mathbf{I}_m \quad \mathsf{L}_2^{2\nu} \right)$$

$$\overline{\mathbf{I}_m \quad A^{n \times n}} \rightarrow \mathbf{I}_m \quad \overline{A^{n \times n}}$$

$$\overline{D} \rightarrow \left(\mathbf{I}_{n/\nu} \quad \mathsf{L}_\nu^{2\nu} \right) \vec{D} \left(\mathbf{I}_{n/\nu} \quad \mathsf{L}_2^{2\nu} \right)$$

$$\overline{P} \rightarrow P \quad \mathbf{I}_2$$

Example

$$\underbrace{\mathbf{DFT}_{mn}}_{\text{vec}(\nu)} \rightarrow \underbrace{(\mathbf{DFT}_m \quad \mathbf{I}_n) \mathbf{T}_n^{mn} (\mathbf{I}_m \quad \mathbf{DFT}_n) \mathbf{L}_m^{mn}}_{\text{vec}(\nu)}$$

...

...

...

$$\rightarrow \underbrace{\begin{pmatrix} \mathbf{I}_{\frac{mn}{\nu}} & \mathbf{L}_\nu^{2\nu} \end{pmatrix} \begin{pmatrix} \overline{\mathbf{DFT}_m} & \overline{\mathbf{I}_{\frac{n}{\nu}}} & \mathbf{I}_\nu \end{pmatrix} \overline{\mathbf{T}}_n'^{mn}}_{\begin{pmatrix} \mathbf{I}_{\frac{m}{\nu}} & (\mathbf{L}_\nu^{2n} & \mathbf{I}_\nu) \end{pmatrix} \begin{pmatrix} \mathbf{I}_{\frac{2n}{\nu}} & \mathbf{L}_\nu^{\nu^2} \end{pmatrix} \begin{pmatrix} \mathbf{I}_{\frac{n}{\nu}} & \mathbf{L}_2^{2\nu} & \mathbf{I}_\nu \end{pmatrix} \begin{pmatrix} \overline{\mathbf{DFT}}_n & \mathbf{I}_\nu \end{pmatrix}} \underbrace{\begin{pmatrix} \mathbf{L}_{\frac{m}{\nu}}^{\frac{mn}{\nu}} & \mathbf{L}_2^{2\nu} \end{pmatrix}}$$

vectorized arithmetic

vectorized data accesses

Automatically Generate Base Case Library

- **Goal:** Given instruction set, generate base cases

$$\nu = 4 : \{ L_2^4, I_2 \quad L_2^4, L_2^4 \quad I_2, L_2^8, L_4^8 \}$$

- **Idea:** Instructions as matrices + search

```
y = _mm_unpacklo_ps(x0, x1);
```

```
y = _mm_shuffle_ps(x0, x1, _MM_SHUFFLE(1,2,1,2));
```

```
y = _mm_shuffle_ps(x0, x1, _MM_SHUFFLE(3,4,3,4));
```

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \vec{x}_0 \\ \vec{x}_1 \end{bmatrix}$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \vec{x}_0 \\ \vec{x}_1 \end{bmatrix}$$

$$y = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{x}_0 \\ \vec{x}_1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

```
y0 = _mm_shuffle_ps(x0, x1,  
                     _MM_SHUFFLE(1,2,1,2));  
y1 = _mm_shuffle_ps(x0, x1,  
                     _MM_SHUFFLE(3,4,3,4));
```



Same Approach for Different Paradigms

Threading:

$$\begin{aligned}
 \underbrace{\text{DFT}_{mn}}_{\text{smp}(p,\mu)} &\rightarrow \underbrace{\left((\text{DFT}_m \otimes \text{I}_n) \text{T}_n^{mn} (\text{I}_m \otimes \text{DFT}_n) \text{L}_m^{mn} \right)}_{\text{smp}(p,\mu)} \\
 &\dots \\
 &\rightarrow \underbrace{\left(\text{DFT}_m \otimes \text{I}_n \right)}_{\text{smp}(p,\mu)} \underbrace{\text{T}_n^{mn}}_{\text{smp}(p,\mu)} \underbrace{\left(\text{I}_m \otimes \text{DFT}_n \right)}_{\text{smp}(p,\mu)} \underbrace{\text{L}_m^{mn}}_{\text{smp}(p,\mu)} \\
 &\dots \\
 &\rightarrow \left((\text{L}_m^{mp} \otimes \text{I}_{n/p\mu}) \otimes_\mu \text{I}_\mu \right) \left(\text{I}_p \otimes \| (\text{DFT}_m \otimes \text{I}_{n/p}) \right) \left((\text{L}_p^{mp} \otimes \text{I}_{n/p\mu}) \otimes_\mu \text{I}_\mu \right) \\
 &\quad \left(\bigoplus_{i=0}^{p-1} \parallel \text{T}_n^{mn,i} \right) \left(\text{I}_p \otimes \| (\text{I}_{m/p} \otimes \text{DFT}_n) \right) \left(\text{I}_p \otimes \| \text{L}_{m/p}^{mn/p} \right) \left((\text{L}_p^{pn} \otimes \text{I}_{m/p\mu}) \otimes_\mu \text{I}_\mu \right)
 \end{aligned}$$

Vectorization:

$$\begin{aligned}
 \underbrace{\overline{\text{DFT}_{mn}}}_{\text{vec}(\nu)} &\rightarrow \underbrace{\left((\text{DFT}_m \otimes \text{I}_n) \text{T}_n^{mn} (\text{I}_m \otimes \text{DFT}_n) \text{L}_m^{mn} \right)}_{\text{vec}(\nu)} \\
 &\dots \\
 &\rightarrow \underbrace{\left(\overline{\text{DFT}_m \otimes \text{I}_n} \right)^\nu}_{\text{vec}(\nu)} \underbrace{\left(\overline{\text{T}_n^{mn}} \right)^\nu}_{\text{vec}(\nu)} \underbrace{\left(\overline{\text{I}_m \otimes \text{DFT}_n} \right) \overline{\text{L}_m^{mn}}^\nu}_{\text{vec}(\nu)} \\
 &\dots \\
 &\rightarrow \left(\text{I}_{mn/\nu} \otimes \underbrace{\text{L}_\nu^{2\nu}}_{\text{sse}} \right) \left(\overline{\text{DFT}_m \otimes \text{I}_{n/\nu}} \vec{\otimes} \text{I}_\nu \right) \left(\overline{\text{T}_n^{mn}} \right)^\nu \\
 &\quad \left(\text{I}_{m/\nu} \otimes (\overline{\text{L}_\nu^{2\nu}} \vec{\otimes} \text{I}_\nu) (\text{I}_{n/\nu} \otimes (\text{L}_\nu^{2\nu} \vec{\otimes} \text{I}_\nu)) (\text{I}_2 \otimes \underbrace{\text{L}_\nu^{\nu^2}}_{\text{sse}}) (\text{L}_2^{2\nu} \vec{\otimes} \text{I}_\nu) \right) (\overline{\text{DFT}_n} \vec{\otimes} \text{I}_\nu) \\
 &\quad \left((\text{L}_m^{mn} \otimes \text{I}_2) \vec{\otimes} \text{I}_\nu \right) (\text{I}_{mn/\nu} \otimes \underbrace{\text{L}_2^{2\nu}}_{\text{sse}})
 \end{aligned}$$

GPUs:

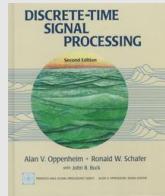
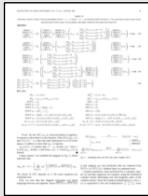
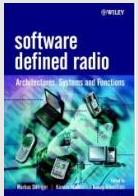
$$\begin{aligned}
 \underbrace{\left(\text{DFT}_{r^k} \right)}_{\text{gpu}(t,c)} &\rightarrow \underbrace{\left(\prod_{i=0}^{k-1} \text{L}_r^{r^k} \left(\text{I}_{r^{k-1}} \otimes \text{DFT}_r \right) \left(\text{L}_{r^{k-i-1}}^{r^k} (\text{I}_{r^i} \otimes \text{T}_{r^{k-i-1}}^{r^{k-i}}) \underbrace{\text{L}_{r^{i+1}}^{r^k}}_{\text{vec}(c)} \right) \right) \text{R}_r^{r^k}}_{\text{gpu}(t,c)} \\
 &\dots \\
 &\rightarrow \left(\prod_{i=0}^{k-1} (\text{L}_r^{r^n/2} \vec{\otimes} \text{I}_2) \left(\text{I}_{r^{n-1}/2} \otimes \times \underbrace{(\text{DFT}_r \vec{\otimes} \text{I}_2) \text{L}_r^{2r}}_{\text{shd}(t,c)} \right) \text{T}_i \right) \\
 &\quad (\text{L}_r^{r^n/2} \vec{\otimes} \text{I}_2) (\text{I}_{r^{n-1}/2} \otimes \times \underbrace{\text{L}_r^{2r}}_{\text{shd}(t,c)}) (\text{R}_r^{r^{n-1}} \vec{\otimes} \text{I}_r)
 \end{aligned}$$

Verilog for FPGAs:

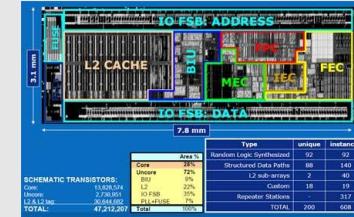
$$\begin{aligned}
 \underbrace{\left(\text{DFT}_{r^k} \right)}_{\text{stream}(r^s)} &\rightarrow \left[\prod_{i=0}^{k-1} \text{L}_r^{r^k} \left(\text{I}_{r^{k-1}} \otimes \text{DFT}_r \right) \left(\text{L}_{r^{k-i-1}}^{r^k} (\text{I}_{r^i} \otimes \text{T}_{r^{k-i-1}}^{r^{k-i}}) \text{L}_{r^{i+1}}^{r^k} \right) \right] \text{R}_r^{r^k} \\
 &\dots \\
 &\rightarrow \left[\prod_{i=0}^{k-1} \underbrace{\text{L}_r^{r^k}}_{\text{stream}(r^s)} \underbrace{\left(\text{I}_{r^{k-1}} \otimes \text{DFT}_r \right)}_{\text{stream}(r^s)} \underbrace{\left(\text{L}_{r^{k-i-1}}^{r^k} (\text{I}_{r^i} \otimes \text{T}_{r^{k-i-1}}^{r^{k-i}}) \text{L}_{r^{i+1}}^{r^k} \right)}_{\text{stream}(r^s)} \right] \underbrace{\text{R}_r^{r^k}}_{\text{stream}(r^s)} \\
 &\dots \\
 &\rightarrow \left[\prod_{i=0}^{k-1} \underbrace{\text{L}_r^{r^k}}_{\text{stream}(r^s)} \left(\text{I}_{r^{k-s-1}} \otimes_s (\text{I}_{r^{s-1}} \otimes \text{DFT}_r) \right) \underbrace{\text{T}_i'}_{\text{stream}(r^s)} \right] \underbrace{\text{R}_r^{r^k}}_{\text{stream}(r^s)}
 \end{aligned}$$

- Rigorous, correct by construction
- Overcomes compiler limitations

Algorithm knowledge



Processor knowledge



Automation:
Spiral

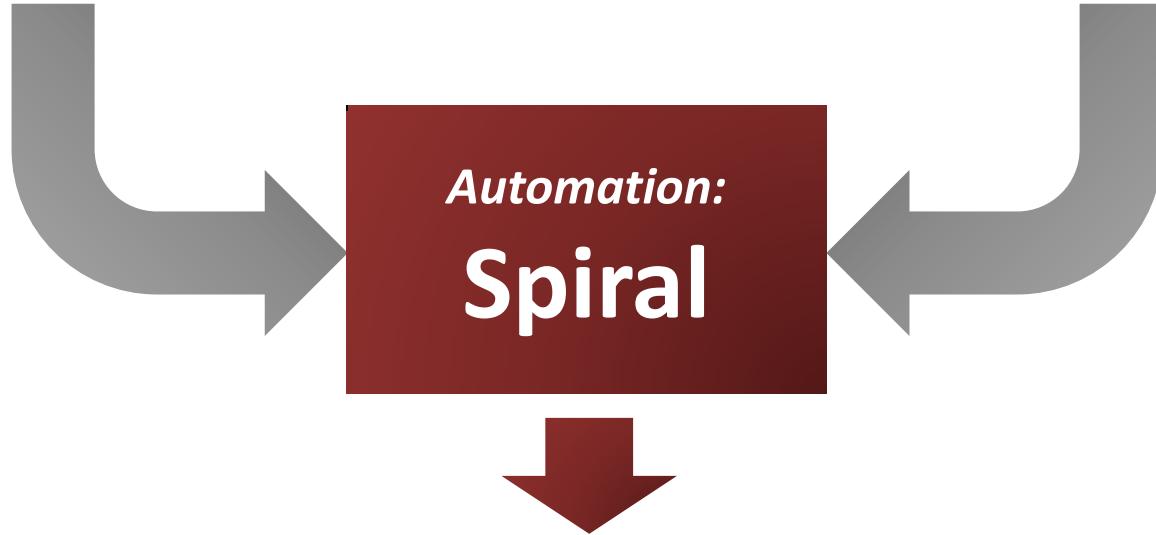
Optimal program
(Regenerated for every processor)

Decomposition rules

$$\begin{aligned}\text{DFT}_n &\rightarrow P_{k/2,2m}^\top \left(\text{DFT}_{2m} \oplus \left(I_{k/2-1} \otimes_i C_{2m} \mathbf{rDFT}_{2m}(i/k) \right) \right) \left(\mathbf{RDFT}'_k \otimes I_m \right) \\ \left| \mathbf{rDFT}_{2n}(u) \right| &\rightarrow L_m^{2n} \left(I_k \otimes_i \left| \mathbf{rDFT}_{2m}((i+u)/k) \right| \right) \left(\left| \mathbf{rDFT}_{2k}(u) \right| \otimes I_m \right) \\ \mathbf{RDFT-3}_n &\rightarrow (Q_{k/2,m}^\top \otimes I_2) (I_k \otimes_i \mathbf{rDFT}_{2m}(i+1/2)/k) (\mathbf{RDFT-3}_k \otimes I_m)\end{aligned}$$

Manipulation rules

$$\begin{aligned}\underbrace{A_m \otimes \mathbf{I}_n}_{\text{smp}(p,\mu)} &\rightarrow \left(\mathbf{L}_m^{mp} \otimes \mathbf{I}_{n/p} \right) \left(\mathbf{I}_p \otimes (A_m \otimes \mathbf{I}_{n/p}) \right) \left(\mathbf{L}_p^{mp} \otimes \mathbf{I}_{n/p} \right) \\ \underbrace{\mathbf{I}_m \otimes A_n}_{\text{smp}(p,\mu)} &\rightarrow \mathbf{I}_p \otimes_{||} \left(\mathbf{I}_{m/p} \otimes A_n \right) \\ \underbrace{(P \otimes \mathbf{I}_n)}_{\text{smp}(p,\mu)} &\rightarrow (P \otimes \mathbf{I}_{n/\mu}) \overline{\otimes} \mathbf{I}_\mu\end{aligned}$$



Optimal program
(Regenerated for every processor)

Organization

- Spiral: Basic system
- Parallelism
- **General input size**
- Results
- Final remarks

Challenge: General Size Libraries

So far:

Code specialized to fixed input size

```
DFT_384(x, y) {  
    ...  
    for(i = ...) {  
        t[2i]    = x[2i] + x[2i+1]  
        t[2i+1] = x[2i] - x[2i+1]  
    }  
    ...  
}
```

- Algorithm fixed
- Nonrecursive code

Challenge:

Library for general input size

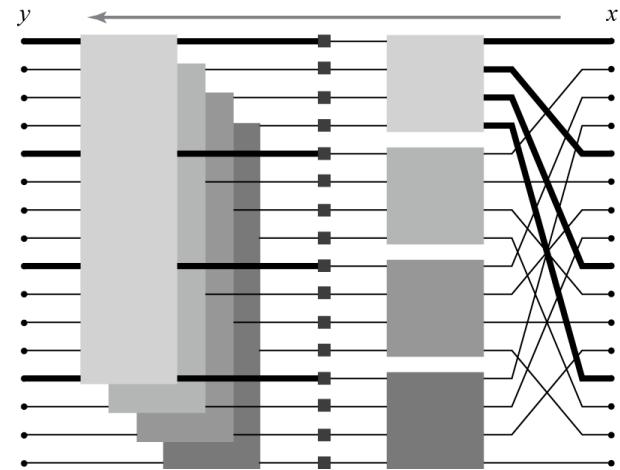
```
DFT(n, x, y) {  
    ...  
    for(i = ...) {  
        DFT_strided(m, x+mi, y+i, 1, k)  
    }  
    ...  
}
```

- Algorithm cannot be fixed
- Recursive code
- Creates many challenges

Challenge: Recursion Steps

■ Cooley-Tukey FFT

$$y = (\text{DFT}_k \otimes I_m) T_m^{km} (I_k \otimes \text{DFT}_m) L_k^{km} x$$



■ Implementation that increases locality (e.g., FFTW 2.x)

```
void DFT(int n, cpx *y, cpx *x) {
    int k = choose_dft_radix(n);

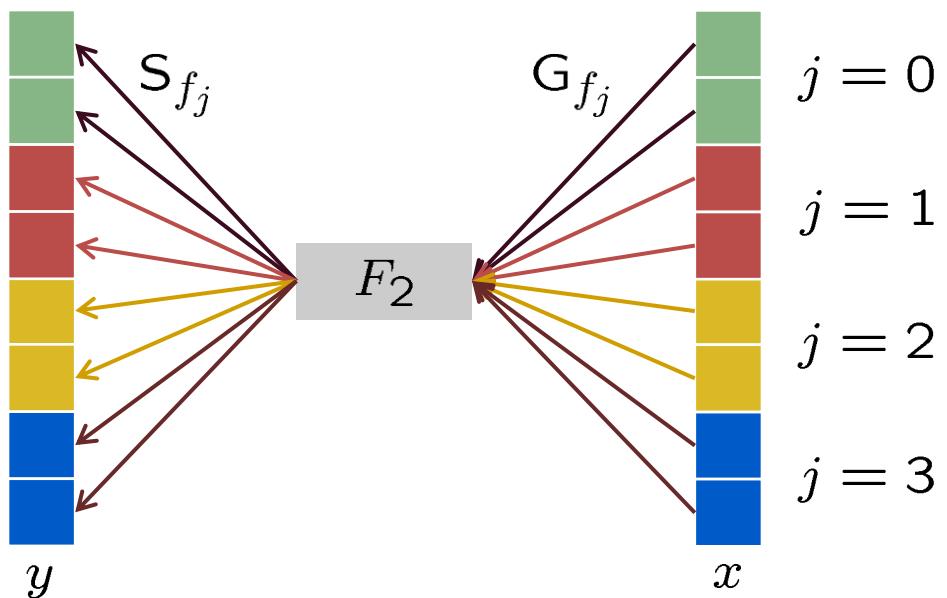
    for (int i=0; i < k; ++i)
        DFTrec(m, y + m*i, x + i, k, 1);
    for (int j=0; j < m; ++j)
        DFTscaled(k, y + j, t[j], m);
}
```

Σ -SPL : Basic Idea

- Four additional matrix constructs: Σ , G , S , Perm
 - Σ (sum) explicit loop
 - G_f (gather) load data with index mapping f
 - S_f (scatter) store data with index mapping f
 - Perm_f permute data with the index mapping f
- Σ -SPL formulas = matrix factorizations

Example: $y = (I_4 \otimes F_2)x \rightarrow y = \sum_{j=0}^3 S_{f_j} F_2 G_{f_j} x$

$$y = \begin{bmatrix} F_2 & & & \\ & F_2 & & \\ & & F_2 & \\ & & & F_2 \end{bmatrix} x$$



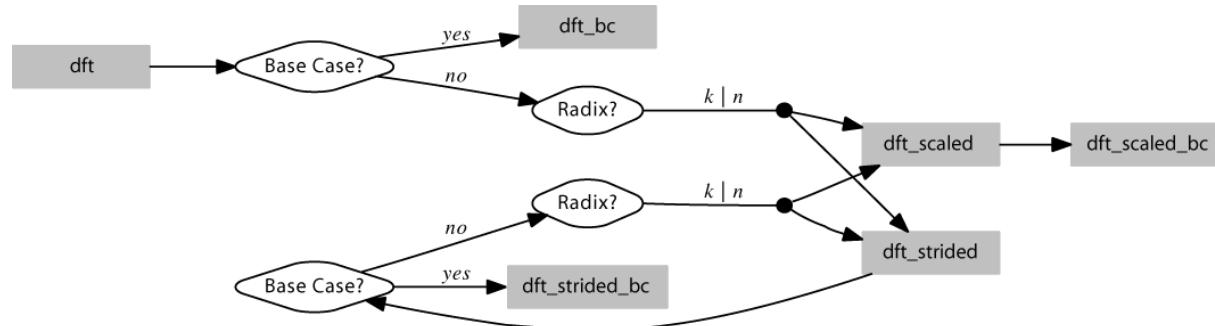
Find Recursion Step Closure

$$\begin{aligned} & \{\text{DFT}_n\} \\ & \downarrow \\ & (\{\text{DFT}_{n/k}\} \otimes I_k) T_k^n (I_{n/k} \otimes \{\text{DFT}_k\}) L_{n/k}^n \\ & \downarrow \\ & \left(\sum_{i=0}^{k-1} S_{h_{i,k}} \{\text{DFT}_{n/k}\} G_{h_{i,k}} \right) \text{diag}(f) \left(\sum_{j=0}^{n/k-1} S_{h_{jk,1}} \{\text{DFT}_k\} G_{h_{jk,1}} \right) \text{perm}(\ell_{n/k}^n) \\ & \downarrow \\ & \sum_{i=0}^{k-1} S_{h_{i,k}} \{\text{DFT}_{n/k}\} \text{diag}(f \circ h_{i,k}) G_{h_{i,k}} \sum_{j=0}^{n/k-1} S_{h_{jk,1}} \{\text{DFT}_k\} G_{h_{j,n/k}} \\ & \downarrow \\ & \sum_{i=0}^{k-1} \left\{ S_{h_{i,k}} \text{DFT}_{n/k} \text{diag}(f \circ h_{i,k}) G_{h_{i,k}} \right\} \sum_{j=0}^{n/k-1} \left\{ S_{h_{jk,1}} \text{DFT}_k G_{h_{j,n/k}} \right\} \end{aligned}$$

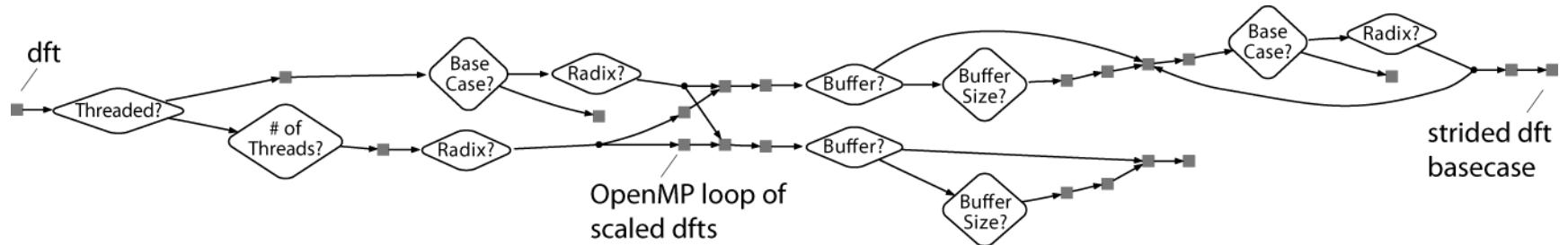
Repeat until closure

Recursion Step Closure: Examples

DFT: scalar code



DFT: full-fledged (vectorized and parallel code)



Summary: Complete Automation for Transforms

- **Memory hierarchy optimization**

Rewriting and search for algorithm selection

Rewriting for loop optimizations

- **Vectorization**

Rewriting

- **Parallelization**

Rewriting

fixed input size code

- **Derivation of library structure**

Rewriting

Other methods *general input size library*

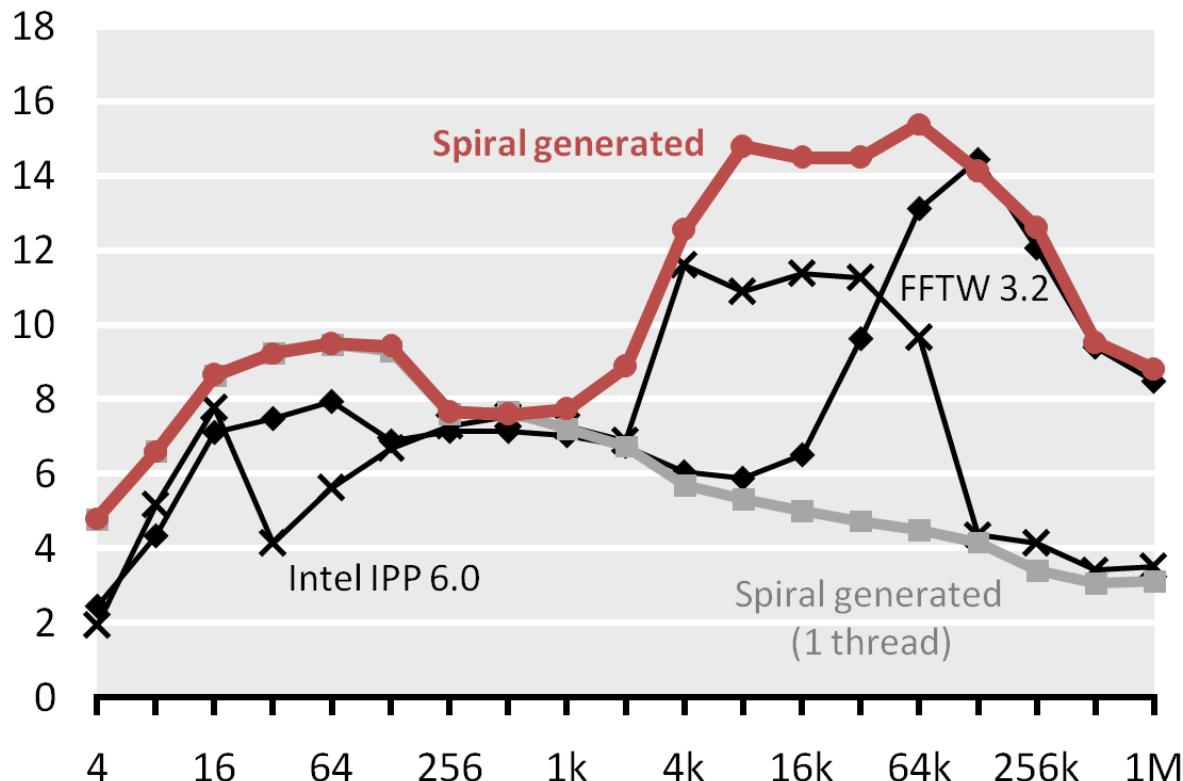
Organization

- Spiral: Basic system
- Parallelism
- General input size
- **Results**
- Final remarks

DFT on Intel Multicore

Complex DFT (Intel Core i7, 2.66 GHz, 4 cores)

Performance [Gflop/s] vs. input size



$$\text{DFT}_n \rightarrow (\text{DFT}_k \otimes I_m) T_m^n (I_k \otimes \text{DFT}_m) L_k^n$$

$$\text{DFT}_n \rightarrow P_{k/2,2m}^\top (\text{DFT}_{2m} \oplus (I_{k/2-1} \otimes_i C_{2m} \text{rDFT}_{2m}(i/k))) (\text{RDFT}_k \otimes I_m)$$

$$\text{RDFT}_n \rightarrow (P_{k/2,m}^\top \otimes I_2) (\text{RDFT}_{2m} \oplus (I_{k/2-1} \otimes_i D_{2m} \text{rDFT}_{2m}(i/k))) (\text{RDFT}_k \otimes I_m)$$

$$\text{rDFT}_{2n}(u) \rightarrow L_m^{2n} (I_k \otimes_i \text{rDFT}_{2m}((i+u)/k)) (\text{rDFT}_{2k}(u) \otimes I_m)$$

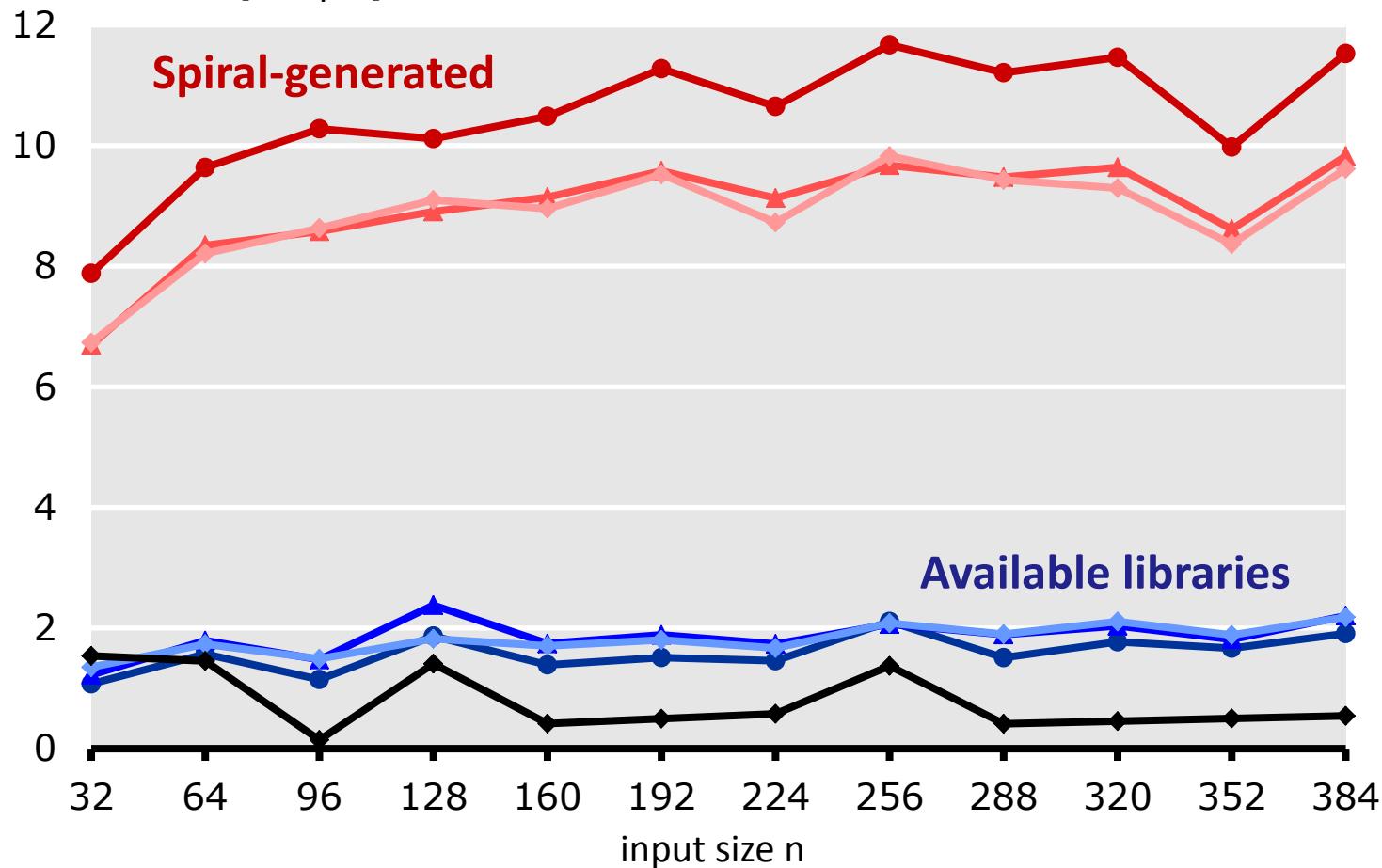
Spiral

5MB vectorized, threaded,
general-size, adaptive library

Often it Looks Like That

DCTs on 2.66 GHz Core2 (4-way SSSE3)

Performance [Gflop/s]



Computer generated Functions for Intel IPP 6.0

Intel® Integrated Performance Primitives (Intel® IPP) 6.0

The Intel® Integrated Performance Primitives (Intel® IPP) 6.0 library provides a wide range of computer-generated functions for signal processing, including DFT, RDFT, DCT2, DCT3, DCT4, DHT, and WHT transforms. These functions are available in various sizes (2-64) and precisions (single, double). The library supports scalar, SSE, and AVX data types. The code is generated for Intel IPP 6.0.

Key features of the library include:

- 3984 C functions
- 1M lines of code
- Transforms: DFT (fwd+inv), RDFT (fwd+inv), DCT2, DCT3, DCT4, DHT, WHT
- Sizes: 2-64 (DFT, RDFT, DHT); 2-powers (DCTs, WHT)
- Precision: single, double
- Data type: scalar, SSE, AVX (DFT, DCT), LRB (DFT)

Computer generated

Results: SpiralGen Inc.

Intel® Integrated Performance Primitives (Intel® IPP) 6.0

The Intel® Integrated Performance Primitives (Intel® IPP) 6.0 library provides a wide range of computer-generated functions for signal processing, including DFT, RDFT, DCT2, DCT3, DCT4, DHT, and WHT transforms. These functions are available in various sizes (2-64) and precisions (single, double). The library supports scalar, SSE, and AVX data types. The code is generated for Intel IPP 6.0.

Key features of the library include:

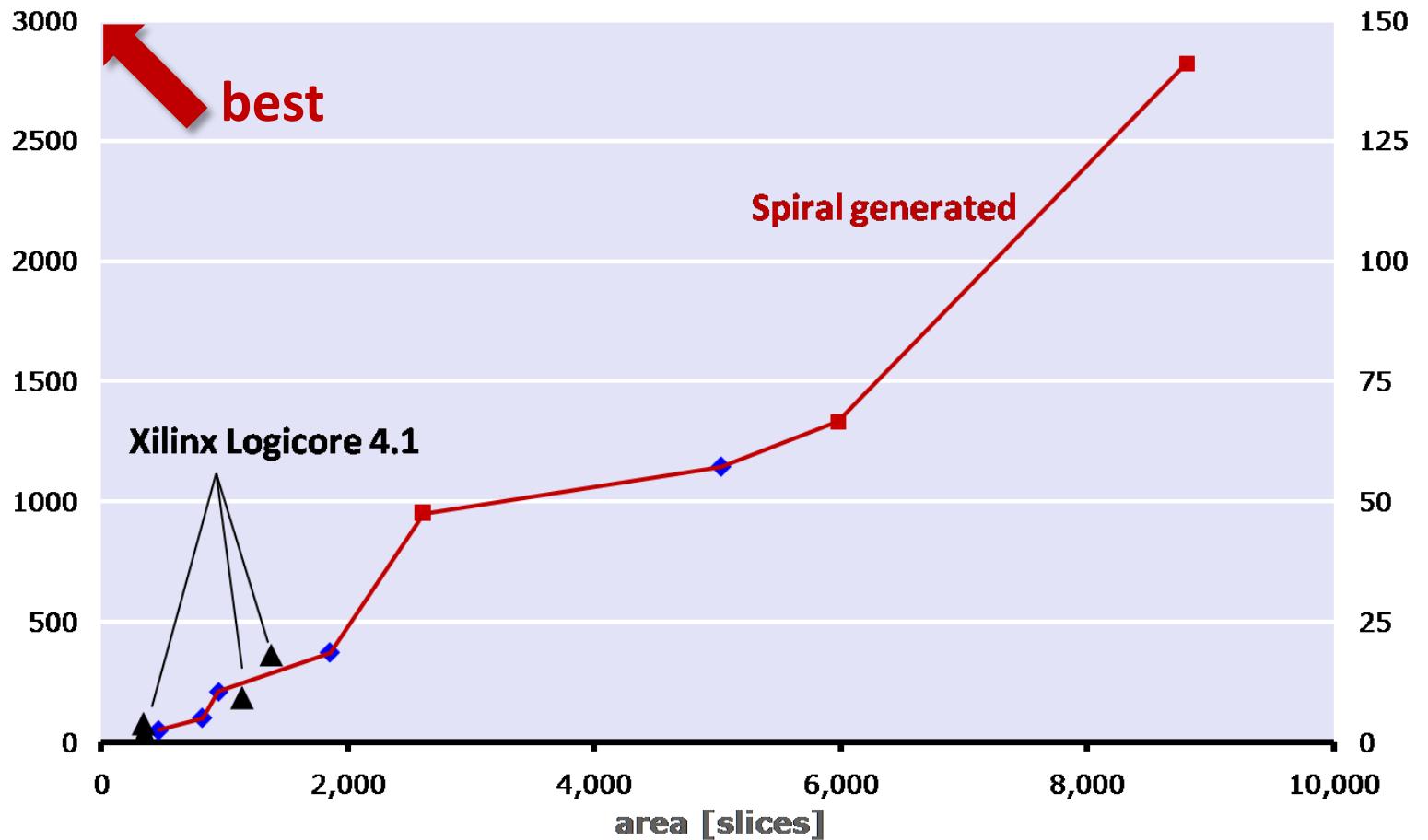
- 3984 C functions
- 1M lines of code
- Transforms: DFT (fwd+inv), RDFT (fwd+inv), DCT2, DCT3, DCT4, DHT, WHT
- Sizes: 2-64 (DFT, RDFT, DHT); 2-powers (DCTs, WHT)
- Precision: single, double
- Data type: scalar, SSE, AVX (DFT, DCT), LRB (DFT)

Mapping to FPGAs

DFT 1024 (16 bit fixed point) on Xilinx Virtex-5 FPGA

throughput [million samples per second]

performance [Gop/s]

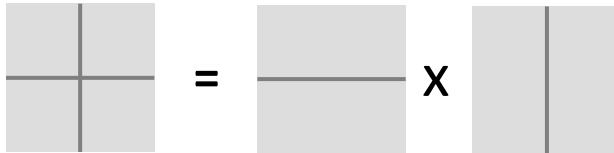


Organization

- Spiral: Basic system
- Parallelism
- General input size
- Results
- Final remarks

Beyond Transforms

Matrix-Matrix Multiplication



$$\text{MMM}_{1,1,1} \rightarrow (\cdot)_1$$

$$\text{MMM}_{m,n,k} \rightarrow (\otimes)_{m/m_b \times 1} \otimes \text{MMM}_{m_b,n,k}$$

$$\text{MMM}_{m,n,k} \rightarrow \text{MMM}_{m,nb,k} \otimes (\otimes)_{1 \times n/nb}$$

$$\begin{aligned} \text{MMM}_{m,n,k} \rightarrow & ((\Sigma_{k/k_b} \circ (\cdot)_{k/k_b}) \otimes \text{MMM}_{m,n,k_b}) \circ \\ & ((L_{k/k_b}^{mk/k_b} \otimes I_{k_b}) \times I_{kn}) \end{aligned}$$

$$\begin{aligned} \text{MMM}_{m,n,k} \rightarrow & (L_m^{mn/n_b} \otimes I_{n_b}) \circ \\ & ((\otimes)_{1 \times n/n_b} \otimes \text{MMM}_{m,n_b,k}) \circ \\ & (I_{km} \times (L_n^{kn/n_b} \otimes I_{n_b})) \end{aligned}$$

JPEG 2000 (Wavelet, EBCOT)



$$\text{SC}(\chi_{m,n}, \sigma_{m,n}) : (\mathbb{Z}_2^9 \times \mathbb{Z}_2^9) \rightarrow (\mathbb{N}, \mathbb{Z}_2)$$

$$(I \times \text{xor}_2) \circ (T_{SC} \times I) \circ (H \times V \times I) \circ (\underline{L_4^2} \times G_4) \circ (\underbrace{\left(\begin{array}{c} 1 \\ 1 \end{array} \right)}_{1} \times \left(\begin{array}{c} 1 \\ 1 \end{array} \right))$$

$$H, V : (\mathbb{Z}_2^9 \times \mathbb{Z}_2^9) \rightarrow \mathbb{N}$$

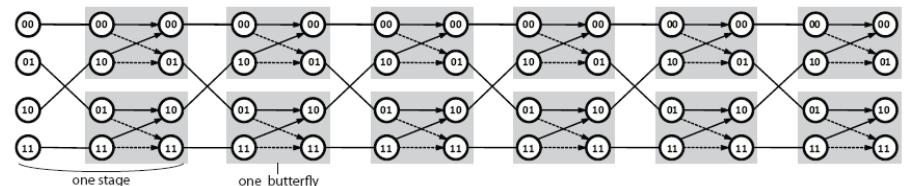
$$H: h \circ (f \times f) \circ (G_1 \times C_{-2} \times G_1 \times G_7 \times C_{-2} \times G_7) \circ \underline{L_4^2} \circ (\underline{\left(\begin{array}{c} 1 \\ 1 \end{array} \right)} \times \underline{\left(\begin{array}{c} 1 \\ 1 \end{array} \right)})$$

$$V: h \circ (f \times f) \circ (G_3 \times C_{-2} \times G_3 \times G_5 \times C_{-2} \times G_5) \circ \underline{L_4^2} \circ (\underline{\left(\begin{array}{c} 1 \\ 1 \end{array} \right)} \times \underline{\left(\begin{array}{c} 1 \\ 1 \end{array} \right)})$$

$$f : \text{mul}_2 \circ (I \times \text{sub}_2) \circ (I \times C_1 \times \text{mul}_2)$$

$$h : \min_2 \circ (C_1 \times \max_2) \circ (C_{-1} \times \text{sum}_2)$$

Viterbi Decoder



$$\mathbf{F}_{K,F} \rightarrow \prod_{i=1}^F \left((\mathbf{I}_{2^K-2} \otimes_j B_{F-i,j}) \mathbf{L}_{2^K-2}^{2^{K-1}} \right)$$

Synthetic Aperture Radar (SAR)



$$\text{SAR} \rightarrow 2\text{D-iDFT} \circ \text{Interpl} \circ \text{MatchFilt} \circ \text{prep}$$

$$2\text{D-iDFT} \rightarrow \text{iDFT} \otimes \text{iDFT}$$

$$\text{MatchFilt} \rightarrow \text{Filt} \circ (\mathbf{I} \times \mathbf{C}_f)$$

$$\text{Filt} \rightarrow (\mathbf{I} \otimes (\cdot))$$

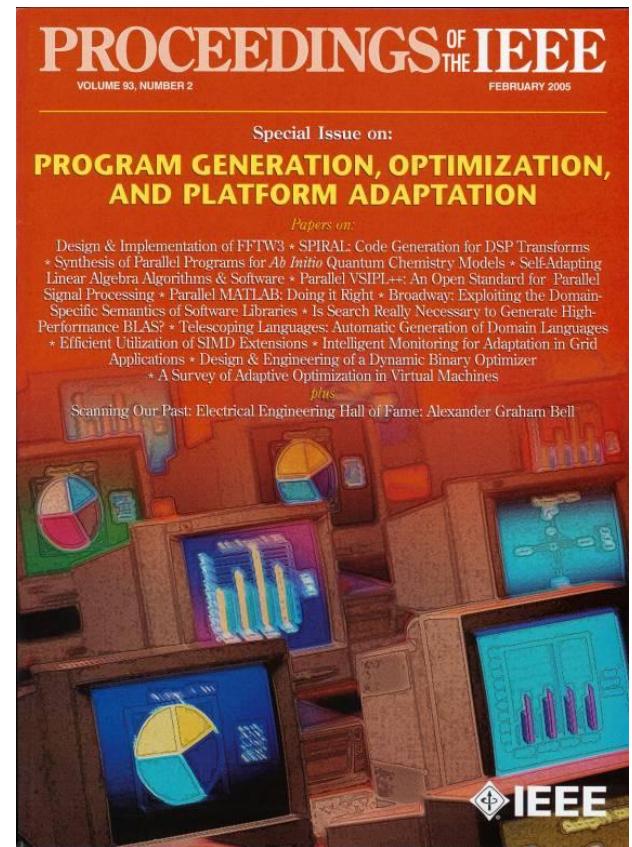
$$\text{Interpl} \rightarrow (\Sigma \otimes \mathbf{I}) \circ (\mathbf{I} \otimes_j S_{x_j \otimes y_j}) \circ \text{Filt} \circ ((\mathbf{I} \otimes \mathbf{1} \otimes \mathbf{I}) \times \mathbf{I}) \circ (\mathbf{I} \times \mathbf{C}_{i \otimes_j g_j})$$

Related Work: Autotuning

- **Goal:** (Partial) automation of runtime optimization

- Projects

- ATLAS (U. Tennessee)
- FFTW (MIT)
- BeBop/OSKI (U. Berkeley)
- Spiral (Carnegie Mellon)
- Tensor contractions (Ohio State)
- Fenics (some universities)
- FLAME (UT Austin)
- Adaptive sorting (UIUC)
- <many others>



Proceedings of the IEEE special issue, Feb. 2005

Spiral: Summary

■ Spiral:

Successful approach to automating
the development of computing software

Commercial proof-of-concept



■ Key ideas:

Algorithm knowledge:

Domain specific symbolic representation

Platform knowledge:

Tagged rewrite rules, SIMD specification

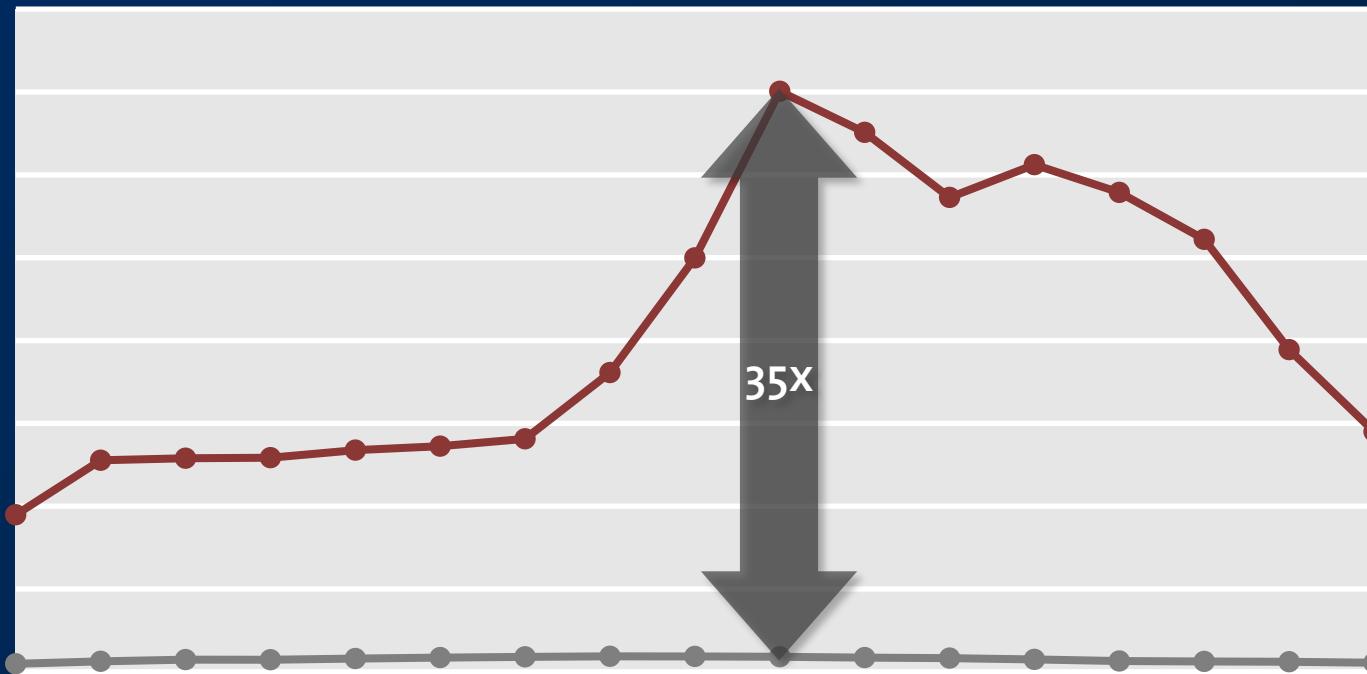
DFT₆₄



```
void dft64(float *y, float *x) {
    __m512 U912, U913, U914, U915, ...
    __m512 *a2153, *a2155;
    a2153 = ((__m512 *) x); s1107 = *(a2153);
    s1108 = *((a2153 + 4)); t1323 = _mm512_add_ps(s1107, s1108);
    t1324 = _mm512_sub_ps(s1107, s1108);
    <many more lines>
    U926 = _mm512_swizupconv_r32(...);
    s1121 = _mm512_madd231_ps(_mm512_mul_ps(_mm512_mask_or_pi(
        _mm512_set_1to16_ps(0.70710678118654757), 0xAAAA, a2154, U926), t1341),
        _mm512_mask_sub_ps(_mm512_set_1to16_ps(0.70710678118654757), ...),
        _mm512_swizupconv_r32(t1341, _MM_SWIZ_REG_CDAB));
    U927 = _mm512_swizupconv_r32
    <many more lines>
}
```

$$\text{DFT}_4 \rightarrow (\text{DFT}_2 \otimes \text{I}_2) \text{T}_2^4 (\text{I}_2 \otimes \text{DFT}_2) \text{L}_2^4$$

$$\underbrace{\text{I}_m \otimes A_n}_{\text{smp}(p,\mu)} \rightarrow \text{I}_p \otimes_{\parallel} \left(\text{I}_{m/p} \otimes A_n \right)$$



*Automating
High-Performance
Numerical Software
Development*

Programming languages
Program generation

Compilers

Symbolic Computation
Rewriting

Search
Learning

Algorithms
Mathematics

Software
Scientific Computing

```

/*
 * Copyright (c) 2003 Matteo Frigo
 * Copyright (c) 2003 Massachusetts Institute of Technology
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 */

```

```

/* This file was automatically generated --- DO NOT EDIT */
/* Generated on Sat Jul 5 21:29:38 EDT 2003 */

```

```
#include "codelet-dft.h"
```

```

/* Generated by: /homee/stevenj/cvs/fftw3.0.1/genfft/gen_notw_noinline -compact -variables 4 -n 32 -
name m1_32 -include n.h */

```

```

/*
 * This function contains 372 FP additions, 84 FP multiplications,
 * (or, 340 additions, 52 multiplications, 32 fused multiply/add),
 * 99 stack variables, and 128 memory accesses
 */
/*
 * Generator Id's :
 * $Id: algsimp.ml,v 1.7 2003/03/15 20:29:42 stevenj Exp $
 * $Id: fft.ml,v 1.2 2003/03/15 20:29:42 stevenj Exp $
 * $Id: gen_notw_noinline.ml,v 1.1 2003/04/17 11:07:19 athena Exp $
 */

```

```
#include "n.h"
```

```

static void m1_32_0(const R *ri, const R *ii, R *ro, R *io, stride is, stride os)
{
    DK(KP831469612, +0.831469612302545237078788377617905756738560812);
    DK(KP555570233, +0.555570233019602224742830813948532874374937191);
    DK(KP195090322, +0.195090322016128267848284868477022240927691618);
    DK(KP980785280, +0.980785280403230449126182236134239036973933731);
    DK(KP923879532, +0.923879532511286756128183189396788286822416626);
    DK(KP382683432, +0.382683432365089771728459984030398866761344562);
    DK(KP707106781, +0.707106781186547524400844362104849039284835938);
    {
        E T7, T4r, T4Z, T18, T1z, T3t, T3T, T2T, Te, T1f, T50, T4s, T2W, T3u, T1G;
        E T3U, Tm, T1n, T1O, T2Z, T3y, T3X, T4w, T53, Tt, T1u, T1V, T2Y, T3B, T3W;
        E T4z, T52, T2t, T3L, T3O, T2K, TR, TY, T5F, T5G, T5H, T5I, T4R, T5j, T2E;
        E T3P, T4W, T5k, T2N, T3M, T22, T3E, T3H, T2j, TC, TJ, T5A, T5B, T5C, T5D;
    }
}

```

```

E T4G, T5g, T2d, T3F, T4L, T5h, T2m, T3I;
{
    E T3, T1x, T14, T2S, T6, T2R, T17, T1y;
    {
        E T1, T2, T12, T13;
        T1 = ri[0];
        T2 = ri[WS(is, 16)];
        T3 = T1 + T2;
        T1x = T1 - T2;
        T12 = ii[0];
        T13 = ii[WS(is, 16)];
        T14 = T12 + T13;
        T2S = T12 - T13;
    }
    {
        E T4, T5, T15, T16;
        T4 = ri[WS(is, 8)];
        T5 = ri[WS(is, 24)];
        T6 = T4 + T5;
        T2R = T4 - T5;
        T15 = ii[WS(is, 8)];
        T16 = ii[WS(is, 24)];
        T17 = T15 + T16;
        T1y = T15 - T16;
    }
    T7 = T3 + T6;
    T4r = T3 - T6;
    T4Z = T14 - T17;
    T18 = T14 + T17;
    T1z = T1x - T1y;
    T3t = T1x + T1y;
    T3T = T2S - T2R;
    T2T = T2R + T2S;
}
{
    E Ta, T1B, T1b, T1A, Td, T1D, T1e, T1E;
    {
        E T8, T9, T19, T1a;
        T8 = ri[WS(is, 4)];
        T9 = ri[WS(is, 20)];
        Ta = T8 + T9;
        T1B = T8 - T9;
        T19 = ii[WS(is, 4)];
        T1a = ii[WS(is, 20)];
        T1b = T19 + T1a;
        T1A = T19 - T1a;
    }
    {
        E Tb, Tc, T1c, T1d;
        Tb = ri[WS(is, 28)];
        Tc = ri[WS(is, 12)];
        Td = Tb + Tc;
        T1D = Tb - Tc;
        T1c = ii[WS(is, 28)];
        T1d = ii[WS(is, 12)];
        T1e = T1c + T1d;
        T1E = T1c - T1d;
    }
}

```

```

}
Te = Ta + Td;
T1f = T1b + T1e;
T50 = Td - Ta;
T4s = T1b - T1e;
{
    E T2U, T2V, T1C, T1F;
    T2U = T1D - T1E;
    T2V = T1B + T1A;
    T2W = KP707106781 * (T2U - T2V);
    T3u = KP707106781 * (T2V + T2U);
    T1C = T1A - T1B;
    T1F = T1D + T1E;
    T1G = KP707106781 * (T1C - T1F);
    T3U = KP707106781 * (T1C + T1F);
}
}

E Ti, T1L, T1j, T1J, Tl, T1I, T1m, T1M, T1K, T1N;
{
    E Tg, Th, T1h, T1i;
    Tg = ri[WS(is, 2)];
    Th = ri[WS(is, 18)];
    Ti = Tg + Th;
    T1L = Tg - Th;
    T1h = ii[WS(is, 2)];
    T1i = ii[WS(is, 18)];
    T1j = T1h + T1i;
    T1J = T1h - T1i;
}
{

E Tj, Tk, T1k, T1l;
Tj = ri[WS(is, 10)];
Tk = ri[WS(is, 26)];
Tl = Tj + Tk;
T1l = Tj - Tk;
T1k = ii[WS(is, 10)];
T1l = ii[WS(is, 26)];
T1m = T1k + T1l;
T1M = T1k - T1l;
}
Tm = Ti + Tl;
T1n = T1j + T1m;
T1K = T1l + T1J;
T1N = T1L - T1M;
T1O = FNMS(KP923879532, T1N, KP382683432 * T1K);
T2Z = FMA(KP923879532, T1K, KP382683432 * T1N);
{
    E T3w, T3x, T4u, T4v;
    T3w = T1J - T1l;
    T3x = T1L + T1M;
    T3y = FNMS(KP382683432, T3x, KP923879532 * T3w);
    T3X = FMA(KP382683432, T3w, KP923879532 * T3x);
    T4u = T1j - T1m;
    T4v = Ti - Tl;
    T4w = T4u - T4v;
    T53 = T4v + T4u;
}

```

```

        }
    }

    E Tp, T1S, T1q, T1Q, Ts, T1P, T1t, T1T, T1R, T1U;
    {
        E Tn, To, T1o, T1p;
        Tn = ri[WS(is, 30)];
        To = ri[WS(is, 14)];
        Tp = Tn + To;
        T1S = Tn - To;
        T1o = ii[WS(is, 30)];
        T1p = ii[WS(is, 14)];
        T1q = T1o + T1p;
        T1Q = T1o - T1p;
    }

    E Tq, Tr, T1r, T1s;
    Tq = ri[WS(is, 6)];
    Tr = ri[WS(is, 22)];
    Ts = Tq + Tr;
    T1P = Tq - Tr;
    T1r = ii[WS(is, 6)];
    T1s = ii[WS(is, 22)];
    T1t = T1r + T1s;
    T1T = T1r - T1s;
}

Tt = Tp + Ts;
T1u = T1q + T1t;
T1R = T1P + T1Q;
T1U = T1S - T1T;
T1V = FMA(KP382683432, T1R, KP923879532 * T1U);
T2Y = FNMS(KP923879532, T1R, KP382683432 * T1U);
{
    E T3z, T3A, T4x, T4y;
    T3z = T1Q - T1P;
    T3A = T1S + T1T;
    T3B = FMA(KP923879532, T3z, KP382683432 * T3A);
    T3W = FNMS(KP382683432, T3z, KP923879532 * T3A);
    T4x = Tp - Ts;
    T4y = T1q - T1t;
    T4z = T4x + T4y;
    T52 = T4x - T4y;
}
}

E TN, T2p, T2J, T4S, TQ, T2G, T2s, T4T, TU, T2x, T2w, T4O, TX, T2z, T2C;
E T4P;
{
    E TL, TM, T2H, T2I;
    TL = ri[WS(is, 31)];
    TM = ri[WS(is, 15)];
    TN = TL + TM;
    T2p = TL - TM;
    T2H = ii[WS(is, 31)];
    T2I = ii[WS(is, 15)];
    T2J = T2H - T2I;
    T4S = T2H + T2I;
}

```

```

}
{
    E TO, TP, T2q, T2r;
    TO = ri[WS(is, 7)];
    TP = ri[WS(is, 23)];
    TQ = TO + TP;
    T2G = TO - TP;
    T2q = ii[WS(is, 7)];
    T2r = ii[WS(is, 23)];
    T2s = T2q - T2r;
    T4T = T2q + T2r;
}
{
    E TS, TT, T2u, T2v;
    TS = ri[WS(is, 3)];
    TT = ri[WS(is, 19)];
    TU = TS + TT;
    T2x = TS - TT;
    T2u = ii[WS(is, 3)];
    T2v = ii[WS(is, 19)];
    T2w = T2u - T2v;
    T4O = T2u + T2v;
}
{
    E TV, TW, T2A, T2B;
    TV = ri[WS(is, 27)];
    TW = ri[WS(is, 11)];
    TX = TV + TW;
    T2z = TV - TW;
    T2A = ii[WS(is, 27)];
    T2B = ii[WS(is, 11)];
    T2C = T2A - T2B;
    T4P = T2A + T2B;
}
T2t = T2p - T2s;
T3L = T2p + T2s;
T3O = T2J - T2G;
T2K = T2G + T2J;
TR = TN + TQ;
TY = TU + TX;
T5F = TR - TY;
{
    E T4N, T4Q, T2y, T2D;
    T5G = T4S + T4T;
    T5H = T4O + T4P;
    T5I = T5G - T5H;
    T4N = TN - TQ;
    T4Q = T4O - T4P;
    T4R = T4N - T4Q;
    T5j = T4N + T4Q;
    T2y = T2w - T2x;
    T2D = T2z + T2C;
    T2E = KP707106781 * (T2y - T2D);
    T3P = KP707106781 * (T2y + T2D);
}
{
    E T4U, T4V, T2L, T2M;
    T4U = T4S - T4T;

```

```

        T4V = TX - TU;
        T4W = T4U - T4V;
        T5k = T4V + T4U;
        T2L = T2z - T2C;
        T2M = T2x + T2w;
        T2N = KP707106781 * (T2L - T2M);
        T3M = KP707106781 * (T2M + T2L);
    }
}
{
E Ty, T2f, T21, T4C, TB, T1Y, T2i, T4D, TF, T28, T2b, T4I, TI, T23, T26;
E T4J;
{
E Tw, Tx, T1Z, T20;
Tw = ri[WS(is, 1)];
Tx = ri[WS(is, 17)];
Ty = Tw + Tx;
T2f = Tw - Tx;
T1Z = ii[WS(is, 1)];
T20 = ii[WS(is, 17)];
T21 = T1Z - T20;
T4C = T1Z + T20;
}
{
E Tz, TA, T2g, T2h;
Tz = ri[WS(is, 9)];
TA = ri[WS(is, 25)];
TB = Tz + TA;
T1Y = Tz - TA;
T2g = ii[WS(is, 9)];
T2h = ii[WS(is, 25)];
T2i = T2g - T2h;
T4D = T2g + T2h;
}
{
E TD, TE, T29, T2a;
TD = ri[WS(is, 5)];
TE = ri[WS(is, 21)];
TF = TD + TE;
T28 = TD - TE;
T29 = ii[WS(is, 5)];
T2a = ii[WS(is, 21)];
T2b = T29 - T2a;
T4I = T29 + T2a;
}
{
E TG, TH, T24, T25;
TG = ri[WS(is, 29)];
TH = ri[WS(is, 13)];
TI = TG + TH;
T23 = TG - TH;
T24 = ii[WS(is, 29)];
T25 = ii[WS(is, 13)];
T26 = T24 - T25;
T4J = T24 + T25;
}

```

```

T22 = T1Y + T21;
T3E = T2f + T2i;
T3H = T21 - T1Y;
T2j = T2f - T2i;
TC = Ty + TB;
TJ = TF + TI;
T5A = TC - TJ;
{
    E T4E, T4F, T27, T2c;
    T5B = T4C + T4D;
    T5C = T4I + T4J;
    T5D = T5B - T5C;
    T4E = T4C - T4D;
    T4F = TI - TF;
    T4G = T4E - T4F;
    T5g = T4F + T4E;
    T27 = T23 - T26;
    T2c = T28 + T2b;
    T2d = KP707106781 * (T27 - T2c);
    T3F = KP707106781 * (T2c + T27);
{
        E T4H, T4K, T2k, T2l;
        T4H = Ty - TB;
        T4K = T4I - T4J;
        T4L = T4H - T4K;
        T5h = T4H + T4K;
        T2k = T2b - T28;
        T2l = T23 + T26;
        T2m = KP707106781 * (T2k - T2l);
        T3I = KP707106781 * (T2k + T2l);
    }
}
{
    E T4B, T57, T5a, T5c, T4Y, T56, T55, T5b;
{
    E T4t, T4A, T58, T59;
    T4t = T4r - T4s;
    T4A = KP707106781 * (T4w - T4z);
    T4B = T4t + T4A;
    T57 = T4t - T4A;
    T58 = FNMS(KP923879532, T4L, KP382683432 * T4G);
    T59 = FMA(KP382683432, T4W, KP923879532 * T4R);
    T5a = T58 - T59;
    T5c = T58 + T59;
}
{
    E T4M, T4X, T51, T54;
    T4M = FMA(KP923879532, T4G, KP382683432 * T4L);
    T4X = FNMS(KP923879532, T4W, KP382683432 * T4R);
    T4Y = T4M + T4X;
    T56 = T4X - T4M;
    T51 = T4Z - T50;
    T54 = KP707106781 * (T52 - T53);
    T55 = T51 - T54;
    T5b = T51 + T54;
}

```

```

ro[WS(os, 22)] = T4B - T4Y;
io[WS(os, 22)] = T5b - T5c;
ro[WS(os, 6)] = T4B + T4Y;
io[WS(os, 6)] = T5b + T5c;
io[WS(os, 30)] = T55 - T56;
ro[WS(os, 30)] = T57 - T5a;
io[WS(os, 14)] = T55 + T56;
ro[WS(os, 14)] = T57 + T5a;
}

{
E T5f, T5r, T5u, T5w, T5m, T5q, T5p, T5v;
{
    E T5d, T5e, T5s, T5t;
    T5d = T4r + T4s;
    T5e = KP707106781 * (T53 + T52);
    T5f = T5d + T5e;
    T5r = T5d - T5e;
    T5s = FNMS(KP382683432, T5h, KP923879532 * T5g);
    T5t = FMA(KP923879532, T5k, KP382683432 * T5j);
    T5u = T5s - T5t;
    T5w = T5s + T5t;
}
{
    E T5i, T5l, T5n, T5o;
    T5i = FMA(KP382683432, T5g, KP923879532 * T5h);
    T5l = FNMS(KP382683432, T5k, KP923879532 * T5j);
    T5m = T5i + T5l;
    T5q = T5l - T5i;
    T5n = T50 + T4Z;
    T5o = KP707106781 * (T4w + T4z);
    T5p = T5n - T5o;
    T5v = T5n + T5o;
}
ro[WS(os, 18)] = T5f - T5m;
io[WS(os, 18)] = T5v - T5w;
ro[WS(os, 2)] = T5f + T5m;
io[WS(os, 2)] = T5v + T5w;
io[WS(os, 26)] = T5p - T5q;
ro[WS(os, 26)] = T5r - T5u;
io[WS(os, 10)] = T5p + T5q;
ro[WS(os, 10)] = T5r + T5u;
}

{
E T5z, T5P, T5S, T5U, T5K, T5O, T5N, T5T;
{
    E T5x, T5y, T5Q, T5R;
    T5x = T7 - Te;
    T5y = T1n - T1u;
    T5z = T5x + T5y;
    T5P = T5x - T5y;
    T5Q = T5D - T5A;
    T5R = T5F + T5I;
    T5S = KP707106781 * (T5Q - T5R);
    T5U = KP707106781 * (T5Q + T5R);
}
{
    E T5E, T5J, T5L, T5M;
}

```

```

T5E = T5A + T5D;
T5J = T5F - T5I;
T5K = KP707106781 * (T5E + T5J);
T5O = KP707106781 * (T5J - T5E);
T5L = T18 - T1f;
T5M = Tt - Tm;
T5N = T5L - T5M;
T5T = T5M + T5L;
}
ro[WS(os, 20)] = T5z - T5K;
io[WS(os, 20)] = T5T - T5U;
ro[WS(os, 4)] = T5z + T5K;
io[WS(os, 4)] = T5T + T5U;
io[WS(os, 28)] = T5N - T5O;
ro[WS(os, 28)] = T5P - T5S;
io[WS(os, 12)] = T5N + T5O;
ro[WS(os, 12)] = T5P + T5S;
}

{
E Tv, T5V, T5Y, T60, T10, T11, T1w, T5Z;
{
E Tf, Tu, T5W, T5X;
Tf = T7 + Te;
Tu = Tm + Tt;
Tv = Tf + Tu;
T5V = Tf - Tu;
T5W = T5B + T5C;
T5X = T5G + T5H;
T5Y = T5W - T5X;
T60 = T5W + T5X;
}
{
E TK, TZ, T1g, T1v;
TK = TC + TJ;
TZ = TR + TY;
T10 = TK + TZ;
T11 = TZ - TK;
T1g = T18 + T1f;
T1v = T1n + T1u;
T1w = T1g - T1v;
T5Z = T1g + T1v;
}
ro[WS(os, 16)] = Tv - T10;
io[WS(os, 16)] = T5Z - T60;
ro[0] = Tv + T10;
io[0] = T5Z + T60;
io[WS(os, 8)] = T11 + T1w;
ro[WS(os, 8)] = T5V + T5Y;
io[WS(os, 24)] = T1w - T11;
ro[WS(os, 24)] = T5V - T5Y;
}
{
E T1X, T33, T31, T37, T2o, T34, T2P, T35;
{
E T1H, T1W, T2X, T30;
T1H = T1z - T1G;
T1W = T1O - T1V;

```

```

T1X = T1H + T1W;
T33 = T1H - T1W;
T2X = T2T - T2W;
T30 = T2Y - T2Z;
T31 = T2X - T30;
T37 = T2X + T30;
}

{

E T2e, T2n, T2F, T2O;
T2e = T22 - T2d;
T2n = T2j - T2m;
T2o = FMA(KP980785280, T2e, KP195090322 * T2n);
T34 = FNMS(KP980785280, T2n, KP195090322 * T2e);
T2F = T2t - T2E;
T2O = T2K - T2N;
T2P = FNMS(KP980785280, T2O, KP195090322 * T2F);
T35 = FMA(KP195090322, T2O, KP980785280 * T2F);
}

{

E T2Q, T38, T32, T36;
T2Q = T2o + T2P;
ro[WS(os, 23)] = T1X - T2Q;
ro[WS(os, 7)] = T1X + T2Q;
T38 = T34 + T35;
io[WS(os, 23)] = T37 - T38;
io[WS(os, 7)] = T37 + T38;
T32 = T2P - T2o;
io[WS(os, 31)] = T31 - T32;
io[WS(os, 15)] = T31 + T32;
T36 = T34 - T35;
ro[WS(os, 31)] = T33 - T36;
ro[WS(os, 15)] = T33 + T36;
}

}

{

E T3D, T41, T3Z, T45, T3K, T42, T3R, T43;
{

E T3v, T3C, T3V, T3Y;
T3v = T3t - T3u;
T3C = T3y - T3B;
T3D = T3v + T3C;
T41 = T3v - T3C;
T3V = T3T - T3U;
T3Y = T3W - T3X;
T3Z = T3V - T3Y;
T45 = T3V + T3Y;
}

{

E T3G, T3J, T3N, T3Q;
T3G = T3E - T3F;
T3J = T3H - T3I;
T3K = FMA(KP555570233, T3G, KP831469612 * T3J);
T42 = FNMS(KP831469612, T3G, KP555570233 * T3J);
T3N = T3L - T3M;
T3Q = T3O - T3P;
T3R = FNMS(KP831469612, T3Q, KP555570233 * T3N);
T43 = FMA(KP831469612, T3N, KP555570233 * T3Q);
}

```

```

        }
    {
        E T3S, T46, T40, T44;
        T3S = T3K + T3R;
        ro[WS(os, 21)] = T3D - T3S;
        ro[WS(os, 5)] = T3D + T3S;
        T46 = T42 + T43;
        io[WS(os, 21)] = T45 - T46;
        io[WS(os, 5)] = T45 + T46;
        T40 = T3R - T3K;
        io[WS(os, 29)] = T3Z - T40;
        io[WS(os, 13)] = T3Z + T40;
        T44 = T42 - T43;
        ro[WS(os, 29)] = T41 - T44;
        ro[WS(os, 13)] = T41 + T44;
    }
}

{
    E T49, T4l, T4j, T4p, T4c, T4m, T4f, T4n;
    {
        E T47, T48, T4h, T4i;
        T47 = T3t + T3u;
        T48 = T3X + T3W;
        T49 = T47 + T48;
        T4l = T47 - T48;
        T4h = T3T + T3U;
        T4i = T3y + T3B;
        T4j = T4h - T4i;
        T4p = T4h + T4i;
    }
}

{
    E T4a, T4b, T4d, T4e;
    T4a = T3E + T3F;
    T4b = T3H + T3I;
    T4c = FMA(KP980785280, T4a, KP195090322 * T4b);
    T4m = FNMS(KP195090322, T4a, KP980785280 * T4b);
    T4d = T3L + T3M;
    T4e = T3O + T3P;
    T4f = FNMS(KP195090322, T4e, KP980785280 * T4d);
    T4n = FMA(KP195090322, T4d, KP980785280 * T4e);
}

{
    E T4g, T4q, T4k, T4o;
    T4g = T4c + T4f;
    ro[WS(os, 17)] = T49 - T4g;
    ro[WS(os, 1)] = T49 + T4g;
    T4q = T4m + T4n;
    io[WS(os, 17)] = T4p - T4q;
    io[WS(os, 1)] = T4p + T4q;
    T4k = T4f - T4c;
    io[WS(os, 25)] = T4j - T4k;
    io[WS(os, 9)] = T4j + T4k;
    T4o = T4m - T4n;
    ro[WS(os, 25)] = T4l - T4o;
    ro[WS(os, 9)] = T4l + T4o;
}
}

```

```

{
    E T3b, T3n, T3l, T3r, T3e, T3o, T3h, T3p;
    {
        E T39, T3a, T3j, T3k;
        T39 = T1z + T1G;
        T3a = T2Z + T2Y;
        T3b = T39 + T3a;
        T3n = T39 - T3a;
        T3j = T2T + T2W;
        T3k = T1O + T1V;
        T3l = T3j - T3k;
        T3r = T3j + T3k;
    }
}

{
    E T3c, T3d, T3f, T3g;
    T3c = T22 + T2d;
    T3d = T2j + T2m;
    T3e = FMA(KP555570233, T3c, KP831469612 * T3d);
    T3o = FNMS(KP555570233, T3d, KP831469612 * T3c);
    T3f = T2t + T2E;
    T3g = T2K + T2N;
    T3h = FNMS(KP555570233, T3g, KP831469612 * T3f);
    T3p = FMA(KP831469612, T3g, KP555570233 * T3f);
}
}

{
    E T3i, T3s, T3m, T3q;
    T3i = T3e + T3h;
    ro[WS(os, 19)] = T3b - T3i;
    ro[WS(os, 3)] = T3b + T3i;
    T3s = T3o + T3p;
    io[WS(os, 19)] = T3r - T3s;
    io[WS(os, 3)] = T3r + T3s;
    T3m = T3h - T3e;
    io[WS(os, 27)] = T3l - T3m;
    io[WS(os, 11)] = T3l + T3m;
    T3q = T3o - T3p;
    ro[WS(os, 27)] = T3n - T3q;
    ro[WS(os, 11)] = T3n + T3q;
}
}

}

static void m1_32(const R *ri, const R *ii, R *ro, R *io, stride is, stride os, int v, int ivs, int ovs)
{
    int i;
    for (i = v; i > 0; i -= 1) {
        m1_32_0(ri, ii, ro, io, is, os);
        ri += ivs;
        ii += ivs;
        ro += ovs;
        io += ovs;
    }
}

static const kdft_desc desc = { 32, "m1_32", {340, 52, 32, 0}, &GENUS, 0, 0, 0, 0 };
void X(codelet_m1_32) (planner *p) {

```

```
X(kdft_register) (p, m1_32, &desc);  
}
```

```

/*
 * Copyright (c) 2003 Matteo Frigo
 * Copyright (c) 2003 Massachusetts Institute of Technology
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 */

```

```

/* This file was automatically generated --- DO NOT EDIT */
/* Generated on Sat Jul 5 21:29:31 EDT 2003 */

```

```
#include "codelet-dft.h"
```

```

/* Generated by: /homee/stevenj/cvs/fftw3.0.1/genfft/gen_notw -compact -variables 4 -n 2 -name n1_2 -
include n.h */

```

```

/*
 * This function contains 4 FP additions, 0 FP multiplications,
 * (or, 4 additions, 0 multiplications, 0 fused multiply/add),
 * 5 stack variables, and 8 memory accesses
 */
/*
 * Generator Id's :
 * $Id: algsimp.ml,v 1.7 2003/03/15 20:29:42 stevenj Exp $
 * $Id: fft.ml,v 1.2 2003/03/15 20:29:42 stevenj Exp $
 * $Id: gen_notw.ml,v 1.22 2003/04/17 11:07:19 athena Exp $
 */

```

```
#include "n.h"
```

```

static void n1_2(const R *ri, const R *ii, R *ro, R *io, stride is, stride os, int v, int ivs, int ovs)
{
    int i;
    for (i = v; i > 0; i = i - 1, ri = ri + ivs, ii = ii + ivs, ro = ro + ovs, io = io + ovs) {
        E T1, T2, T3, T4;
        T1 = ri[0];
        T2 = ri[WS(is, 1)];
        ro[WS(os, 1)] = T1 - T2;
        ro[0] = T1 + T2;
        T3 = ii[0];
        T4 = ii[WS(is, 1)];
        io[WS(os, 1)] = T3 - T4;
        io[0] = T3 + T4;
    }
}

```

```
}

static const kdft_desc desc = { 2, "n1_2", {4, 0, 0, 0}, &GENUS, 0, 0, 0, 0 };
void X(codelet_n1_2) (planner *p) {
    X(kdft_register) (p, n1_2, &desc);
}
```

```

/*
 * Copyright (c) 2003 Matteo Frigo
 * Copyright (c) 2003 Massachusetts Institute of Technology
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 */

```

```

/* This file was automatically generated --- DO NOT EDIT */
/* Generated on Sat Jul 5 21:29:32 EDT 2003 */

```

```
#include "codelet-dft.h"
```

```

/* Generated by: /homee/stevenj/cvs/fftw3.0.1/genfft/gen_notw -compact -variables 4 -n 3 -name n1_3 -
include n.h */

```

```

/*
 * This function contains 12 FP additions, 4 FP multiplications,
 * (or, 10 additions, 2 multiplications, 2 fused multiply/add),
 * 15 stack variables, and 12 memory accesses
 */
/*
 * Generator Id's :
 * $Id: algsimp.ml,v 1.7 2003/03/15 20:29:42 stevenj Exp $
 * $Id: fft.ml,v 1.2 2003/03/15 20:29:42 stevenj Exp $
 * $Id: gen_notw.ml,v 1.22 2003/04/17 11:07:19 athena Exp $
 */

```

```
#include "n.h"
```

```

static void n1_3(const R *ri, const R *ii, R *ro, R *io, stride is, stride os, int v, int ivs, int ovs)
{
    DK(KP5000000000, +0.500000000000000000000000000000000000000000000000000000000000000);
    DK(KP866025403, +0.866025403784438646763723170752936183471402627);
    int i;
    for (i = v; i > 0; i = i - 1, ri = ri + ivs, ii = ii + ivs, ro = ro + ovs, io = io + ovs) {
        E T1, Ta, T4, T9, T8, Tb, T5, Tc;
        T1 = ri[0];
        Ta = ii[0];
        {
            E T2, T3, T6, T7;
            T2 = ri[WS(is, 1)];
            T3 = ri[WS(is, 2)];
            T4 = T2 + T3;

```

```

T9 = KP866025403 * (T3 - T2);
T6 = ii[WS(is, 1)];
T7 = ii[WS(is, 2)];
T8 = KP866025403 * (T6 - T7);
Tb = T6 + T7;
}
ro[0] = T1 + T4;
io[0] = Ta + Tb;
T5 = FNMS(KP500000000, T4, T1);
ro[WS(os, 2)] = T5 - T8;
ro[WS(os, 1)] = T5 + T8;
Tc = FNMS(KP500000000, Tb, Ta);
io[WS(os, 1)] = T9 + Tc;
io[WS(os, 2)] = Tc - T9;
}
}

static const kdft_desc desc = { 3, "n1_3", {10, 2, 2, 0}, &GENUS, 0, 0, 0, 0 };
void X(codelet_n1_3) (planner *p) {
    X(kdft_register) (p, n1_3, &desc);
}

```

Matrix-matrix multiplication (MMA)

$$\begin{matrix} \cdot \\ C \end{matrix} = \begin{matrix} - \\ A \end{matrix} \cdot \begin{matrix} | \\ B \end{matrix} + \begin{matrix} \cdot \\ C \end{matrix}$$

square: $n \times n$ times $n \times n$

for $i = 0 \dots n-1$

for $j = 0 \dots n-1$

for $l = 0 \dots n-1$

$$c_{i,j} = c_{i,j} + a_{i,l} \cdot b_{l,j}$$

asymptotic run time

$$\mathcal{O}(n^3)$$

$$\text{cost} = (\# \text{ adds}, \# \text{ mults}) \\ (n^3, n^3)$$

$$\text{cost} = \# \text{ flops} \\ 2n^3$$

Cost analysis: solving recurrences

First order: $f_0 = c, f_k = af_{k-1} + s_k, k \geq 1$
 $\Rightarrow f_k = a^k c + \sum_{i=0}^{k-1} a^i s_{k-i}$

Example: $f_0 = 0, f_k = 2 \cdot f_{k-1} + 3 \cdot 2^{k-1} - 1$
 $\Rightarrow f_k = \sum_{i=0}^{k-1} 2^i (3 \cdot 2^{k-i-1} - 1) = \dots = \frac{3}{2} k 2^k - 2^k + 1$

Exponential version: $g_0 = c, g_n = a g_{n/2} + s_n \quad n = 2^k$
 [substitute $n = 2^k, g_n = f_k, s_n = s_k$]

$$\Rightarrow f_0 = c, f_k = a f_{k-1} + s_k$$

solve as before, translate back

Example: $g_0 = 0, g_n = 2 g_{n/2} + \frac{3}{2} n - 1$
 $\stackrel{n=2^k}{\Rightarrow} f_0 = 0, f_k = 2 f_{k-1} + 3 \cdot 2^{k-1} - 1$

$$\xrightarrow{\text{solve}} f_k = \frac{3}{2} k 2^k - 2^k + 1$$

$$\xrightarrow{\text{translate}} g_n = \frac{3}{2} n \log_2(n) - n + 1$$

rectangular: $k \times m$ times $m \times n$

for $i = 0 \dots k-1$

for $j = 0 \dots n-1$

for $l = 0 \dots m-1$

$$c_{i,j} = c_{i,j} + a_{i,l} \cdot b_{l,j}$$

$$\mathcal{O}(kmn)$$

$$(kmn, kmn)$$

$$2kmn$$

$$n=2^k$$

Master theorem

$$g_n = \mathcal{O}(n \log n)$$

Exact solution of higher order recurrences

Method of generating functions

sequence f_0, f_1, f_2, \dots \longleftrightarrow generating function $\sum_{n=0}^{\infty} f_n x^n = F(x)$

second order example:

$f_0 = 0, f_1 = 1, f_k = f_{k-1} + f_{k-2}, k \geq 2$ (Fibonacci numbers)

1.) Multiply by x^k and sum

$$\sum f_n x^n = \sum f_{n-1} x^n + \sum f_{n-2} x^n$$

2.) Determine summation boundaries (here: $k \geq 2$)

$$\sum_{k=2}^{\infty} f_k x^k = x \sum_{k=1}^{\infty} x^k + x^2 \sum_{k=0}^{\infty} x^k$$

3.) Use generating function

$$F(x) - x = x(F(x)) + x^2 F(x)$$

4.) Solve

$$F(x) = \frac{x}{1-x-x^2} \quad \begin{matrix} \leftarrow \\ \text{normalize to 1} \end{matrix} \quad \begin{matrix} \text{characteristic} \\ \text{polynomial} \\ \text{of recurrence} \end{matrix}$$

5.) Partial fraction expansion

$$F(x) = \frac{x}{(1-\alpha x)(1-\alpha' x)} = \frac{A}{1-\alpha x} + \frac{B}{1-\alpha' x} \quad (\alpha \neq \alpha')$$

$1/\alpha, 1/\alpha'$ are zeros of $1-x-x^2$

$\Leftrightarrow \alpha, \alpha'$ " $x^2 - x - 1$ (mirrored polynomial)

$$\alpha, \alpha' = \frac{1}{2} \pm \frac{1}{2}\sqrt{5}$$

$$A = F(x)(1-\alpha x) \Big|_{x=\frac{1}{\alpha}} = \frac{x}{1-\alpha' x} \Big|_{x=\frac{1}{\alpha}} = \frac{1}{\sqrt{5}}$$

$$B = F(x)(1-\alpha' x) \Big|_{x=\frac{1}{\alpha'}} = \frac{x}{1-\alpha x} \Big|_{x=\frac{1}{\alpha'}} = -\frac{1}{\sqrt{5}}$$

6.) Evolve into series

$$F(x) = A \sum_{k=0}^{\infty} \alpha^k x^k + B \sum_{k=0}^{\infty} (\alpha')^k x^k$$

$$f_k = A \alpha^k + B (\alpha')^k = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^k - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^k$$

7.) Read off f_k : $f_k = A \alpha^k + B (\alpha')^k = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^k - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^k$

Enables solving recurrences of the form

$$g_1 = c, g_2 = d, g_n = a g_{n/2} + b g_{n/4}, n = 2^k, k \geq 2$$

by substituting $n = 2^k, g_n = f_k$

Why blocking?

assume: cache size 64u
 cache line = 8 doubles
 only 1 cache

$\text{C17} = \text{cache miss}$

1.) Triple loop MIM



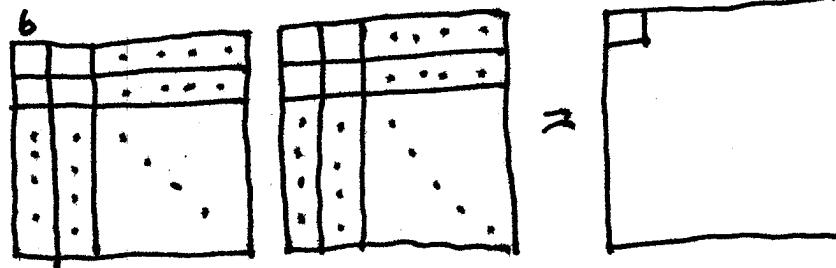
1. entry: $\frac{n}{8} + n$ C17s (compulsory)

afterwards: x is in cache

2. entry: no reuse, so again $\frac{n}{8} + n$ C17s

$$\Rightarrow \text{total} = \left(\frac{n}{8} + n\right)n^2 = \frac{3}{8}n^3 \text{ C17s}$$

2.) Blocked MIM



choose: $b \geq 8$
 (cache line)
 and $8 \mid b$
 and $3b^2 \leq c$
 $c = \text{cache size}$

$$1. \text{ block: } \frac{nb}{8} + \frac{4b}{8} = \frac{5b}{4} \text{ C17s}$$

2. block: same

$$\Rightarrow \text{total} = \frac{nb}{4} \cdot \left(\frac{n}{b}\right)^2 = \frac{n^3}{4b}$$

$$\text{choose } b = \sqrt{\frac{c}{3}} \Rightarrow \frac{\sqrt{3}}{4\sqrt{c}} n^3 \text{ C17s}$$

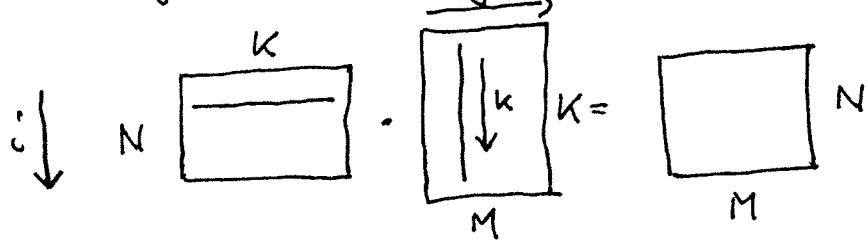
$$\text{gain: } \approx 2.5\sqrt{c}$$

- Explains much of triple loop's poor performance (the other major optimization is unrolling and scalar replacement for better instruction parallelism and register usage)

- Blocking achieves both: better spatial and better temporal locality with respect to the cache

MM generation with ATLAS

Starting point: standard triple loop



for $i = 0 : 1 : N - 1$

 for $j = 0 : 1 : M - 1$

 for $k = 0 : 1 : K - 1$

$$c_{ij} = c_{ij} + a_{ik} b_{kj}$$

$$C = C + AB$$

Important cases (from most to least):

- ~~two~~ two out of (N, K, M) are small

- one out of (N, K, M) is small

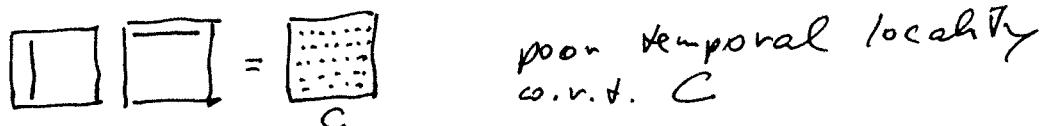
- none out of (N, K, M) is small

Reason: Based on how it is used inside LAPACK

- 1.) loop order: i, j, k loops can be permuted into any order
- ijk : B is reused, good if $M < N$
 - ijk : A " , " " $N < M$

ATLAS generates versions for both

- other choices are bad, e.g., kij :



2.) blocking for cache



we assume for simplicity
 $N_B (N, M, K)$

mini-MTM

for $i = 0 : N_B : N - 1$
 for $j = 0 : N_B : M - 1$
 for $k = 0 : N_B : K - 1$
 for $i' = i : 1 : i + N_B - 1$
 for $j' = j : 1 : j + N_B - 1$
 for $k' = k : 1 : k + N_B - 1$
 $c_{ij'} = c_{ij'} + a_{ik'} b_{kj'}$

This blocking is done using loop tiling + loop exchange

loop tiling: for $i = 0 : 1 : N-1$
(example) $a_i = \dots$

\Rightarrow for $i = 0 : 4 : N-1$
for $j = i : 1 : i+3$
 $a_j = \dots$

N_B becomes a search parameter.

Bound $N_B^2 \leq C$ (cache size)

this is a loose bound — the search takes care of the rest.

For which cache should one block?

- The original ATLAS blocked for L1-cache

- L2 was off-die

- the important cases (see above) of MM M have one or two of N, k, M small

- For large, square matrices blocking for L2 can make sense

- Blocking for L1 & L2 can make sense

- Versions with different block sizes can make sense

3.) Blocking $M_{hi} - M_{lo}$ for registers

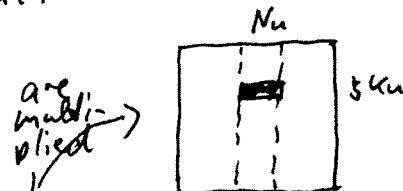
Now: k as outermost for ILP:

ijk

$$\boxed{\quad} \boxed{\quad} = \boxed{\quad}$$

In instructions:
- n independent mults
- n dependent adds
($\geq \log_2(n)$ steps)

But: $2n+1$ "live" variables



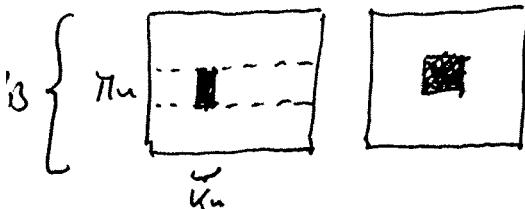
U_{ij} :

$$\boxed{\quad} \boxed{\quad} = \boxed{\quad}$$

$2n^2$ instructions:
- n^2 indep. mults
- n^2 " adds

$n^2 + 2n$ "live" variables

More "register pressure" since larger working set (always when more ILP)



for $i = 0 : N_B : N - 1$
 for $j = 0 : N_B : M - 1$
 for $k = 0 : N_B : K - 1$
 for $i' = i : N_u : i + N_B - 1$
 for $j' = j : N_u : j + N_B - 1$
 for $k' = k : K_u : k + N_B - 1$
 for $k'' = k' : l : k' + K_u - 1$
 for $i'' = i' : l : i' + N_u - 1$
 for $j'' = j' : l : j' + N_u - 1$
 $c_{i''j''} = c_{i''j''} + a_{i''k''} b_{k''j''}$

mini-MMM
 { micro-MMM } \otimes

M_u, N_u, K_u become search parameters, we assume all $/N_B$
 Bound: $\underbrace{K_u + N_u + N_u N_u}_{\text{live variables in } \otimes} \leq N_R$ (no. of neighbors)

4.) Basic block optimizations

step 1: unroll \otimes

$$\begin{aligned}
 c_{...} &= c_{...} + a_{...} s_{...} \\
 c_{...} &= c_{...} + a_{...} s_{...} \\
 &\vdots
 \end{aligned}
 \quad \left. \begin{array}{l} M_u N_u \text{ many} \\ \text{a..., b... reused} \\ c... \text{ not} \end{array} \right\}$$

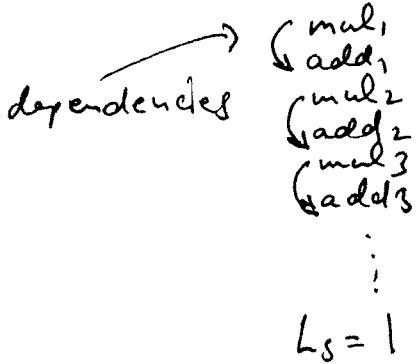
step 2: scalar replacement

Typically we would only do it on $a_{...}, s_{...}$
 (since reused) but since we later unroll the k'' -loop, $c_{...}$ will be reused as well.

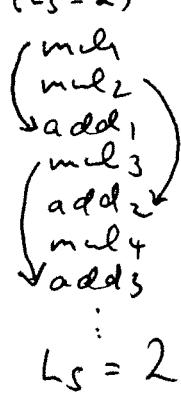
$$\begin{aligned}
 t_0 &= c_{...} \\
 t_1 &= a_{...} \\
 t_2 &= s_{...} \\
 &\vdots \\
 t_3 &= t_0 + t_1 t_2 \\
 &\vdots \\
 c_{...} &= t_3
 \end{aligned}
 \quad \left. \begin{array}{l} \text{loads } (M_u + N_u + N_u N_u) \text{ many} \\ \rightarrow \text{all fit into register} \end{array} \right\} \text{computations } \quad \begin{array}{l} M_u N_u \text{ adds} \\ M_u N_u \text{ multi} \end{array} \\
 \quad \left. \begin{array}{l} \\ \end{array} \right\} \text{stores } M_u N_u \text{ many}$$

step 3: optimize computation part by "skewing"

computation as is:



skewed:
 $(L_s = 2)$



skewed by L_s :

dependent mul \rightarrow add separated by $2L_s - 1$ instructions
(except for beginning)

L_s is search parameter

Bound: $M_u + N_u + M_u N_u + L_s \leq N_{12}$

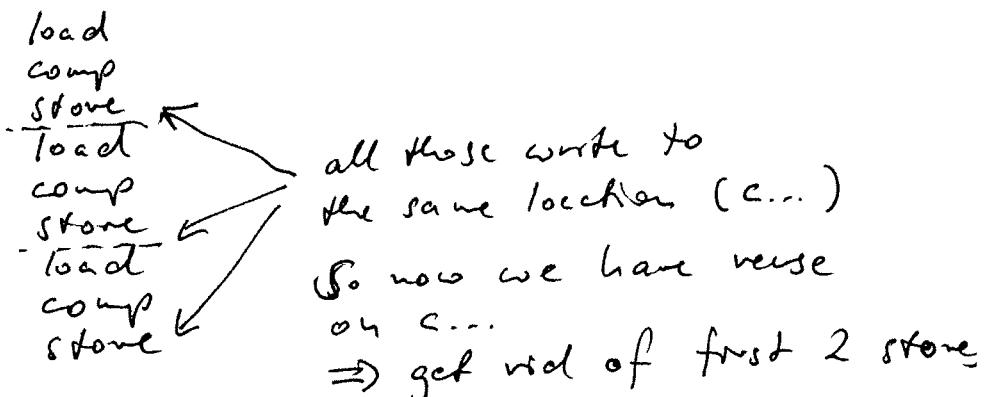
Also: Not all loads are done in the beginning. Only block of I_F loads and later blocks of N_F loads as needed.

step 4: unroll k^y loop

K_u is limited by 1-cache size

Bound: $K_u \leq \frac{N_0}{2}$

Example: $K_u = 3$



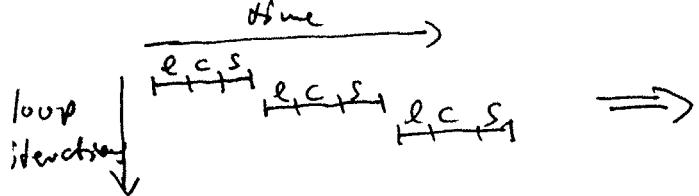
Result:

load
comp
load
comp
load
comp
store

step 5: (Software pipelining)

Reorganise k'-loop so loads of one iteration
are moved to previous iteration

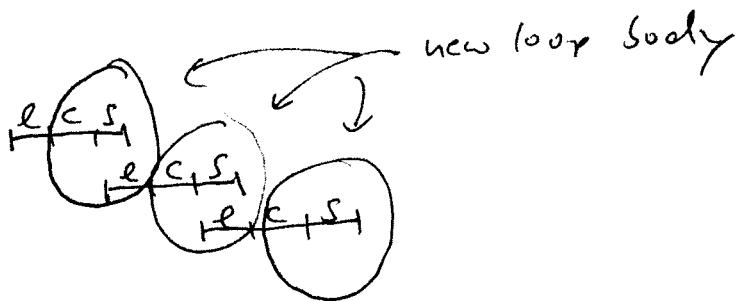
Conceptually:



l = load

c = compute

s = store



5.) Buffering to avoid TCB misses

Comes a little later

Model - Kassel H(LHS) (Yodor et al.)

Basic idea: Use more hardware parameters than ATLAS and use models to predict mini-MIN parameters.

Additional hardware parameters: $C_1 = L1$ cache size
 $B_1 = \text{cache block size}$
 $L_x = f!. \text{ point latency}$

Note: I work with ijk loop order, the paper with jik
 (reason: in class we work with C, ATLAS with Fortran)

1.) Estimate N_B

Assume cache is fully associative, refine model in steps

Idea: working set has to fit in cache

$$\text{mini-MIN: } \underset{\substack{\text{outermost} \\ \text{loop}}}{\downarrow} i \cdot \boxed{a} \cdot \underset{i}{\overrightarrow{\boxed{b}}} = \boxed{c} \quad \{N_B\}$$

a.) easy bound: working set = $3N_B^2$
 $\Rightarrow 3N_B^2 \leq C_1$

b.) closer analysis

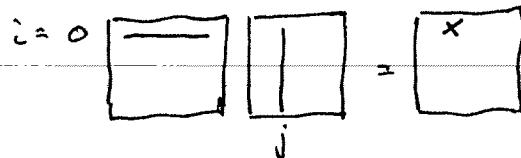
$$N_B^2 + N_B + 1 \leq C_1$$

\uparrow \uparrow \nearrow
 \mathbf{b} row of a element of c

c.) take into account cache block size, i.e.,
 units are cache blocks now

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq \frac{C_1}{B_1}$$

d.) take into account LRU replacement
 build a history of array elements say touched



$i=0$:

$$\begin{array}{lll}
 a_{00} b_{00} \dots a_{0N_B-1} b_{N_B-1,0} & c_{00} & (j=0) \\
 a_{00} b_{01} \dots a_{0N_B-1} b_{N_B-1,1} & c_{01} & (j=1) \\
 \dots & \dots & \dots \\
 a_{00} b_{0N_B-1} \dots a_{0N_B-1} b_{N_B-1,N_B-1} & c_{0N_B-1} & (j=N_B-1)
 \end{array}$$

corresponding history:

$$\begin{array}{ll}
 b_{00} \dots b_{N_B-1,0} & c_{00} \\
 b_{01} \dots b_{N_B-1,1} & c_{01} \\
 \dots & \dots \\
 a_{00} b_{0N_B-1} \dots a_{0N_B-1} b_{N_B-1,N_B-1} & c_{0N_B-1}
 \end{array}$$

more recent

Observations:

- all of b , starting with b_{00} , has to be in cache for $i=1$
- When $i=1$, row 1 of a will not cleanly replace row 0 of a .
- When $i=1$, element of c will not cleanly replace the previous elements of c .

\Rightarrow This has to fit:

- entire b
- 2 rows of a (here: a_{0*}, a_{1*})
- 1 row of c (here: c_{0*})
- 1 element of c (here: $c_{1,0}$)

$$\text{Result: } \left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq \frac{C_1}{B_1}$$

e.) take into account register blocking

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B M_u}{B_1} \right\rceil + \left\lceil \frac{M_u N_u}{B_1} \right\rceil \leq \frac{C_1}{B_1}$$

f.) shrink N_B so $M_u, N_u / N_B$ (avoids clean-up code)

2.) Estimate M_u, N_u

micro-MMM: $M_u \begin{array}{|c|} \hline 1 \\ \hline a \\ \hline \end{array} \begin{array}{|c|} \hline s \\ \hline \end{array} = \begin{array}{|c|} \hline \text{...} \\ \hline c \\ \hline \end{array}$

a.) since scalar replacement tools routine ~~row~~^{column} of a, column row of s and c is reused:

$$M_u N_u + M_u + N_u \leq N_R$$

b.) after skipping, L_s more registers are needed

$$M_u N_u + M_u + N_u + L_s \leq N_R$$

- solve assuming $M_u = N_u$, but both have to be at least 1
- adjust (increase) N_u if possible

so roughly $M_u \approx N_u$ (will perform poorly on Intel, move later)

3.) Estimate L_s

mul.
;
mults
add1
mults+1
add2
;
add L_s

2 $L_s - 1$ instructions
in between

$$\Rightarrow L_s = \lceil \frac{L_x + 1}{2} \rceil$$

4.) Estimate K_u

choose such that loop body fits into L-cache

make sure $K_u \mid N_B$ (avoid clean-up code)

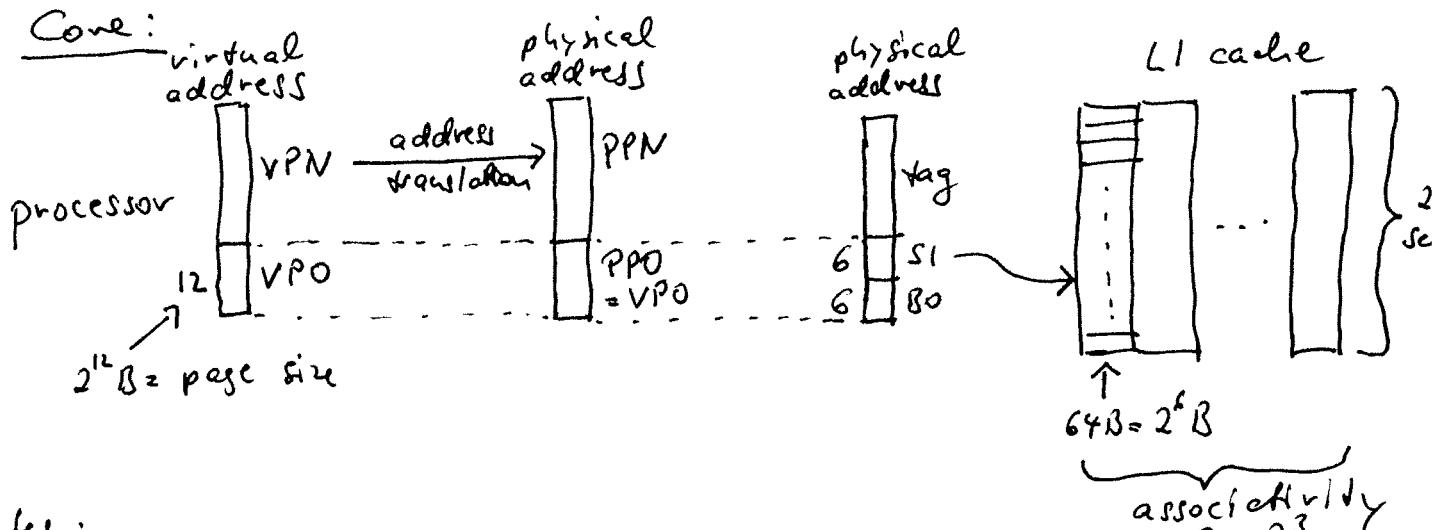
In most cases: $K_u = N_B$.

Optimizations Related to the Virtual Memory System

Background: (6re)

- the processor works with virtual addresses
- all caches work with physical addresses
- both address spaces are organized in pages
- typical page size: 4KB
- so the address translation translates virtual page numbers into physical page numbers

Cave:



Notes:

VPN = virtual page number

VPO = " " offset

PPN = physical page number

PPO = " " offset

S1 = set index

BO = block offset

address translation: $VPN \rightarrow PPN$

$VPO = PPO = S1 \cup BO \Rightarrow$ cache lookup can start before $VPN \rightarrow PPN$ translation is finished

address translation

- uses a cache called translation lookaside buffer (TLB)

- Case 2: two levels of caches for loads

DTLB0: 16 entries

DTLB1: 256 entries

Case 1: DTLB0 hit: no penalty

DTLB1 hit: 2 cycle penalty

miss: possibly very expensive

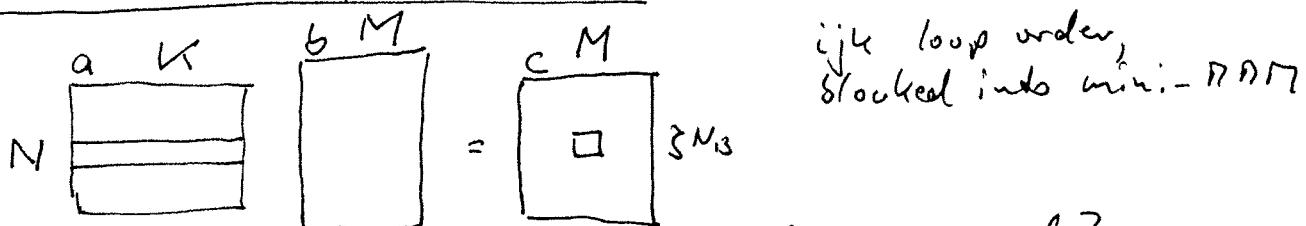
Pentium 4:
- one TCB
- 64 entries

Consequence: Repeatedly accessing a working set that is spread over >256 pages leads to TLB misses \rightarrow possible severe slowdown

Solution 1: use larger pages
may require different kernel (os) and C std library

Solution 2 (if possible): copy working set into contiguous memory
 \Rightarrow less pages are used

How does this affect MM2?



which memory regions are repeatedly accessed?

- block rows of a : is contiguous

- all of b : is contiguous

- file of c : can be spread over N_B pages

if $M > 512$ (512 doubles = 4KB = page size)

But: typically $N_B \ll 100 \ll \text{size (DTLB)}$

so at most 2 cycles penalty per row

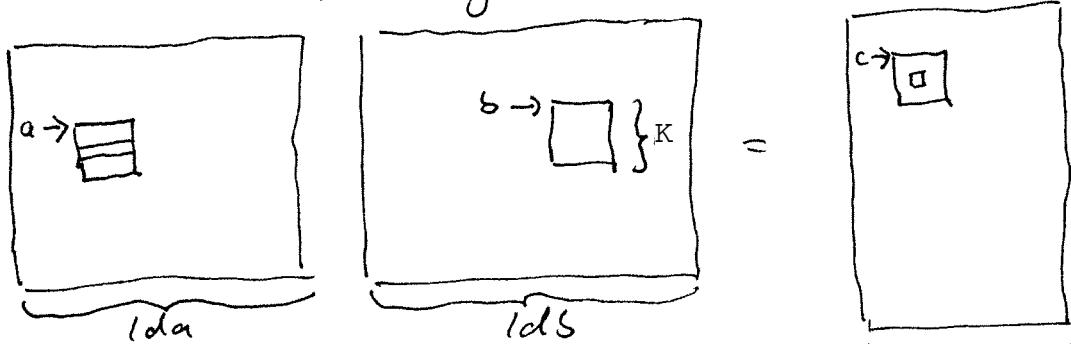
\Rightarrow not worth to copy (on Core)

But: the BLAS 3 function `dgemm` has this interface:

`dgemm(a, b, c, N, K, M, lda, ldb, ldc)`

$\underbrace{a, b, c}_{\text{matrix pointers}}$ $\underbrace{N, K, M}_{\text{matrix dimensions}}$ $\underbrace{\overbrace{\text{leading}}^{\text{"leading"}}$ $\overbrace{\text{dimensions}}^{\text{"leading"}}$

The leading dimensions enable `dgemm` to be called on submatrices of larger matrices:



which memory regions are repeatedly accessed? ldc

- block rows of a : spread over $\leq N_B$ pages

- all of b : spread over $\leq K$ pages

- file of c : spread over $\leq N_B$ pages

Here copy may pay for large enough K

Code :

```
// all of B reused : possibly copy  
for i = 0 : N_B : N - 1  
    // block row of A reused : possibly copy  
    for j = 0 : N_A : M - 1  
        // tile of C reused : possibly copy  
        for k = 0 : N_B : K - 1  
            . . . . .
```

Gauss Elimination and LU factorization

Standard Gauss Elimination: $A = [a_{ij}]_{1 \leq i, j \leq n}$

$$\xrightarrow{-\frac{a_{ii}}{a_{11}}} \left(\begin{array}{c|ccccc} a_{11} & a_{12} & \cdots & \cdots \\ 0 & \square & & & \\ \vdots & & & & \\ 0 & & & & \end{array} \right)$$

$$\downarrow$$

$$\left(\begin{array}{c|ccccc} a_{11} & a_{12} & \cdots & \cdots \\ 0 & \square & & & \\ \vdots & & A' & & \\ 0 & & & & \end{array} \right) = \left(\begin{array}{cccc|c} -\frac{a_{21}}{a_{11}} & 1 & \cdots & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ -\frac{a_{n1}}{a_{11}} & 0 & \cdots & \cdots & 1 \end{array} \right) \cdot A$$

$$\downarrow$$

$$\left(\begin{array}{c|ccccc} a_{11} & a_{12} & \cdots & \cdots \\ 0 & a_{22} & \cdots & & \\ 0 & 0 & \square & & \\ \vdots & & A'' & & \\ 0 & 0 & & & \end{array} \right) = \left(\begin{array}{cccc|c} 1 & 0 & & & 0 \\ 0 & 1 & & & 0 \\ 1 & -\frac{a_{21}}{a_{11}} & 1 & \cdots & 0 \\ 0 & -\frac{a_{31}}{a_{11}} & 0 & \cdots & 1 \end{array} \right) \left(\begin{array}{cccc|c} -\frac{a_{31}}{a_{11}} & 1 & \cdots & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ -\frac{a_{n1}}{a_{11}} & 0 & \cdots & \cdots & 1 \end{array} \right) A$$

$$\downarrow$$

$$U = \left(\begin{array}{c|ccccc} & & & & \\ \diagdown & & & & \\ & & & & \\ 0 & & & & \end{array} \right) = \left(\begin{array}{cccc|c} 1 & & & & 0 \\ 0 & 1 & & & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{array} \right) \cdots \left(\begin{array}{cccc|c} 1 & & & & 0 \\ x & 1 & & & 0 \\ 1 & 0 & 1 & \cdots & 0 \\ x & 0 & 0 & \cdots & 1 \end{array} \right) A$$

$$\Rightarrow A = \left(\begin{array}{cccc|c} \frac{a_{21}}{a_{11}} & 1 & & & 0 \\ \vdots & \ddots & \ddots & & \vdots \\ \frac{a_{n1}}{a_{11}} & 0 & \cdots & 1 & 0 \end{array} \right) \left(\begin{array}{cccc|c} 1 & 0 & & & 0 \\ 0 & 1 & & & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{array} \right) \cdots \quad U$$

$$= \underbrace{\left(\begin{array}{cccc|c} \frac{a_{21}}{a_{11}} & \frac{a_{31}}{a_{11}} & 1 & & 0 \\ \vdots & \frac{a_{n1}}{a_{11}} & \frac{a_{n2}}{a_{11}} & \cdots & 1 \\ \frac{a_{21}}{a_{11}} & \frac{a_{32}}{a_{11}} & 0 & \cdots & \vdots \\ \vdots & & & \ddots & \vdots \\ \frac{a_{21}}{a_{11}} & \frac{a_{32}}{a_{11}} & 0 & \cdots & 1 \end{array} \right)}_{L} \cdot U$$

Gauss-Elimination factors $A = LU$

$L = \text{lower triangular} + \text{diagonal elements} = I$

$U = \text{upper triangular}$

Let $A = LU$. Then

$$Ax = b \Leftrightarrow LUx = b$$

can be solved in 2 steps:

$$\text{solve } Ly = b$$

$$\text{solve } Ux = y$$

Cost:

$$Ly = b \Leftrightarrow \triangle \cdot y = b$$

$$y_1 = b_1$$

$$y_2 = b_2 - l_{21} b_1$$

$$y_3 = b_3 - l_{31} b_1 - l_{32} b_2$$

2 flops

4 flops

$$\text{flops: } \sum_{i=1}^{n-1} 2i = n^2 + O(n)$$

$Ux = y$: same flop count + n divisions

\Rightarrow Given $A = LU$, $Ax = b$ can be solved in $2n^2 + O(n)$ flops

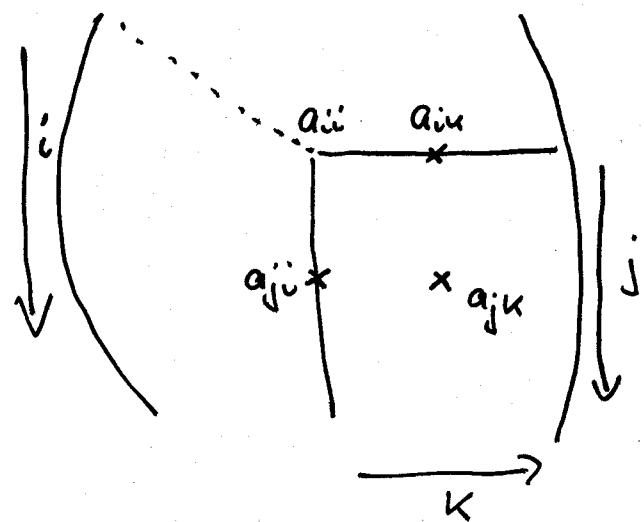
LU factorization: Algorithm

for $i = 1:n-1$

 for $j = i+1:n$

 for $k = i:n$

$$a_{jk} = a_{jk} - \frac{a_{ji}}{a_{ii}} a_{ik}$$



Let's optimize

1.) $k = i$ yields zero elements (we know that)

for $i = 1 : n-1$
for $j = i+1 : n$ change
for $k = i+1 : n$
 $a_{jk} = a_{jk} - \frac{a_{ii}}{a_{ii}} a_{ik}$

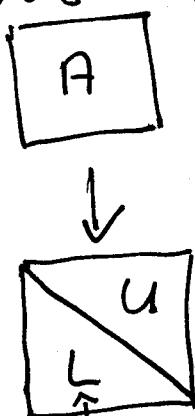
2.) $\frac{a_{ii}}{a_{ii}} = l_{ji}$ independent of k

for $i = 1 : n-1$
for $j = i+1 : n$
 $l_{ji} = \frac{a_{ji}}{a_{ii}}$
for $k = i+1 : n$
 $a_{jk} = a_{jk} - \cancel{\frac{a_{ii}}{a_{ii}}} l_{ji} \cdot a_{ik}$

3.) Store l_{ji} in q_{ji} , since q_{ji} never touched again after zeroed out. Also, avoid division by a_{ii}

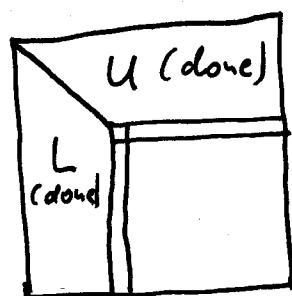
for $i = 1 : n-1$
 $c = 1/a_{ii}$
for $j = i+1 : n$
 $a_{ji} = c \cdot a_{ji}$
for $k = i+1 : n$
 $a_{jk} = a_{jk} - \cancel{a_{ii}} q_{ji} \cdot a_{ik}$

computes:



without
diagonal
(all 1's anyway)

intermediate step:



Cost analysis

- $(n-1)$ divisions

$$\begin{aligned} \text{- flops: } & \sum_{i=1}^{n-1} \sum_{j=i+1}^n \left(1 + \sum_{k=i+1}^n 2 \right) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n (2(n-i) + 1) \\ & = \sum_{i=1}^{n-1} (2(n-i)^2 + (n-i)) = \sum_{i=1}^{n-1} (2i^2 + i) \end{aligned}$$

$$\begin{aligned} \text{Use: } & \sum_{i=1}^{n-1} i^2 = \frac{(n-1)n(2n-1)}{6} = \frac{1}{3}n^3 + O(n^2) \\ & = \frac{2}{3}n^3 + O(n^2) \end{aligned}$$

Result:

- $A = LU$ can be computed using $\frac{2}{3}n^3 + O(n^2)$ flops
- $Ax = b$ can be solved using $\frac{2}{3}n^3 + O(n^2)$ flops
(provided $A = LU$ exists, more later)

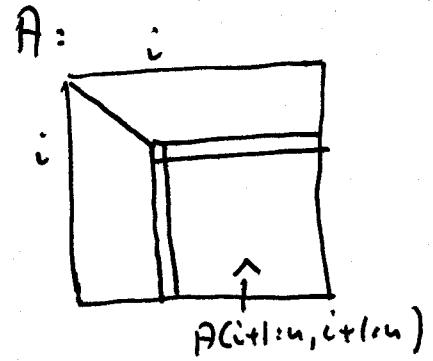
Vector form of LU factorization

for $i = 1 : n-1$

$$\boxed{\text{BLAS1}} \quad A(i+1:n, i) = A(i+1:n, i) / A(i, i)$$

$$\boxed{\text{BLAS2}} \quad A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) \cdot A(i, i+1:n)$$

called
"rank-1 update"



Pivoting:

Theorem: $A = LU$ exists if in each step of Gauss elimination

$$a_{ii} \neq 0$$

\Leftrightarrow all matrices $A(1:j, 1:j)$ are invertible
(non-singular)

Example: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ has no LU factorization

Modify Gauss elimination:

GEPP (Gauss elimination with partial pivoting):

for $i = 1:n-1$

permute rows s.t. $a_{ii} \neq 0$ and $|a_{ii}|$ largest in column

for numerical stability

[as before]

- computes $A = PLU$, P a permutations matrix
- adds $O(n^2)$ comparisons

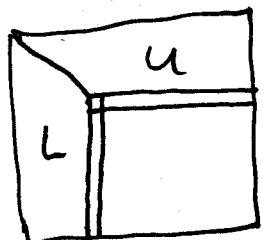
Blocking (as implemented in LAPACK)

Reminder: Why blocking?

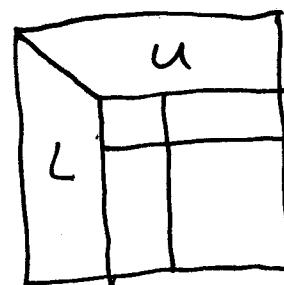
	memory ops	flops	ratio
BLAS 1	$3n$	$2n-1$	$\sim \frac{2}{3}$
BLAS 2	$n^2 + 2n$	$2n^2 - n$	~ 2
BLAS 3	$3n^2$	$2n^3 - n^3$	$\sim \frac{2}{3}n$
FFT	$2n(3n)$	$\sim 4n \log n$	$O(\log n)$

[

Basic idea:



standard



$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

step 1: LU factorization of

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} \cdot U_{11}$$

using standard algorithm

then: $A = \begin{pmatrix} L_{11} U_{11} & A_{12} \\ L_{21} U_{11} & A_{22} \end{pmatrix}$

$$= \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} U_{11} & L_{11}^{-1} A_{12} \\ 0 & A_{22} - L_{21} L_{11}^{-1} A_{12} \end{pmatrix}$$

$$= \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & A_{22} - L_{21} U_{12} \end{pmatrix}$$

$$U_{12} = L_{11}^{-1} A_{12}$$

$$U_{22} = A_{22} - L_{21} U_{12}$$

thus:

step 2: $U_{12} = L_{11}^{-1} A_{12}$

step 3: $U_{22} = A_{22} - L_{21} U_{12}$

final algorithm:

for $i = 1 : b : n-1$

use standard algorithm to factorize $A(i:i+n, i:i+b-1) = (L_{ii})U_{11}$

$$A(i:i+b-1, i:b:n) = L_{ii}^{-1} A(i:i+b-1, i:b:n)$$

$$A(i+b:n, i:b:n) = A(i+b:n, i:b:n)$$

$$- A(i+b:n, i:i+b-1) \cdot A(i:i+b-1, i:b:n)$$

BLAS3

BLAS3
(mm)

Matrix inversion

Given A , find M :

$AM = I_n \Leftrightarrow$ solving n systems
of linear equations

$$A m_i = e_i$$

↑
ith column of M

step 1: compute $A = PLU$
 $(\frac{2}{3}n^3 + O(n^2))$

step 2: solve n systems of i.e.
 $(2n^2 + O(n)$ each)

$$\Rightarrow \text{cost } \frac{8}{3}n^3 + O(n^2)$$

Determinant

step 1: compute $A = PLU$

step 2: $\det(A) = \prod_{i=1}^n u_{ii} \cdot \underbrace{\det(P)}_{\pm 1}$

$$\Rightarrow \text{cost } \frac{2}{3}n^3 + O(n^2)$$

Upper performance bound for blocked sparse MVM

Vuduc et al., Supercomputing 2002

Idea: Determine amount of data that needs to be transferred from memory.

Upper bound from memory bandwidth.

Amount of data for $r \times c$ BCSR:

$A: rc K_{r,c}$ doubles

$K_{r,c} + \frac{m}{r} + 1$ ints

$x: n$ doubles

$y: m$ doubles

$$\Rightarrow 8(rc K_{r,c} + m + n) + 4(K_{r,c} + \frac{m}{r} + 1) \text{ bytes}$$

Core 2: Memory bandwidth $2B/\text{cycle}$

$$\Rightarrow \text{runtime} \geq 4(rc K_{r,c} + m + n) + 2(K_{r,c} + \frac{m}{r} + 1) \text{ cycle}$$

Bound based on computation:

Every $r \times c$ block incurs $2rc$ ops, total: $2rc K_{r,c}$ ops

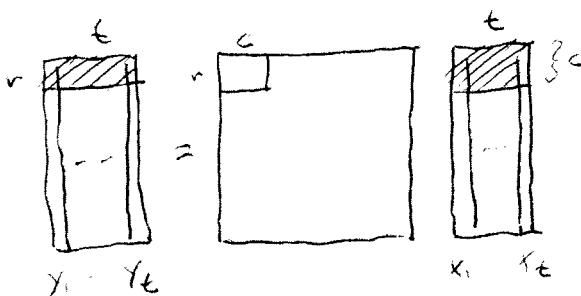
Core 2 (no SSE): $2\text{ops}/\text{cycle}$

$$\Rightarrow \text{runtime} \geq rc K_{r,c} \text{ cycles} \quad \underline{\text{looser bound}}$$

Multiple vectors in MVM

$$y_i = y_i + Ax_i, i=1..t \quad \text{Now reuse in } A!$$

To exploit perform $SIMMs$ interleaved and blocked



Similar to $MATM$

4.4 Matrix-vector multiplication

$$\begin{matrix} a \\ b \\ c \\ d \end{matrix} \begin{matrix} | & | & | & | \\ | & | & | & | \\ | & | & | & | \\ | & | & | & | \end{matrix} \cdot \begin{matrix} | \\ | \\ | \\ | \end{matrix} = \begin{matrix} | \\ | \\ | \\ | \end{matrix}$$

$x \qquad \qquad \qquad y$

scalar: $\frac{16 \text{ mults}}{12 \text{ adds}}$
 $\underline{\qquad\qquad\qquad 28 \text{ ops}}$

1. step: $4 \times \text{mm-mul-ps}$ to compute pointwise products ax, bx, cx, dx

Result:

$$\begin{matrix} ax \\ bx \\ cx \\ dx \end{matrix} \begin{matrix} | & | & | & | \\ | & | & | & | \\ | & | & | & | \\ | & | & | & | \end{matrix}$$

SSE

2. step: transpose ~~-MM-TRANSPOSE4-ps~~ (8 instructions)

Result

$$\begin{matrix} | & | & | & | \\ | & | & | & | \\ | & | & | & | \\ | & | & | & | \end{matrix}$$

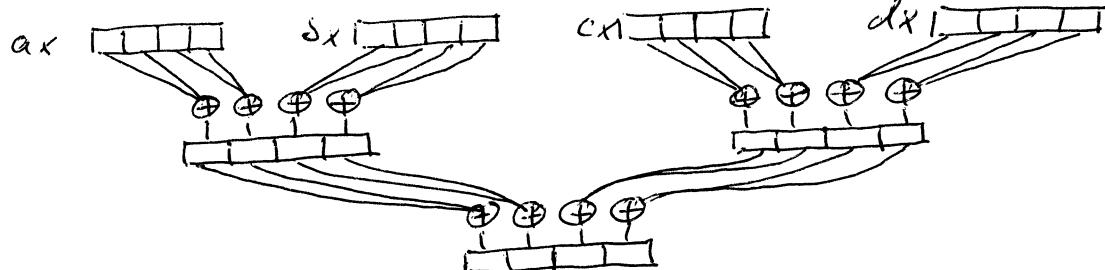
3. step: $3 \times \text{mm-add-ps}$ to add columns

Total: 15 instructions

Vectorization efficiency: $\frac{28}{15} \approx 1.87$

SSE3

2 steps: tree reduction with $3 \times \text{mm-addd-ps}$



Total: 7 instructions

Vectorization efficiency: $\frac{28}{7} = 4$ perfect!

SSE4

has dot product but yields still 7 instructions

Linear transform

compute $y = Tx$

where x is the input vector, y the output vector
and T the fixed transform matrix

Example: DFT

1. form (standard in SP): given x_0, \dots, x_{n-1} , compute

$$y_k = \sum_{e=0}^{n-1} e^{-2\pi j k e / n} \cdot x_e, \quad \text{for } 0 \leq k < n, \quad j = \sqrt{-1}$$

2. form: set $\omega_n = e^{-2\pi j / n}$ (primitive, n th root of 1)

$$y_k = \sum_{e=0}^{n-1} \omega_n^{ke} x_e, \quad \text{for } 0 \leq k < n$$

3. form: $x = (x_0, \dots, x_{n-1})^T$, $y = (y_0, \dots, y_{n-1})^T$, compute

$$y = DFT_n \cdot x, \quad DFT_n = [\omega_n^{ke}]_{0 \leq k, e < n}$$

For example:

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad DFT_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & j & -1 & -j \end{bmatrix}$$

Transform algorithms

you want to compute $y = Tx$, T is $n \times n$.

an algorithm is given by a factorization

$$T = T_1 T_2 \cdots T_m, \quad \text{all } T_i \text{ are } n \times n$$

namely you can compute y as:

$$t_1 = T_m x$$

$$t_2 = T_{m-1} t_1 \quad (\text{m steps} = m \text{ VIs})$$

$$\vdots$$

$$y = T_1 \cdot t_{m-1}$$

this reduces the operations count only if

- the T_i are sparse

- m is not too large

Note: generic $N \times N$ (i.e., matrix is not known beforehand) has complexity $\Theta(n^2)$

Example: fast Fourier transform (FFT) for $n=4$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & j \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

compare cost:

- by definition: 12 adds, 4 mults by j (all complex)
- using FFT (4 steps): 8 adds, 1 mult by j

the sparse matrices are structured:

$$= (\text{DFT}_2 \otimes \text{I}_2) \text{ diag}(1, 1, 1, j) (\text{I}_2 \otimes \text{DFT}_2) L_2^4$$

(explained next)

Structured matrices

- $\text{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ (butterfly matrix)
- $\text{I}_n = \begin{pmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & 1 \end{pmatrix}$
- $\text{diag}(a_0, \dots, a_{n-1}) = \begin{pmatrix} a_0 & & & \\ & \ddots & & \\ & & a_{n-1} & \\ & & & a_{n-1} \end{pmatrix}$
- $A \oplus B = \begin{pmatrix} A & \\ & B \end{pmatrix}$
- $A \otimes B = [a_{k,e} \cdot B]_{0 \leq k, e \leq n}$ if $A = [a_{k,e}]_{0 \leq k, e \leq n}$
 $A \text{ } n \times n, \text{ } B \text{ } m \times m \Rightarrow A \otimes B \text{ } nm \times nm$

Example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \text{DFT}_2 = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & -1 & 1 & -2 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ \frac{3}{2} & \frac{3}{2} & \frac{3}{4} & \frac{3}{4} \\ 3 & -3 & 1 & -4 \end{bmatrix}$$

\uparrow
coarse
structure \uparrow
fine
structure

$$I_n \otimes A = \begin{bmatrix} A & & \\ & A & \\ & & \ddots \\ & & & A \end{bmatrix}$$

contains n As

$$A \otimes I_n = \begin{bmatrix} \bullet & \bullet & \bullet & \dots \\ \bullet & \bullet & \bullet & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

- each block is an $A \otimes I_n$
- all circles together constitute one A
- contains n As

General radix, recursive Cooley-Tukey FFT

assume $n = km$

$$\mathcal{DFT}_{km} = (\mathcal{DFT}_k \otimes I_m) \overline{T}_m (\overline{I}_n \otimes \mathcal{DFT}_m) L_k^n$$

↑
radix ↗
diagonal
matrix

3 key structures: $I_n \otimes A_m$, $A_k \otimes I_m$, L_k^n

1.) $y = (I_n \otimes A_m)x$

$$\begin{pmatrix} y \\ \vdots \end{pmatrix} = \begin{pmatrix} A & & \\ & A & \\ & & \ddots & \\ & & & A \end{pmatrix} \begin{pmatrix} x \\ \vdots \end{pmatrix}$$

n A's at stride 1

for $i = 0 : k-1$
 $y[i:m:1:i+m-1] = A \cdot x \quad "$

2.) $y = (A_k \otimes I_m)x$

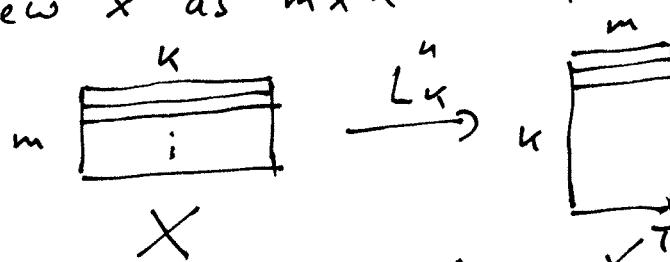
$$\begin{pmatrix} y \\ \vdots \end{pmatrix} = \begin{pmatrix} A & & & \\ & A & & \\ & & A & \\ & & & \ddots & \cdots \end{pmatrix} \begin{pmatrix} x \\ \vdots \end{pmatrix}$$

m A's at stride m

for $i = 0 : m-1$
 $y[i:m:i+(k-1)m] = A \cdot x \quad "$

3.) $y = L_k^n x$: different ways of viewing it

a.) view x as $m \times k$ matrix:



transposition!

b.) L_k^n "reads at stride 1 and writes at stride m "

stride 1 -> m
is the same as
stride k -> 1

c.) L_k^n performs permutation $im+j \rightarrow j^{k+i}$

$0 \leq i \leq k$
 $0 \leq j \leq n$

FFT again:

$$\mathcal{DFT}_{km} = \underbrace{(\mathcal{DFT}_k \otimes I_m)}_{\text{stride } m \rightarrow m} \overline{T}_m \underbrace{(\overline{I}_n \otimes \mathcal{DFT}_m)}_{\text{stride } 1 \rightarrow 1} \underbrace{L_k^n}_{\text{stride } 1 \rightarrow m}$$

stride 1 -> m
is the same as
stride k -> 1

this is the "decimation-in-time" version

Decomposition in frequency: transpose!

Use: - DFT is symmetric

$$- (L_n^m)^T = L_m^n$$

$$- (A \otimes B)^T = A^T \otimes B^T$$

Gives:

$$DFT_{un} = L_m^n (I_n \otimes DFT_m) T_m^n (DFT_n \otimes I_m)$$

Cost analysis: (was in exam for radix 2), assume $n=2$

Measure: (complex adds, complex mults)

Cost: independent of radix ($n \log_2(n)$, $\frac{1}{2}n \log_2(n)$)

$$\begin{aligned} \text{complex add} &= 2 \text{ real adds} \\ \text{"mult"} &\leq 4 \text{ real multi} \\ &\quad 2 \text{ real adds} \end{aligned}$$

$$\Rightarrow \text{real cost} \leq 2n \log_2(n) + 3n \log_2(n) = 5n \log_2(n)$$

Iterative radix-2 FFT

$$DFT_2^t = R_2^t \prod_{i=1}^t \underbrace{(I_{2^{t-i}} \otimes \overline{T}_{2^{t-i}})}_{\substack{\text{bit-reversal} \\ \text{permutation}}} \underbrace{(I_{2^{t-i}} \otimes DFT_2 \otimes I_{2^{t-i}})}_{\substack{\text{diagonal} \\ \text{matrix} \\ 2^{t-1} DFT_2's \\ \text{at varying scales}}}$$

Most people consider this "the FFT"

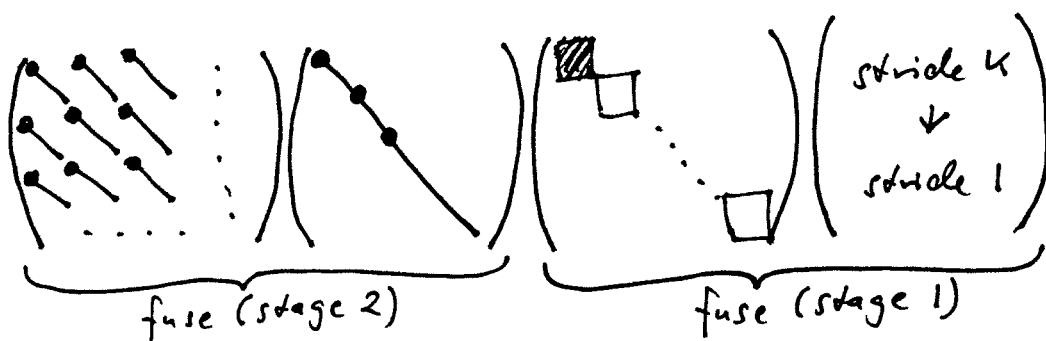
FFT, fast implementation (following Tewar &c)

1.) Choice of algorithm: Choose recursive FFT, not iterative FFT

2.) Locality optimization:

$$\mathcal{DFT}_{km} = (\mathcal{DFT}_k \otimes I_m) T_m^{km} (I_k \otimes \mathcal{DFT}_m) L_k^{km}$$

(schematic) =



compute m many

$$\mathcal{DFT}_k \cdot D$$

part of
diagonal T_m^{km}

at stride m
(input and output)

- writes to the same location it reads from
- inplace

$\mathcal{DFTscaled}(k, *x, *d, \text{stride})$

size
input diagonal
= output elements
vector

this interface cannot handle
arbitrary recursions
→ in FFTW a base case



compute K many \mathcal{DFT}_m
with input stride k and
output stride 1.

- writes to different locations
if reads from
- out-of-place

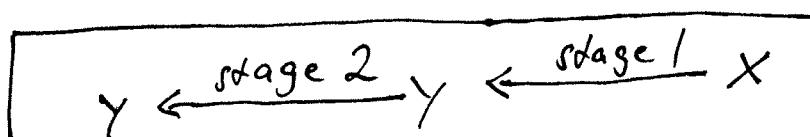
$\mathcal{DFTrec}(m, *x, *y, \text{instride}, \text{outstride})$

size
input output
vector vector

this interface can handle
arbitrary recursions

Pseudo code:

```
DFT(n, x, y) = DFTrec(n, x, y, 1, 1)
for (int i = 0; i < k; ++i)
    DFTrec(m, y + m*i, x + i, k, 1); // implemented as DFT(...) is
for (int j = 0; j < m; ++j)
    DFTscaled(k, y + j, t[j], m); // always a base case
    ↑
    precomputed twiddles
```



3.) Constants:

The matrix T_m^{km} yields multiplications by constants:
 $y_i = \omega_m^k x_i$
 some root of unity

which in the code, on real numbers, gives multiplications
 by sines and cosines

$$y_i = \sin\left(\frac{i\pi}{128}\right)x_i \text{ etc...}$$

Problem: Computing $\sin(\dots)$ is very expensive (HW 2)

Solution:

- precompute once

- reuse many times

- assumes a transform for one size
 is used many times

Changed library interface:

```
d = dft-plan(1024);           // precomputes constants
d(*x, *y);                   // computes DFT, size 1024
```

4.) Fast basic blocks

We do not want to recurse all the way to $n=2$

- function call overhead

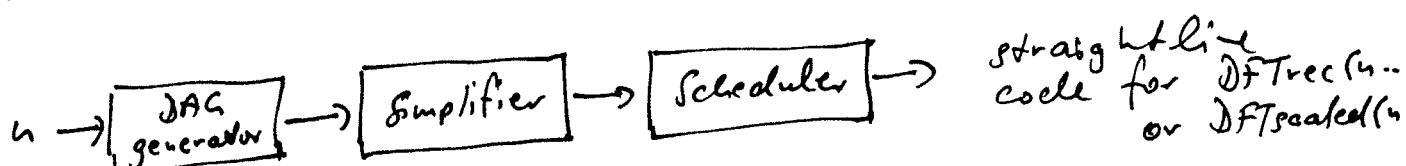
- suboptimal register use

Solution: - unroll recursion for small enough n

- practice shows $n \leq 32$ is sufficient

- requires 62 functions! Why?

FFTW: "cooley" generator for small size FFT



a.) DAG generator $\xrightarrow{\text{recursively}}$

- generates DAG from stored algorithms

- DAGs have only adds/subs/mults by const

Example:

$$\begin{array}{c} x_0 \\ x_1 \end{array} \xrightarrow{\quad \oplus \quad} \begin{array}{c} y_0 \\ y_1 \end{array} \quad \leftrightarrow \quad \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

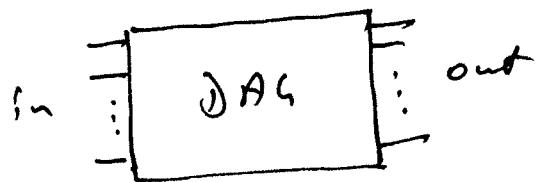
b.) Simplifier

- simplifies mults by 0, 1, -1
- distributivity law: $kx + ky = k(x+y)$
- canonicalization: $x-y, y-x \rightarrow x-y, -(x-y)$
- common subexpression elimination (CSE)
- all constants are made positive:
reduces register pressure
- CSE also on transposed DAG

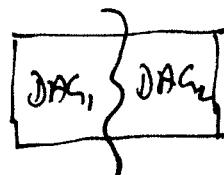
c.) Scheduler:

Theoretical result: 2-power FFT needs
 $\Omega\left(\frac{n \log(n)}{R}\right)$ register spills
 for R registers

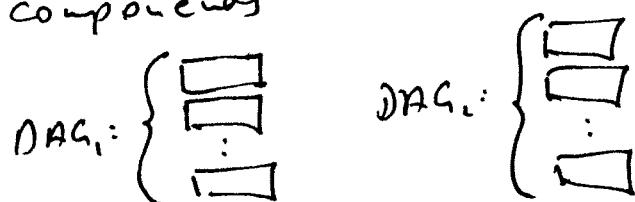
The following algorithm achieves that:



step 1: cut DAG in middle (how do we do that)



step 2: DAG_1, DAG_2 ~~are~~ decompose into independent components



schedule these recursively

Finally: output straightline, SSA code

```
/*
 * Copyright (c) 2003 Matteo Frigo
 * Copyright (c) 2003 Massachusetts Institute of Technology
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */
```

* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*

* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1
*/

/* This file was automatically generated --- DO NOT EDIT */
/* Generated on Sat Jul 5 21:29:51 EDT 2003 */

/^ Generated on Sat Jul 5 21:29:51 EDT 2003 ^/

```
#include "codelet-dft.h"
```

```
/* Generated by: /homee/stevenj/cvs/fftw3.0.1/genfft/gen_twiddle -compact -variables 4 -n 3 -name t1_3 -include t.h */
```

```
/*
 * This function contains 16 FP additions, 12 FP
 * (or, 10 additions, 6 multiplications, 6 fused m
 * 15 stack variables, and 12 memory accesses
```

```
/*
 */
/* Generator Id's :
 * $Id: algsimp.ml,v 1.7 2003/03/15 20:29:42 stevenj Exp $
 * $Id: fft.ml,v 1.2 2003/03/15 20:29:42 stevenj Exp $
 * $Id: gen_twiddle.ml,v 1.16 2003/04/16 19:51:27 athena Exp $
 */
```

```
#include "t.h"
```

```
static const R *t1_3(R *ri, R *ji, const R *W, stride jos, int m, int dist)
```

```

static const R = TV_S(R - R, R - R, const
{
    DK(KP866025403, +0.86602540378
    DK(KP5000000000, +0.500000000000
    int i;
    for (i = m; i > 0; i = i - 1, ri = ri +
        E T1, Ti, T6, Te, Tb, Tf, Tc, Th,
        T1 = ri[0];
        Ti = ii[0];
        {
            E T3, T5, T2, T4;
            T3 = ri[WS(ios, 1)];
            T5 = ii[WS(ios, 1)];
            T2 = W[0];

```

```

T4 = W[1];
T6 = FMA(T2, T3, T4 * T5);
Te = FNMS(T4, T3, T2 * T5);
}
{
E T8, Ta, T7, T9;
T8 = ri[WS(ios, 2)];
Ta = ii[WS(ios, 2)];
T7 = W[2];
T9 = W[3];
Tb = FMA(T7, T8, T9 * Ta);
Tf = FNMS(T9, T8, T7 * Ta);
}
Tc = T6 + Tb;
Th = Te + Tf;
ri[0] = T1 + Tc;
ii[0] = Th + Ti;
{
E Td, Tg, Tj, Tk;
Td = FNMS(KP500000000, Tc, T1);
Tg = KP866025403 * (Te - Tf);
ri[WS(ios, 2)] = Td - Tg;
ri[WS(ios, 1)] = Td + Tg;
Tj = KP866025403 * (Tb - T6);
Tk = FNMS(KP500000000, Th, Ti);
ii[WS(ios, 1)] = Tj + Tk;
ii[WS(ios, 2)] = Tk - Tj;
}
}
return W;
}

static const tw_instr twinstr[] = {
{TW_FULL, 0, 3},
{TW_NEXT, 1, 0}
};

static const ct_desc desc = { 3, "t1_3", twinstr, {10, 6, 6, 0}, &GENUS, 0, 0, 0 };

void X(codelet_t1_3) (planner *p) {
    X(kdft_dit_register) (p, t1_3, &desc);
}

```