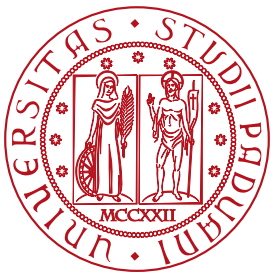
Calcolo Parallelo – Project

Lecture 3 – libhpc, Optimizations ,MPI example

Michele Schimd

schimdmi@dei.unipd.it

May 6th, 2019



Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Outline

Analisi delle prestazioni

HPM Toolkit [1] [2]

Cos'è?

È una libreria scritta da IBM per accedere ai **contatori hardware**

Power7 - contatori hardware

Contano eventi: cicli macchina, TLB misses, cache misses, flops, load e store...

Ma...

- I registri per i contatori sono una risorsa scarsa!
- Ogni set contiene alcuni degli eventi che possono essere contati
- Alcune misure non sono ottenibili direttamente (es. *flops*)

[1] http://goo.gl/0iyMf3
[2] http://goo.gl/ZIomIZ

```
#include <libhpc.h>
...
hpmInit( taskID, "myprogram" );
hpmStart( 1, "work1+work2" );
do_work1();
hpmStart( 2, "work2" );
do_more_work();
hpmStop( 2 );
hpmStop( 1 );
hpmTerminate( taskID );
...
```

NOTE

- In compilazione, con compilatori IBM, va aggiunto:
  `-I/usr/lpp/ppe.hpct/include -L/usr/lpp/ppe.hpct/lib -lhpc -lpmapi`
- Al termine crea i file `myprogram_taskID_tid` con i risultati
- La variabile d'ambiente `HPM_EVENT_SET` seleziona il set di eventi

Struttura a parentesi!

Giusto

```
hpmStart(1,"primo");
...
hpmStop(1);
hpmStart(2,"secondo");
hpmStart(3,"terzo");
...
hpmStop(3);
...
hpmStop(2);
```

Sbagliato

```
hpmStart(1,"primo");
...
hpmStop(1);
hpmStart(2,"secondo");
hpmStart(3,"terzo");
...
hpmStop(2);
...
hpmStop(3);
```

Outline

Siano date tre matrici dense $\mathbf{A}$, $\mathbf{B}$ e $\mathbf{C} \in \mathbb{R}^{m \times m}$

Vogliamo calcolare $\mathbf{C} \leftarrow \mathbf{A} \times \mathbf{B}$ utilizzando n processi di calcolo (supponiamo m divisibile per n)

Occorre definire:

1. Il formato di input/output
2. L'algoritmo parallelo:
   - Come distribuire i dati tra i processi
   - Cosa fa ciascun processo
   - Quali dati occorre scambiare

Algoritmo naïve

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j}$$

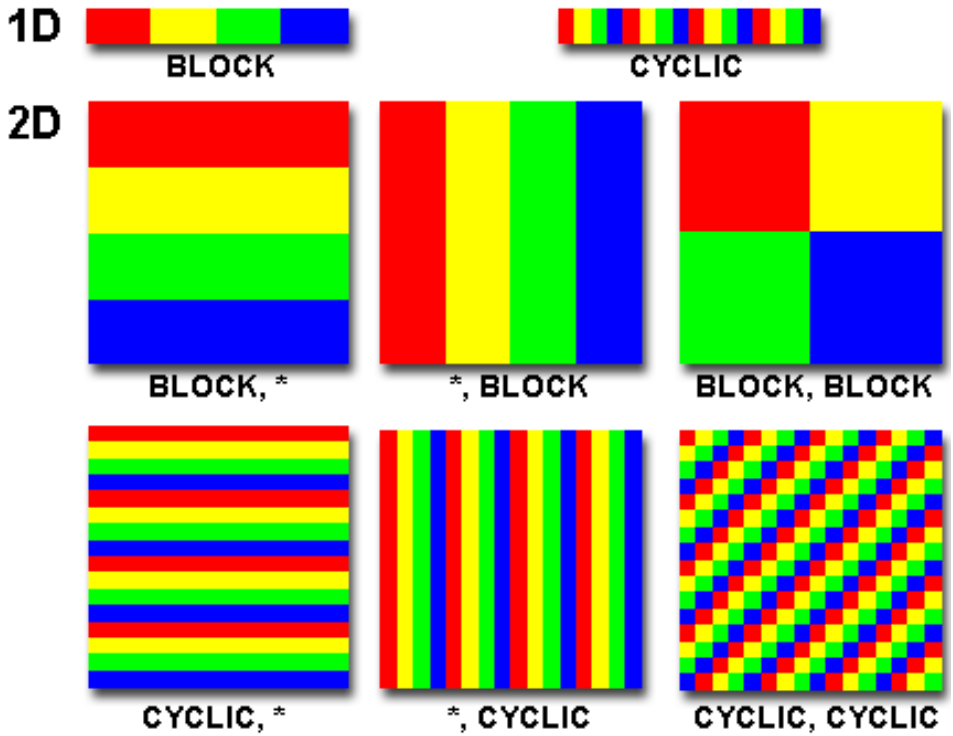
Tempo seriale $\mathcal{O}(m^3)$

Moltiplicazione di matrici
Distribuzione dei dati

La scelta di come distribuire i dati tra i processi dipende da:

- Eventuali **vincoli** sul formato in input/output
- Proprietà dell'algoritmo
- **Tradeoff** tra comunicazione e replicazione
- Semplicità implementativa

Esempi di distribuzione di vettori/matrici tra 4 processi:



Master / Slave

Distribuire i dati e raccogliere i risultati

1. Si designa un processo master ad esempio il processo 0 su `MPI_COMM_WORLD`.
2. Il master carica da file i dati ($\mathbf{A}$ e $\mathbf{B}$ nella moltiplicazione di matrici).
3. Il master suddivide e distribuisce agli altri processi slave i dati utilizzando le primitive MPI opportune.
4. Tutti i processi (master e slave) eseguono l'algoritmo parallelo il quale può prevedere scambi intermedi di risultati che *non debbono necessariamente passare attraverso il master*.
5. Al termine della computazione ogni slave spedisce al master i risultati.
6. Al termine della computazione il master recupera tutti i risultati, costruisce e salva sul file l'output (la matrice $\mathbf{C}$ nella moltiplicazione di matrici).

Vediamo qualche esempio per MM

Distribuzione 2D a blocchi su righe di $\mathbf{A}$ e $\mathbf{C}$ (con replicazione)

- Il processo p ottiene $\mathbf{A}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- La matrice $\mathbf{B}$ viene replicata
- Calcola $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$

Master $p = 0$

1. Legge $\mathbf{A}$ e $\mathbf{B}$ da un file
2. Distribuisce $\mathbf{A}(\mathbf{p})$ al processo p
3. Replica $\mathbf{B}$ in ciascun processo
4. Calcola $\mathbf{C}(0) \leftarrow \mathbf{A}(0) \times \mathbf{B}$
5. Raccoglie $\mathbf{C}(\mathbf{p})$ dal processo p
6. Scrive $\mathbf{C}$ in un file

Distribuzione 2D a blocchi su righe di $\mathbf{A}$ e $\mathbf{C}$ (con replicazione)

- Il processo p ottiene $\mathbf{A}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- La matrice $\mathbf{B}$ viene replicata
- Calcola $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$

Master $p = 0$

1. Legge $\mathbf{A}$ e $\mathbf{B}$ da un file
2. Distribuisce $\mathbf{A}(\mathbf{p})$ al processo p (**MPI_Scatter**)
3. Replica $\mathbf{B}$ in ciascun processo (**MPI_Bcast**)
4. Calcola $\mathbf{C}(0) \leftarrow \mathbf{A}(0) \times \mathbf{B}$
5. Raccoglie $\mathbf{C}(\mathbf{p})$ dal processo p (**MPI_Gather**)
6. Scrive $\mathbf{C}$ in un file

Distribuzione 2D a blocchi su righe di $\mathbf{A}$ e $\mathbf{C}$ (con replicazione)

- Il processo p ottiene $\mathbf{A}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- La matrice $\mathbf{B}$ viene replicata
- Calcola $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$

Slave $p > 0$

1. ...
2. Ottiene $\mathbf{A}(\mathbf{p})$ distribuito dal master
3. Ottiene $\mathbf{B}$ replicato dal master
4. Calcola $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$
5. Invia $\mathbf{C}(\mathbf{p})$ al master che lo raccoglie
6. ...

Distribuzione 2D a blocchi su righe di $\mathbf{A}$ e $\mathbf{C}$ (con replicazione)

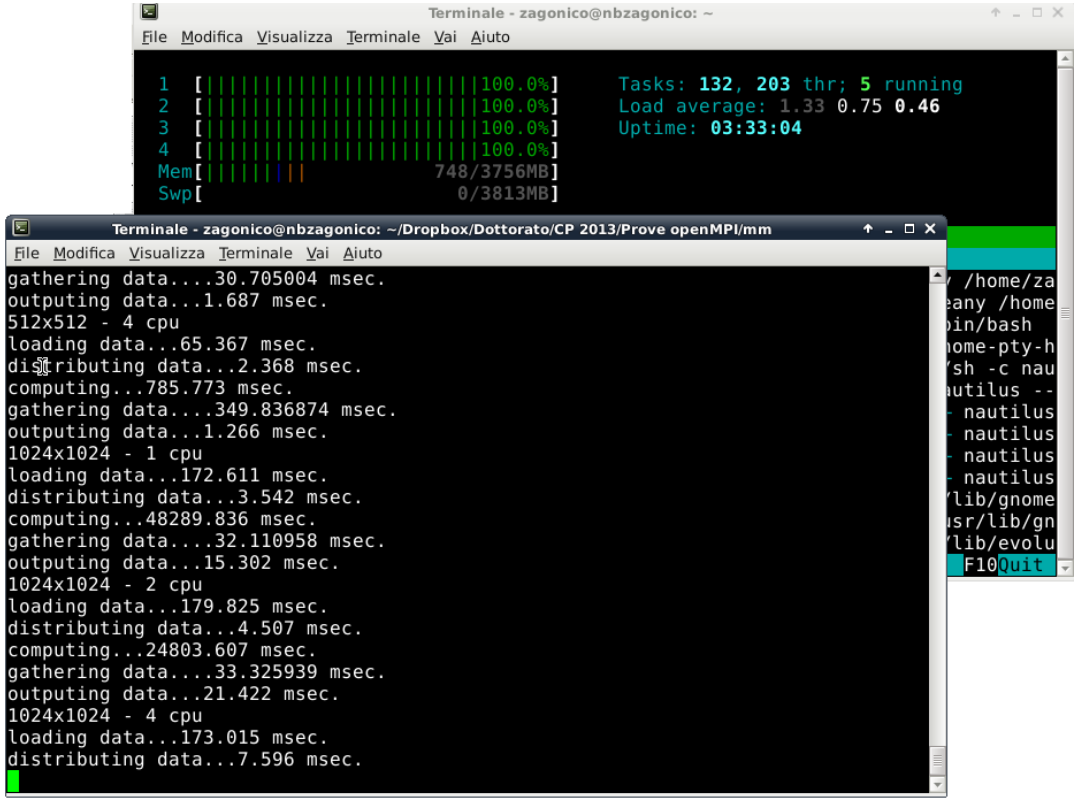
- Il processo p ottiene $\mathbf{A}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- La matrice $\mathbf{B}$ viene replicata
- Calcola $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$

Slave $p > 0$

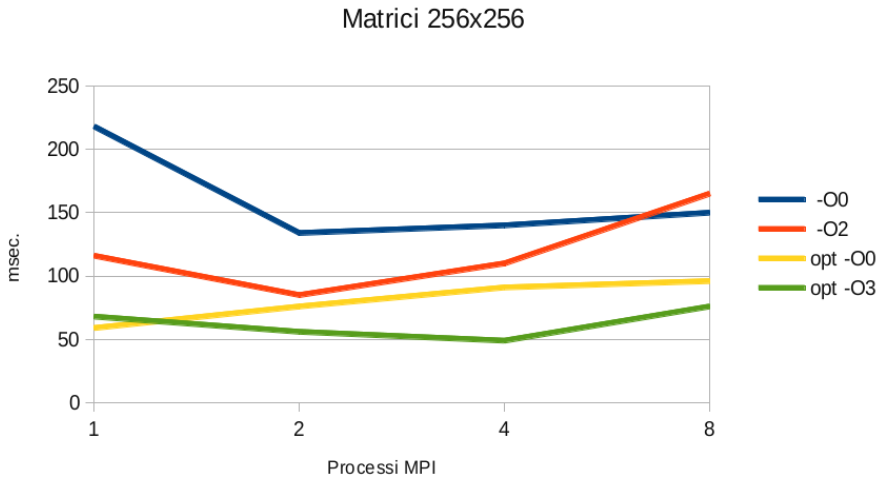
1. ...
2. Ottiene $\mathbf{A}(\mathbf{p})$ distribuito dal master (**MPI_Scatter**)
3. Ottiene $\mathbf{B}$ replicato dal master (**MPI_Bcast**)
4. Calcola $\mathbf{C}(p) \leftarrow \mathbf{A}(p) \times \mathbf{B}$
5. Invia $\mathbf{C}(\mathbf{p})$ al master che lo raccoglie (**MPI_Gather**)
6. ...

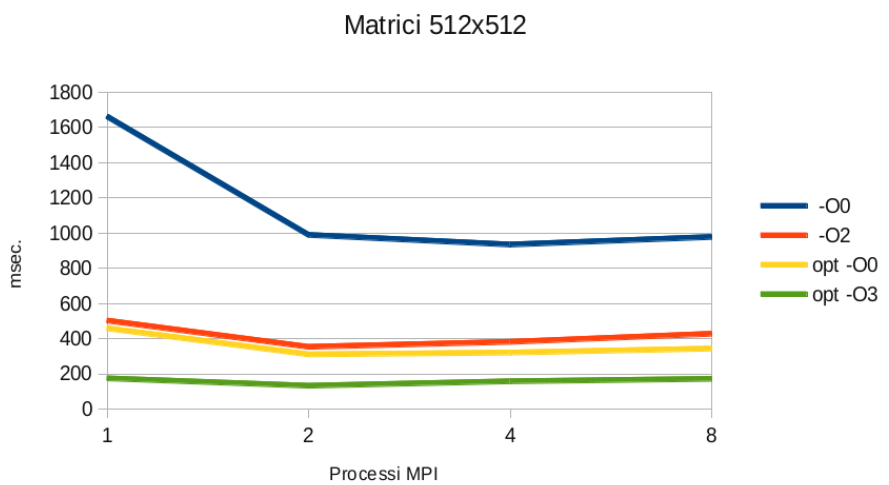
Esempi di prove fatte su un Intel i5 2.5 GHz (2 core reali - 4 core virtuali)

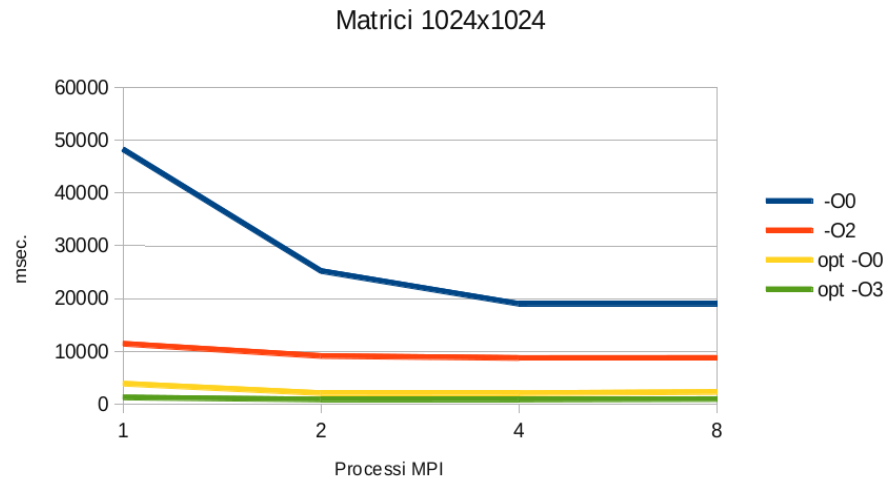
- `MPI_Wtime` usato come cronometro
- comportamento all'aumentare delle istanze
- comportamento all'aumentare delle cpu
- metodo "rozzo": su Power7 saranno disponibili librerie apposite per leggere i contatori delle CPU



Tempi di esecuzione totali







Trova le differenze:

```
void mult1(float **A, float **B, \
    float **C, int l, int n) {
  for (int i=0; i<l; i++)
    for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

```
void mult_opt(float **Apart, float **B, float **Cpart, \
      int l, int n) {
  float *T = malloc(n*n*sizeof(float));
  float *ma, *mb;
  int i1,i2;
  int n2=n<<1, n3= n*3, n4 = n<<2;
  float *Bf=B[0];
  i1=0;                    // first: transpose B
  for (int i=0; i<n; i++) {     // m2 b x c, T c x b
    i2=i;
    for (int j=0; j<n; j+=4, i2+=n4 , i1+=4) {
      T[i2] = Bf[i1];
      T[i2+n] = Bf[i1+1];
      T[i2+n2] = Bf[i1+2];
      T[i2+n3] = Bf[i1+3];
    }
  }
  float *row_a, temp;
  float t1,t2,t3,t4;      // C = A*B
  for (int i=0; i<l; i++) {
    mb = T;
    row_a = Apart[i];
    for (int j=0; j<n; j++, mb+=n) {
      temp = 0;
      for (int k=0; k<n; k+=4) {
        t1 = row_a[k] * mb[k];
        t2 = row_a[k+1] * mb[k+1];
        t3 = row_a[k+2] * mb[k+2];
        t4 = row_a[k+3] * mb[k+3];
        temp += (t1+t2)+(t3+t4);
      }
      Cpart[i][j] = temp;
    }
  }
  free(T);
}
```

Nota bene nel caso 1024x1024!

	1 cpu	4 cpu
impl. banale -O0:	48278 msec.	19007 msec.
impl. banale -O2:	11443 msec.	8748 msec.
impl. opt -O0:	3650 msec.	2086 msec.
impl. opt -O2:	1275 msec.	881 msec.

- non ha senso dire che un algoritmo scala bene/male analizzandone una versione non ottimizzata
- idem se si utilizza con istanze piccole. Le misurazioni sono sfalsate da processi in background ecc.
- in questo esempio anche la versione 1024x1024 rischia di essere rumorosa per la versione ottimizzata
- il codice NON andrà messo nella tesina, ma allegato come già stabilito (sorgente, Makefile, . . .)

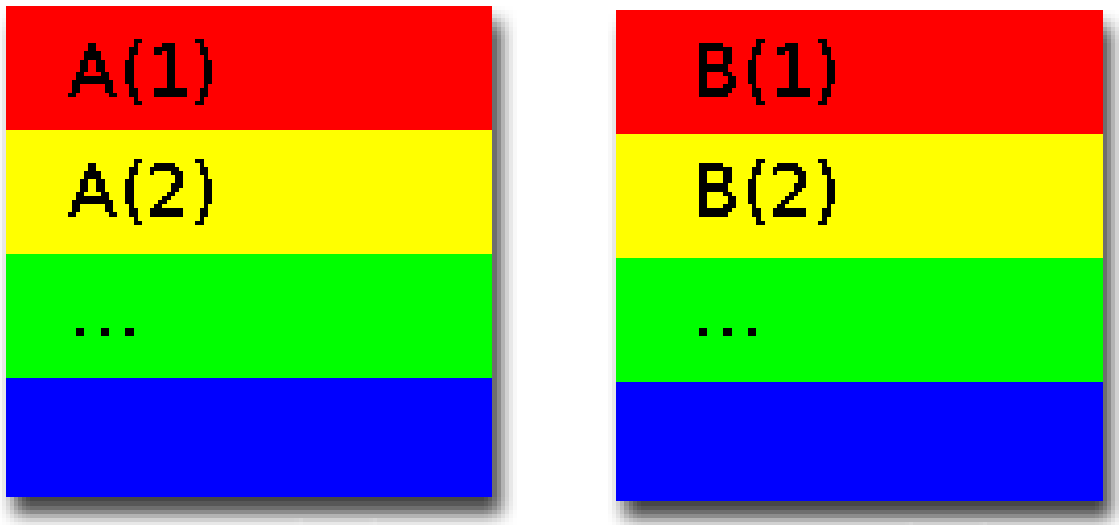
- Tempo totale = lettura da disco della matrice + scattering A + broadcast B + computazione + gathering C + salvataggio su disco di C
- Comunicazioni = scattering + broadcast + gathering
- Lavoro = computazione locale dei processi
- Lavoro “utile” = Comunicazioni + Lavoro
- Ignoro tempi per caricare/salvare su disco.
- Analisi di impatto comunicazioni. Come scalano comunicazioni? Come scala lavoro?

$$\frac{\text{Comunicazioni}}{\text{Lavoro Utile}} \qquad \frac{\text{Lavoro}}{\text{Lavoro Utile}}$$

- Verificate differenze tra -O0, -O2, -O3...
- ...nella tesina usate *algoritmi ottimi e ottimizzati*.
- Analisi all'aumentare delle dimensioni del problema.
- Istanze grandi (altrimenti risultati "rumorosi")
- Analisi all'aumentare dei processi $(1, 2, 4, 8 \ldots 48)$: che speedup si ottengono? quando satura il parallelismo?
- Misurazioni con libreria IBM (ha accesso a contatori flop ecc. della CPU)
- Considerazioni su come programma interagisce con l'hardware circostante

Distribuzione 2D a blocchi su righe di $\mathbf{A}$, $\mathbf{B}$ e $\mathbf{C}$

- Il processo p ottiene $\mathbf{A}(p), \mathbf{B}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- Calcola $\mathbf{C}(p) \leftarrow \sum_q \mathbf{A}(p,q) \times \mathbf{B}(q)$
- Il processo p necessita di $\mathbf{B}(q)$ durante il calcolo di $\mathbf{C}(p)$, $\forall q \neq p$



Ogni processo memorizza solo 2 blocchi di $\mathbf{B}$ alla volta

Distribuzione 2D a blocchi su righe di $\mathbf{A}$, $\mathbf{B}$ e $\mathbf{C}$

- Il processo p ottiene $\mathbf{A}(p), \mathbf{B}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- Calcola $\mathbf{C}(p) \leftarrow \sum_q \mathbf{A}(p,q) \times \mathbf{B}(q)$
- Il processo p necessita di $\mathbf{B}(q)$ durante il calcolo di $\mathbf{C}(p)$, $\forall q \neq p$

Per ogni $q = 0 \ldots n-1$

1. Se $p = q$ distribuisco $\mathbf{B}(p)$ a $\forall q \neq p$
2. Se $p \neq q$ ottengo $\mathbf{B}(q)$ da q
3. Calcola $\mathbf{C}(p) \leftarrow \mathbf{C}(p) + \mathbf{A}(p,q) \times \mathbf{B}(q)$

Ogni processo memorizza solo 2 blocchi di $\mathbf{B}$ alla volta

Distribuzione 2D a blocchi su righe di $\mathbf{A}$, $\mathbf{B}$ e $\mathbf{C}$

- Il processo p ottiene $\mathbf{A}(p), \mathbf{B}(p) \in \mathbb{R}^{\frac{m}{n} \times m}$
- Calcola $\mathbf{C}(p) \leftarrow \sum_q \mathbf{A}(p,q) \times \mathbf{B}(q)$
- Il processo p necessita di $\mathbf{B}(q)$ durante il calcolo di $\mathbf{C}(p)$, $\forall q \neq p$

Per ogni $q = 0 \ldots n-1$

1. Se $p = q$ distribuisco $\mathbf{B}(p)$ a $\forall q \neq p$ (**MPI_Bcast**)
2. Se $p \neq q$ ottengo $\mathbf{B}(q)$ da q (**MPI_Bcast**)
3. Calcola $\mathbf{C}(p) \leftarrow \mathbf{C}(p) + \mathbf{A}(p,q) \times \mathbf{B}(q)$

Nel complesso ogni processo riceve tutta la matrice $\mathbf{B}$

Outline

Nota

Nota

Parlerò di ottimizzazioni in generale, non solo in riferimento al progetto

- alcuni esempi sono per ambiente Linux e macchine AMD/Intel, non funzioneranno su AIX con processore Power7
- le tecniche vanno riadattate all'architettura in cui ci si trova
- **non** usate nel progetto **tutte** le cose che vedremo, cercate solo di capire i principi sottostanti
- nel progetto probabilmente vi serviranno solo alcuni sottoinsiemi di quanto vedremo.

Quattro Principi

Principio di Pareto (o legge 80-20)

L'80% degli effetti è dovuto al 20% delle cause

- L'80% dell'ottimizzazione è dovuto a un 20% di trucchi
- L'80% delle performance finali è ottenuto con un 20% di sforzo
- Vale la pena spendere **4 volte** più tempo e fatica per un miglioramento del 25%?

Clever Man Principle

Sfruttate i flag di ottimizzazione del compilatore

Il rasoio di Occam

Entia non sunt multiplicanda praeter necessitatem[a]

[a] Non moltiplicare gli elementi più del necessario

Wheel Reinvention Principle

Riutilizzate il più possibile codice già ottimizzato

Flag del compilatore

Segue il *Clever Man* Principle

GNU compiler

-Ox abilita ottimizzazioni indipendenti dalla macchina (es. -O2)

-fx abilita/disabilita ottimizzazioni indipendenti dalla macchina (es. `-funroll-loops`, `-funsafe-math-optimization`)

-mx abilita/disabilita ottimizzazioni dipendenti dalla macchina

-mtune=cpu-type ottimizza senza usare un ISA specifico (*Instruction Set Architecture*)

-march=cpu-type ottimizza usando un ISA specifico

-mcpu=cpu-type ottimizza per il modello di CPU specificato

Documentazione

Con `man gcc` avete a disposizione l'elenco e la descrizione completa dei flag supportati nella vostra installazione.

IBM compiler (`xlc`, ...)

- fa ottimizzazioni migliore di `gcc` su architetture IBM
- non supporta ISA non presenti in processori IBM
- flag analoghi a `gcc`, vedi `man xlc`...

Intel compiler (`icc`)

- molto buono su processori Intel, in AMD sembra ignorare alcuni ISA
- utilizza ISA presenti in CPU Intel
- flag analoghi a `gcc`, vedi `man icc`.

AMD Compiler (x86 Open64 Compiler Suite, `open64`, ...)

- supporta ISA AMD e Intel
- risultati paragonabili a `icc` su Intel
- flag analoghi a `gcc`, vedi la sua documentazione

Altro sui compilatori

Cosa può fare un compilatore

- prefetch dei dati
- riorganizzare cicli
- loop unrolling
- vettorizzazione automatica dei cicli
- allineare i dati ai memory boundary
- ...

Altre tecniche per compilatori

- Usare direttive compiler-specific, per facilitarne il lavoro (`#pragma`)
- Scrivere parti in assembly (`asm { }`)
- Usare intrinsics (funzioni particolari, trattate diversamente da compilatore a compilatore, che permettono al compilatore di generare codice particolarmente ottimizzato)

Usare Librerie Ottimizzate

Segue il *Wheel Reinvention* Principle

Librerie Matematiche

- BLAS & LAPACK, FFT, random number generation...
- ATLAS, Goto, Spiral, FFTW, ...
- Librerie proprietarie (AMD CML, IBM ESSL, Intel MKL)

Librerie per altre applicazioni

- Image/Signal Processing, String Processing, Compression, Cryptography...
- Librerie Open Source
- Librerie proprietarie (AMD PL, Intel IPP)

Template Library

- strutture dati utili, containers, iterators, algorithms, ...
- C++ Standard Library
- Boost C++, ...

Nota

Consideriamo C/C++, ma molti valgono in generale per ogni linguaggio

Tipi di dati fondamentali numerici:

- a seconda della piattaforma hardware possono esserci sensibili differenze nelle performance di diverse operazioni
- operazioni intere sono più veloci di quelle floating-point
- `float` sono più performanti dei `double`
- specifica `f` a fine di un floating-point perchè il compilatore lo consideri `float` (`3.14159f`)
- moltiplicazioni più veloci delle divisioni (meglio `17.0f * (1/213.0f)` che `17.0f / 213.0f`)
- somme più veloci di moltiplicazioni
- negli interi, operatori bitwise più veloci di operatori aritmetici
- estrarre sotto-espressioni comuni, specie se costose

Nelle **istruzioni condizionali**:

- mettere per primi i casi più probabili negli `switch`
- mettere prima i casi più probabili negli if annidiati
- sfruttare i corto-circuiti nelle condizioni
- sfruttare operazioni bitwise anzichè logiche
- evitare post-pre incremento nelle condizioni
- usare array anzichè switch o if per selezionare valori

```
if (flag==0)
    ris='a';
else if (flag==1)
    ris='b';
else if (flag==2)
    ris='c';
else

...
```

```
switch (flag) {
    case 0:  ris='a'
    break;
    case 1:  ris='b'
    break;
    case 2:  ris='c'
    break;

...
```

```
char *value="abcde";
...
ris=value[flag];
```

- **binary breakdown**:

```
if(a==1) {
} else if(a==2) {
} else if(a==3) {
} else if(a==4) {
} else if(a==5) {
} else if(a==6) {
} else if(a==7) {
} else if(a==8) {

}
```

```
if(a<=4) {
    if(a==1) {
    } else if(a==2) {
    } else if(a==3) {
    } else {
    }
}
else {
    if(a==5) {
    } else if(a==6) {
    } else if(a==7) {
    } else if(a==8) {
    }

}
```

da utilizzare ricorsivamente, così ho un numero logaritmico di condizioni da testare anzichè lineare.

Chiamate di funzioni:

- evitare puntatori a funzione
- non ritornare valori se non servono
- dichiarare `static` le funzioni che non sono membri dei una classe
- prototype all functions
- evitare virtual functions e virtual inheritance

Look-up Table (es. conversioni cromatiche (RGB-YUV...), CRC...)

```
y=(int)( 0.299f*r + 0.587f*g + 0.114f*b);
u=(int)(-0.147f*r - 0.289f*g + 0.436f*b);
v=(int)( 0.615f*r - 0.515f*g - 0.1*b);

y=( 66*r + 129*g + 25*b + 128)»8+16;
u=(-38*r - 74*g + 112*b)»8+128;
v=(112*r - 94*g - 18*b)»8+128;
```

```
// lookup table per y,u,v
// 9 array (2.25 KB)
...
y = y_r[r] + y_g[g] + y_b[b];
u = u_r[r] + u_g[g] + u_b[b];
v = v_r[r] + v_g[g] + v_b[b];
```

Variabili e **argomenti di funzioni**:

- dichiarare `const` gli argomenti non modificati
- variabili globali possono prevenire ottimizzazioni locali
- passare e ritornare puntatori di struct/union/classi, o passarle per riferimento
- passare/ritornare tipi fondamentali non aggregati per valore piuttosto che riferimento
- argomenti usati più di frequente a sinistra

Macro e **funzioni inline**:

- funzioni corte, semplici e chiamate spesso
- constanti usate in switch, for, if...
- `inline` da C99
- Svantaggio: allungano codice, scarsa I-cache efficency

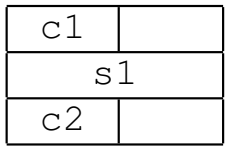
Allocazione dinamica di memoria

- minimizza i `malloc`
- forza allineamento array (`posix_memalign`, ...)
- `free`, cerca eventuali memory leaks...

Structures

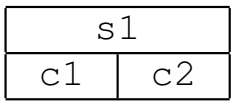
- dichiara prima i membri più grandi (favorisce packing dei dati)
- dichiarare vicini membri usati frequentemente assieme
- dimensioni multiple di 4-8-16 byte

```
struct prova {
    char c1;
    short s1;
    char c2;
};
```



tot: 6 bytes

```
struct prova2 {
    short s1;
    char c1;
    char c2;
};
```



tot: 4 bytes

Variabili:

- preferire variabili locali
- dichiarale vicino al punto di utilizzo
- le variabili globali `static` piuttosto che `extern`
- quando serve una variabile globale, conviene farne una copia locale su cui lavorare
- l'operatore `&` è costoso
- preferire costanti ($\rightarrow$ macro) come sentinelle per i cicli
- preferire interi delle dimensioni dei registri
- usare la precisione floating-point minima sufficiente
- `register`, `volatile`, `const`

Espressioni:

- assegna sotto-espressioni comuni a più espressioni a variabili automatiche locali
- evitare conversioni interi/floating-point implicite ed esplicite
- tenere separate aritmetica intera e floating-point
- trasformare divisioni per uno stesso denominatore in prodotti
- non gestire eccezioni (C++)
- preferire inizializzazione all'assegnamento

64-bit mode

- usa i 64 bit solo se servono
- evita di mescolare operazioni a 32 e 64 bit

Il compilatore fa già molte trasformazioni ai loop

Le nostre ottimizzazione “intelligenti” potrebbero prevenire le ottimizzazioni (molto più intelligenti) del compilatore.

- condizioni di terminazione e espressioni di incremento semplici (eventuale precalcolo)
- contare all’indietro (risparmio differenza/accesso a variabile)

```
for (i=0; i<n; i++)      for (i=n-1; i>=0; i--)      for (i=n-1; i--; )
```

- minimizzare corpo del loop
- loop unrolling
- considerare tradeoff tra memorizzare e ricalcolare
- tenere semplici le espressioni usate negli indici di array (precalcolo degli indici)
- se i cicli sono molto complicati, cercare di trasformarli a mano[3]
- Nota: i compilatori in genere riconoscono alcuni pattern di loop semplici e riescono a vettorizzarli. Sfruttare questa caratteristica.

[3] http://en.wikipedia.org/wiki/Loop_optimization

- usare funzioni di I/O di basso livello, come `open`, `close`, `read`...
- usare i propri meccanismi di bufferizzazione, usando `O_DIRECT` tra i flag con cui i apre il file

```
FILE *fi = fopen("prova.txt", "r");
int *data = malloc(max*sizeof(int));
for (i=0; i<max; i++) {
    fscanf(fi, "%d", data+i);
}
```

```
int fi = open("prova.bin",O_RDONLY |
O_DIRECT);
int *data = malloc(MAX*sizeof(int));
read(fi, data, sizeof(int)*MAX);
```

- accedere a multipli di 4K (page size)
- per input/output su disco, bufferizzare i dati e scriverli solo quando sono sufficenti per ammortizzare la latenza (es. > 128 KB)

Attenzione

Abbiamo visto una carrellata di tecniche e trucchi che potrebbero migliorare la velocità del codice:

- **Non è detto** che sia sempre necessario usarli, anzi di solito ne servono solo alcuni e solo nei metodi *core* dell'applicazione. Ricordiamo il principio del Rasoio di Occam: è inutile complicare ulteriormente il codice con ottimizzazioni che non contribuiscono al miglioramento finale.
- **Premature optimization is the root of all evil:**[a] evitare ottimizzazioni prima di finire di scrivere il programma. In genere la parte critica è solo del 3%, quindi evitare di ottimizzare (e rendere illeggibile e difficilmente debuggabile) tutto, ma capire cosa serve essere ottimizzato.
- servirsi di strumenti di debug per capire cosa non funziona
- servirsi di strumenti di profiling per capire cosa ottimizzare

[a] Donald Knuth, *Structured Programming With* `goto` *Statements*

Strategia

- Make it work.
- Make it right.
- Make it fast.

- Consideriamo la Moltiplicazione di Matrici
- Make it work/make it right:

```
void mult1(float **A, float **B, float **C, int l, int n) {
    for (int i=0; i<l; i++) {
        for (int j=0; j<n; j++) {
            for (int k=0; k<n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Esempio: Matrix Multiplication
Osservazioni

Questo è il *core* del programma, vale la pena ottimizzarlo il più possibile:

1. accedere a `C[i][j]` significa per il programma accedere a `C[i*n+j]` e quest'indice viene calcolato ogni volta
2. potrei pensare di trasporre `B`, in modo che sia `A` che `B` siano acceduti per righe (meno cache miss)
3. Nota: se anzichè eseguire i cicli nell'ordine i-j-k, sfrutto la proprietà commutativa della somma e li eseguo nell'ordine k-i-j, `A[i][k]` nel ciclo interno è costante, mentre `B` e `C` avanzano ordinatamente per righe
4. Dato 3), la 1) è ancora valida, la 2) diventa inutile.

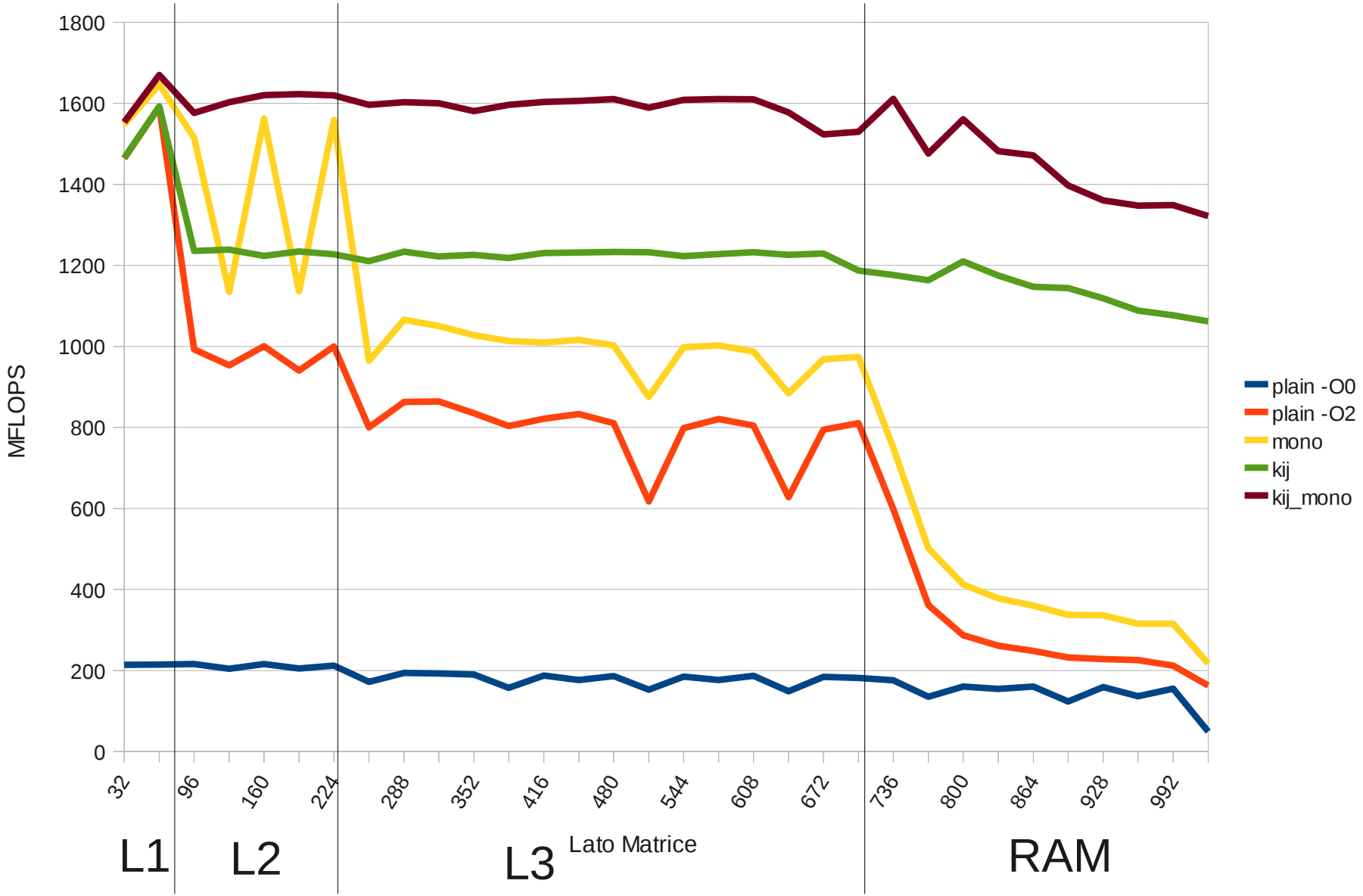
Esempio: Matrix Multiplication

Prima Ottimizzazione

```
void mult2(float **A, float **B, float **C, int l, int n) {
    float *Ct, *Bt, *At, Ac;
    for (int k=0; k<n; k++) {
        At = (A[0])+k;
        Bt = B[k];
        for (int i=0; i<l; i++) {
            Ct = C[i];
            Ac = *At;
            for (int j=0; j<n; j++) {
                Ct[j] += Ac * Bt[j];
            }
            At+=n;
        }
    }
}
```

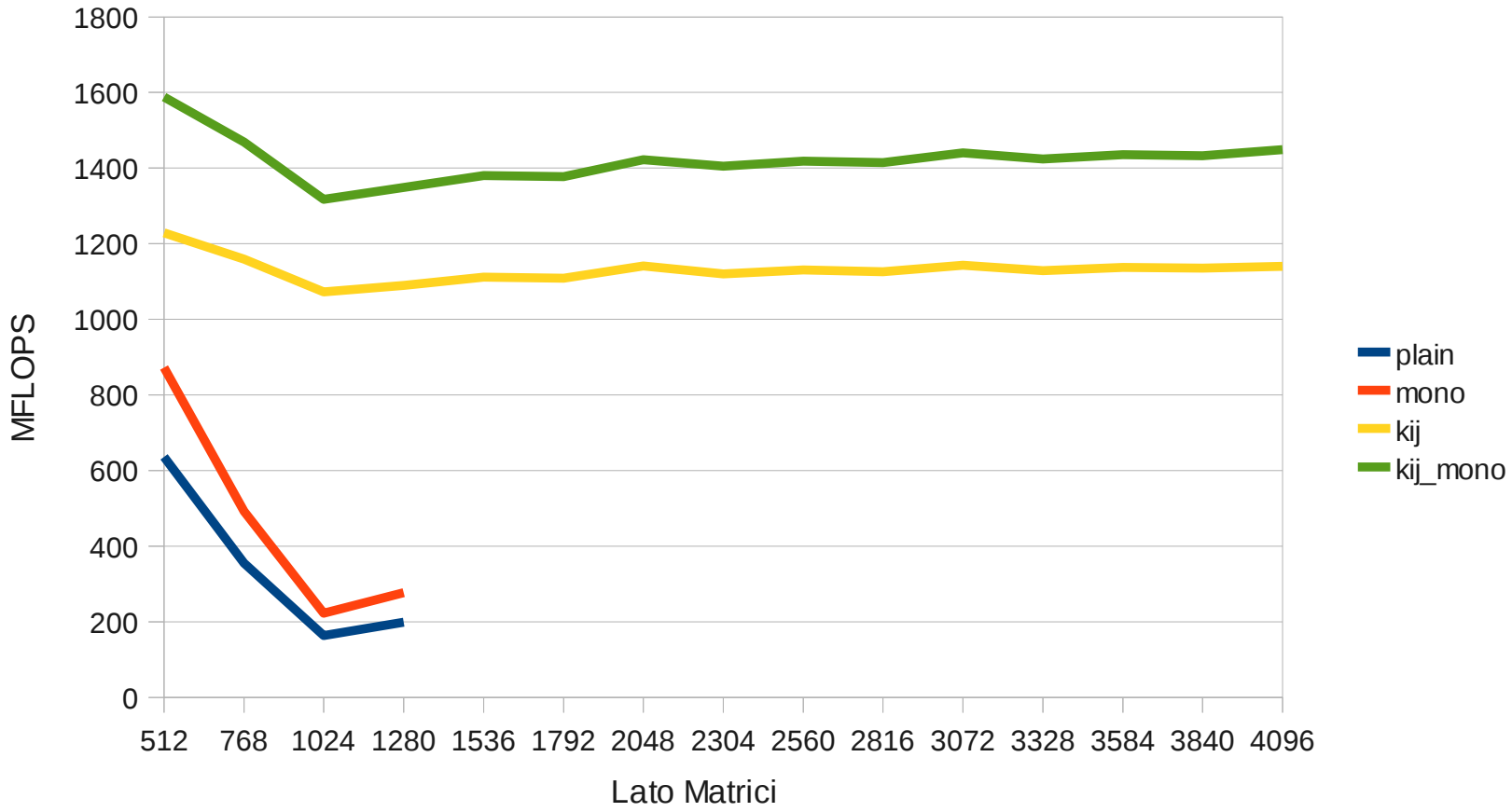
Esempio: Matrix Multiplication

Prima Ottimizzazione



Esempio: Matrix Multiplication

Prima Ottimizzazione



Legenda: **plain**: algoritmo con array bidimensionali, **mono**: algoritmo con array monodimensionali, **kij**: riorganizzazione dei cicli in k-i-j, **kij_mono**: kij+mono

- nella versione non ottimizzato è ben visibile come la velocità della memoria che contiene i dati influenza la computazione
- la riorganizzazione dei cicli da i-j-k a k-i-j permette di ottenere un codice che sfrutta sempre la L2
- possiamo fare ancora qualcosa?
- il corpo nel loop più interno è molto piccolo. Se facciamo un **unrolling manuale** del loop sfruttiamo meglio la pipeline del processore.
- possiamo suddividere i cicli in modo da lavorare con sottomatrici piccole, che possono essere contenute dalla **cache L1**.

Esempio: Matrix Multiplication

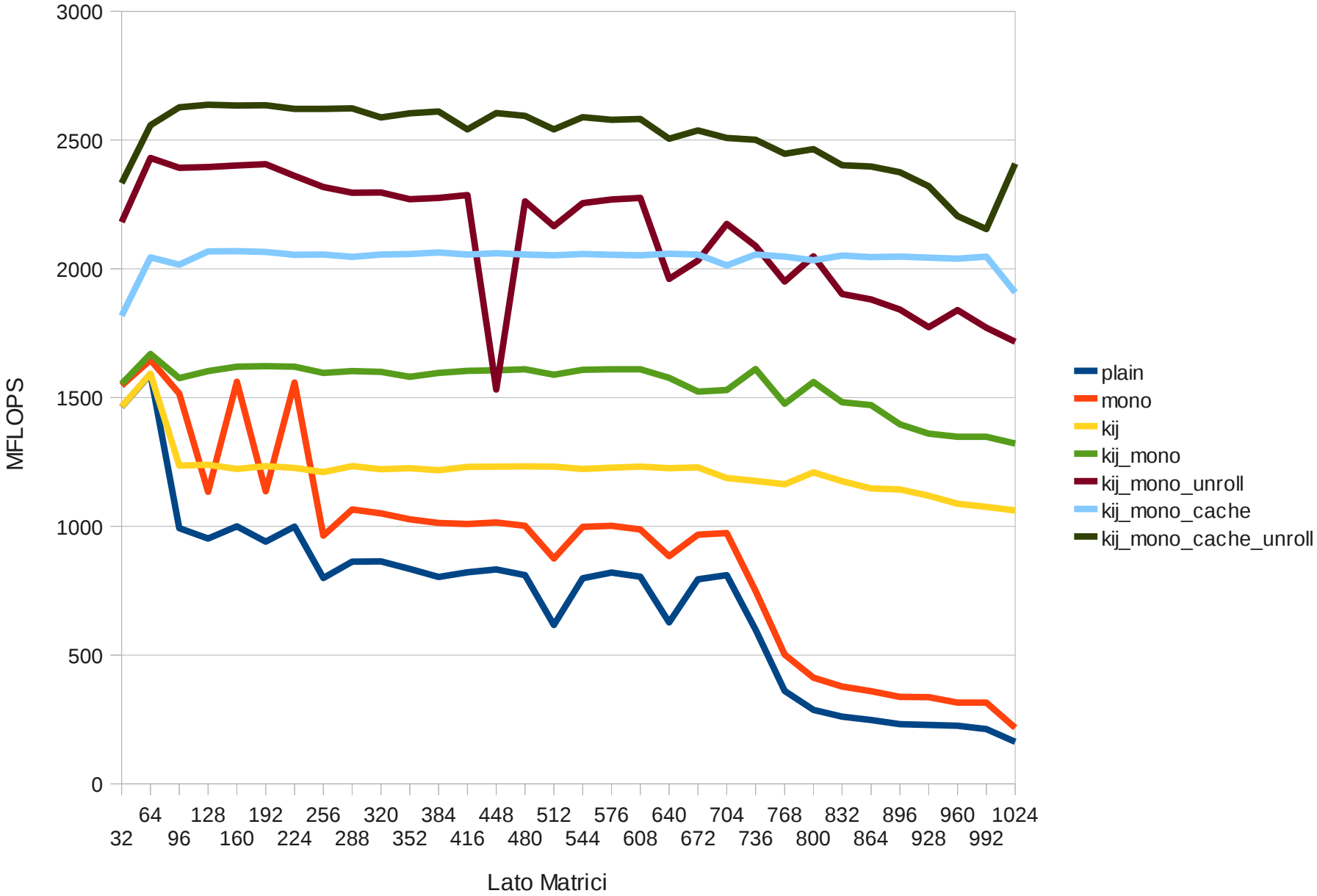
Prima Ottimizzazione

```
void mult3unroll(float **A, float **B,
                 float **C, int l, int n) {
    float *Ct, *Bt, *At, Ac;
    for (int k=0; k<n; k++) {
        At = (A[0])+k;
        Bt = B[k];
        for (int i=0; i<l; i++) {
            Ct = C[i];
            Ac = *At;
            for (int j=0; j<n; j+=8) {
                Ct[j] += Ac * Bt[j];
                Ct[j+1] += Ac * Bt[j+1];
                Ct[j+2] += Ac * Bt[j+2];
                Ct[j+3] += Ac * Bt[j+3];
                Ct[j+4] += Ac * Bt[j+4];
                Ct[j+5] += Ac * Bt[j+5];
                Ct[j+6] += Ac * Bt[j+6];
                Ct[j+7] += Ac * Bt[j+7];
            }
            At+=n;
        }
    }
}
```

```
#define CHUNK 16
void mult4(float **A, float **B, float **C, int l, int n) {
    float *At1, *Bt1, *Ct1;
    float *At2, *Bt2, *Ct2;

    for (int k=0; k<n; k+=CHUNK) {
        for (int i=0; i<l; i+=CHUNK) {
            At1 = A[i]+k;
            for (int j=0; j<n; j+=CHUNK) {
                Bt1 = B[k]+j;
                Ct1 = C[i]+j;
                for (int k1=0; k1<CHUNK; k1++, Bt1+=n, Ct1+=n) {
                    At2 = At1+k1;
                    int i2,i3;
                    for (i2=0; i2<CHUNK; i2++, At2+=n) {
                        float Ac = *At2;
                        for (i3=0; i3<CHUNK; i3++) {
                            Ct1[i3] += Ac*Bt1[i3];
                        }
                    }
                }
            }
        }
    }
}
```

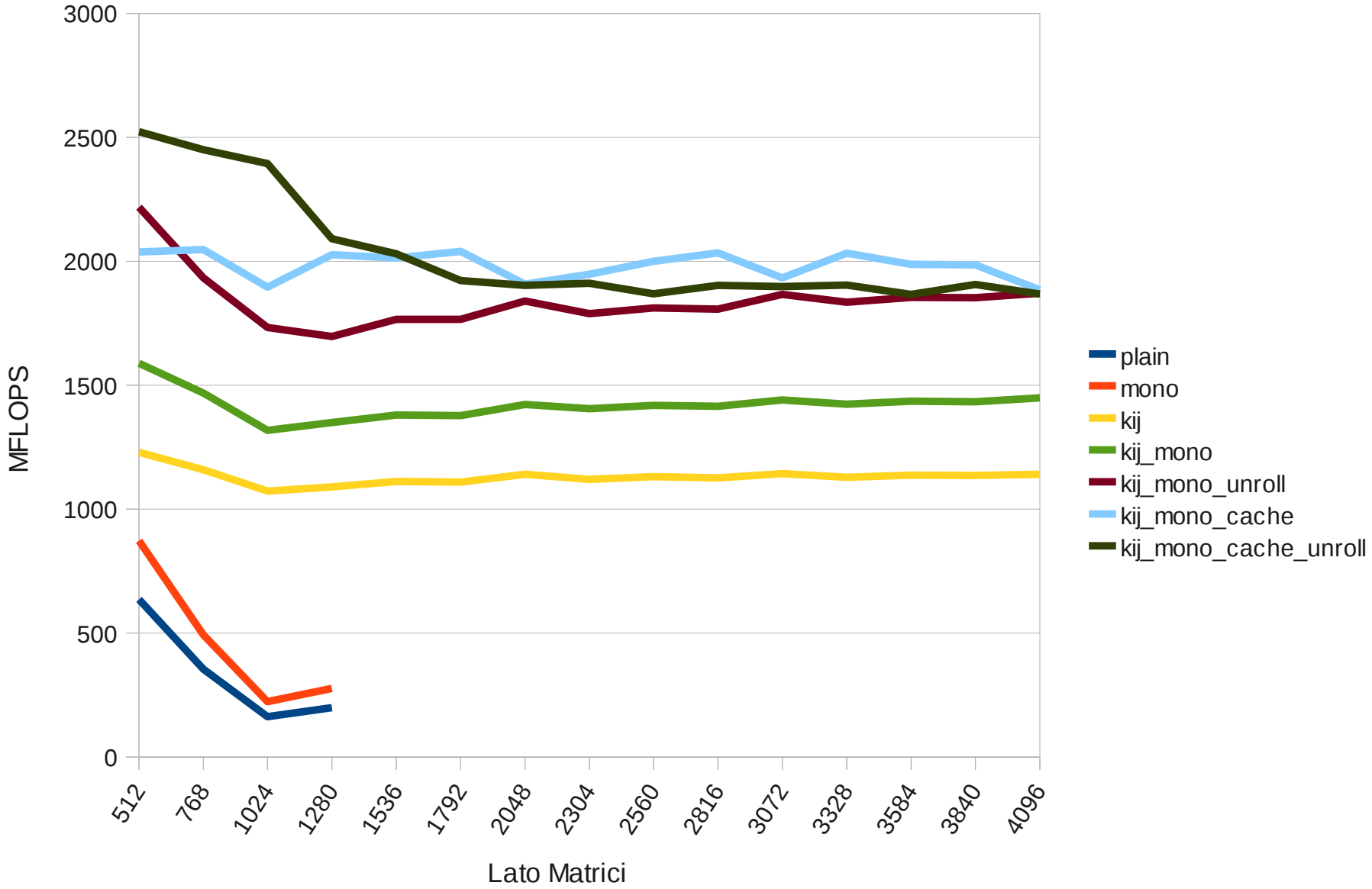
Esempio: Matrix Multiplication

Loop Unrolling + L1 friendly code



Esempio: Matrix Multiplication

Loop Unrolling + L1 friendly code



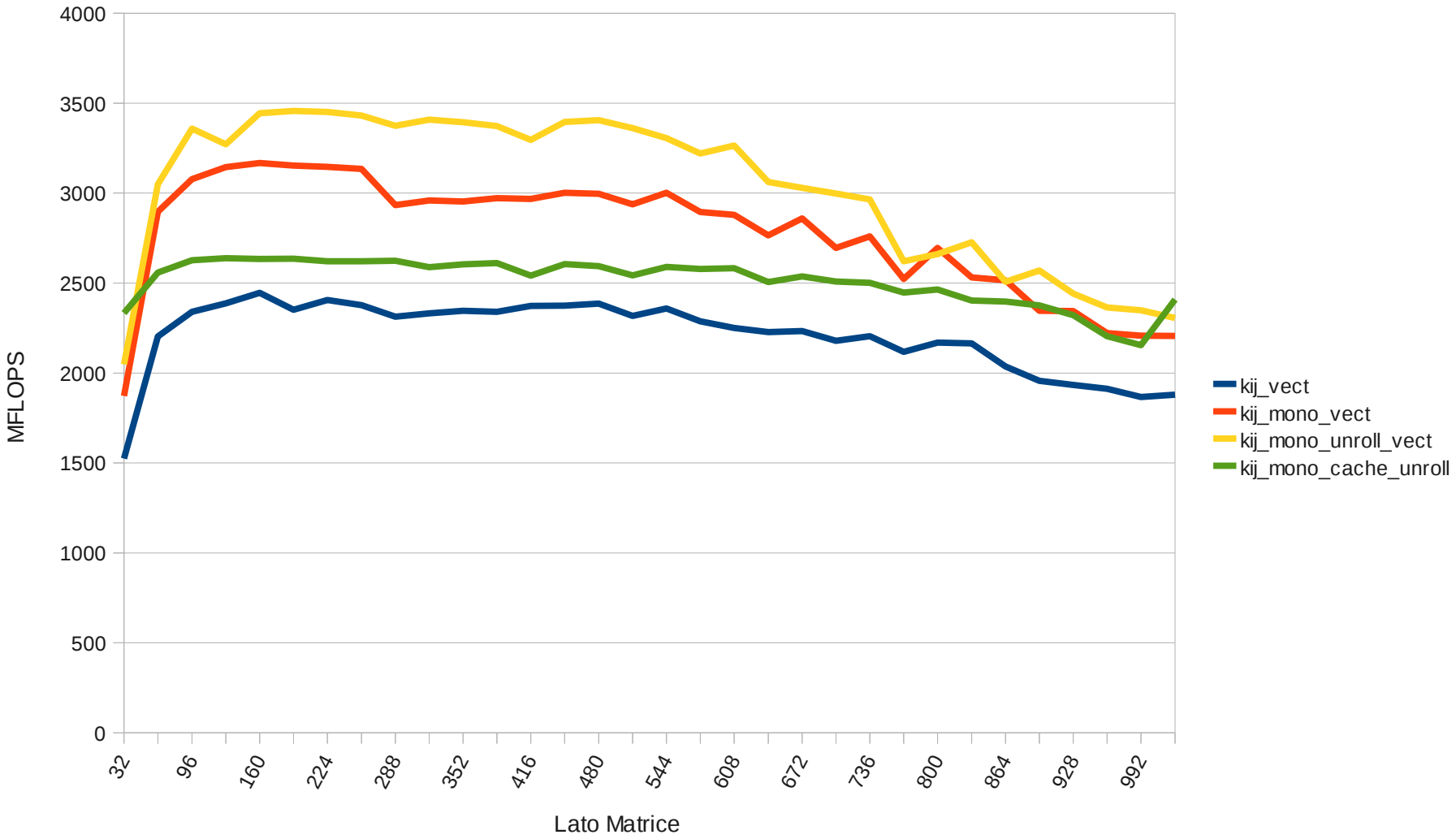
Esempio: Matrix Multiplication

Intrinsics e vettorizzazione del codice

- molti processori dispongono di unità in grado di eseguire istruzioni in modo SIMD (es. Intel: MMX, SSE, AVX, IBM: Altivel, AMD: 3DNow! + supporto per quelle Intel)
- un'unica istruzione è applicata a un vettore di più dati (in genere 128 bit, trattati come 2 double o long, 4 int o float, 8 short...)
- possibile aumento di 2x, 4x del throughput
- è possibile utilizzare questi instruction sets speciali con degli intrinsics del compilatore, ovvero macro che vengono convertite in istruzioni assembly di quell'instruction set.
- il compilatore riconosce pattern particolari di cicli e li vettorizza automaticamente (con flag `-O3` oppure `-O2 -ftree-vectorize`)
- usare `-ftree-vectorizer-verbose=2` per un report di gcc su cos'è riuscito a vettorizzare

Esempio: Matrix Multiplication

Vettorizzazione automatica del compilatore



Esempio: Matrix Multiplication

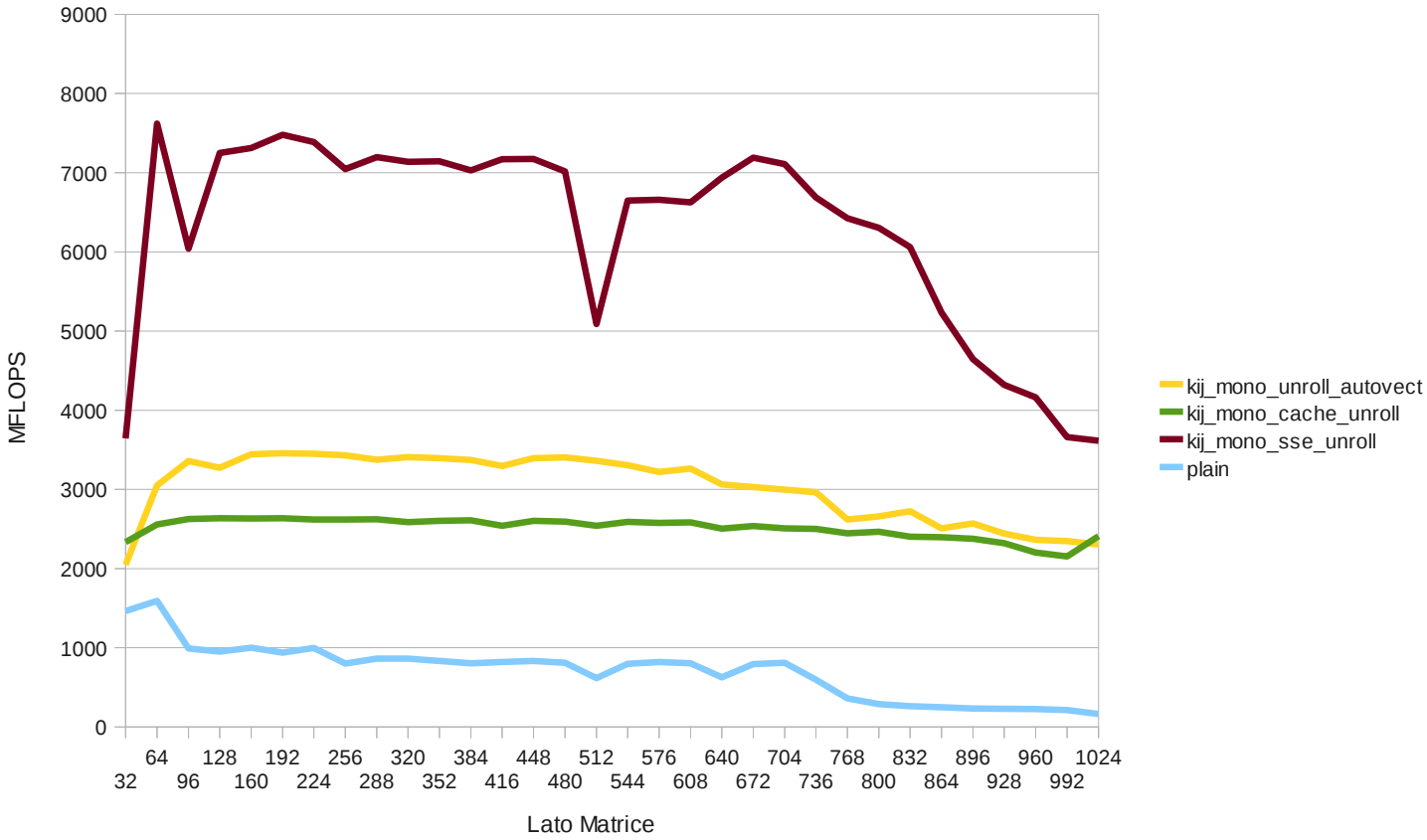
Intrinsics + unrolling

```
#include <xmmintrin.h>

void mm_kij_sse_unroll4(float *A, float *B, float *C, int l, int n) {
    __m128 ma, *mb, *mc = (__m128 *) C;
    int t1,t2;

    int n_4 = (n>>3)<<1;
    for (int k=0; k<n; k++) {
        mb = (__m128 *) B;
        for (int i=0; i<l; i++, mb+=n_4) {
            ma = _mm_load1_ps(A++);      //k*a+i
            for (int j=0; j<n_4; j+=4) {
                mc[j] = _mm_add_ps(mc[j], _mm_mul_ps(ma, mb[j]));
                mc[j+1] = _mm_add_ps(mc[j+1], _mm_mul_ps(ma, mb[j+1]));
                mc[j+2] = _mm_add_ps(mc[j+2], _mm_mul_ps(ma, mb[j+2]));
                mc[j+3] = _mm_add_ps(mc[j+3], _mm_mul_ps(ma, mb[j+3]));
            }
        }
        mc+=n_4;
    }
}
```

- istruzioni SSE, la stessa istruzione è applicata a 4 float.
- molto di basso livello (devo gestire load, store...)

Esempio: Matrix Multiplication

Vettorizzazione manuale



7.5 GFLOPS su 10, su un core con 2.5GHz $\Rightarrow$ 75% del picco di un core

Nota: qui è necessario dividere il calcolo in sottomatrici che fittino la L3

Esempio: Matrix Multiplication

Parallelizzare lavoro su più core/computer

- Thread Posix
  - gestione esplicita dei thread
  - sincronizzazione, accesso a risorse...
- OpenMP
  - framework per implementare il modello shared memory
  - direttive `#pragma` per parallelizzare cicli, sincronizzare ecc...

```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];
    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;
    return 0;
}
```

- MPI
  - message passing
  - potrebbe essere più lento dei precedenti in una macchina a memoria condivisa
- molte altre librerie/estensioni per compilatore

Esempio: Matrix Multiplication

Riepilogo

Tecnica	speedup	difficoltà
usare array monodimensionali	x2	1
unrolling dei loop	x2	1
compilare con -O2 o -O3	5x-10x	-10
esplicitare località, lavorare in cache	2x-5x	6
lookup table per ricomputazioni frequenti	2x-5x	3
scrivere metacodice che generi il codice con le costanti “hardcoded” nel codice (es. in FFT)	10x	7-8
usare dati binari e meccanismi di bufferizzazione manuali per accedere su disco	100x	3-5
scrivere cicli in modo che il compilatore possa vettorizzarli	2x-4x	4
scrivere codice vettorizzato a mano	x5	10
multi-threading	x*n core*	6-10