地球流体電脳 davis/ruby ワークショップ 2008年3月10日

x86なCPU上でのFFTの高速化について

石岡圭一

目次

FFTJプロジェクト

FFTJ = Fastest Fourier Transform from Japan
(FFTW = Fastest Fourier Transform in the West に対抗して)

状況認識:

ISPACK はこれまで主にベクトル計算機上での高速化を念頭において設計してきた.

しかし, 近年, ベクトル計算機は利用しづらい環境になってきている. (ベクトル計算機を作り続けているのは NEC のみであり, 地球シミュレータや阪大・東北大などの大計センターあたりでしか使えない).

HPC業界はどんどん超並列スカラー計算機に移行していっている (京大大計センターもその一例). 京速コンピュータも単純なベクトル機にはなりえない筈 (NEC・日立連合の方はベクトル機を作るという噂もあるが).

従って，今後はこのような計算機環境を念頭にライブラリの開発をしていかざるをえない (個人的には比較的容易に理論ピーク性能近くに到達が可能なベクトル機は好きだったのだが).

これまで:

ライブラリの並列化自体はそれほど困難なことではない.

共有メモリ環境では OpenMP ディレクティブをプログラムに埋め込めばよいし，分散メモリ環境では MPI でプログラムを書けばよい(どちらもデファクトスタンダードになっているので，普通どこでも使える).

ISPACK でも，主要な変換ルーチンにおいてこれらの並列化を施したものを既に作成して公開してある.

しかし，問題はそう簡単ではない.

問題点:

スカラー計算機とベクトル計算機ではCPUのアーキテクチャが根本的に異なるので，ベクトル計算機向けに開発したプログラムそのままではスカラー計算機の性能を十分には引き出せない.

というわけで，並列化を追求する以前に，単一CPUでの性能を十分に引き出すべくプログラムの抜本的な設計変更が必要である.

そのためにはスカラー計算機のアーキテクチャをまず十分に理解しなければならない.

最近やっていること:

身近なスカラー計算機として, Intel Pentium4, Core のアーキテクチャの理解とその上でのプログラムの高速化に取り組んでいる.

これは, Intel Pentium4, Core (またはその互換 CPU) の計算機環境はいたるところで利用可能であり,かつ非常に高速である(プログラムによっては, 1CPUで数GFlopsの演算性能を引き出せる)ので, このCPU上でまず高速化しておけば将来的にも有利である(次期京大大計は Quad Core Opteron(Barcelona)のクラスタになる).

また, Pentium4, Core のような広く流通しているCPUで高速化しておくことにより, 「誰にでも簡単に数値実験の追試のできる環境」を提供することが可能になりうると考える(追試のできない実験は実験ではない).

CPUのアーキテクチャを理解し，性能をフルに引き出そうとするならば，
アセンブリ言語を理解することが必須である．

最近ではコンパイラもそれなりに賢くなってきてはいるが，Pentium4，
CoreのSSE2(Streaming SIMD Extension 2)などを活用しようと
するならば，アセンブリ言語でのプログラミングは避けて通れない．

というわけで，ISPACKの高速化に向けた準備段階として，アセンブリ言
語の勉強を兼ねて，Pentium4，Core で高速に走る FFT をアセンブリ
言語で開発中である(FFTJプロジェクト)．

現在，とりあえず長さ 512 までの FFTを作成してみてある．
benchFFT(`http://www.fftw.org/speed/`) の指標では
Pentium4 Xeon(Prestonia)) や Core 2 Duo Xeon(Woodcrest)
でFFTW や Intel IPPS を凌駕している．

開発の方針:

いきなりアセンブリ言語で書くと訳分からなくなってしまうので，基本的アルゴリズムを考えた段階でまず FORTRAN77 で書いてみて，それからアセンブリ言語に書き換える．

こうすることによってFORTAN77プログラムがアセンブリ言語のプログラムの「注釈」のようになるし，バグフィックスが容易になる(基本的アルゴリズムに間違いがあるのか，それともアセンブリ化に誤りがあるのかが明解になる)．

また，もし将来的に PCの CPUのアーキテクチャが大きく変わり，現在書いているアセンブリコードが動かなくなっても(そんなことは当面無いと思われるが)，FORTRAN77コンパイラがあればスピードはともかくとして同じ機能が提供できる．

簡単なアセンブリコードの例を対応するFORTAN77コードと対応させて示す.

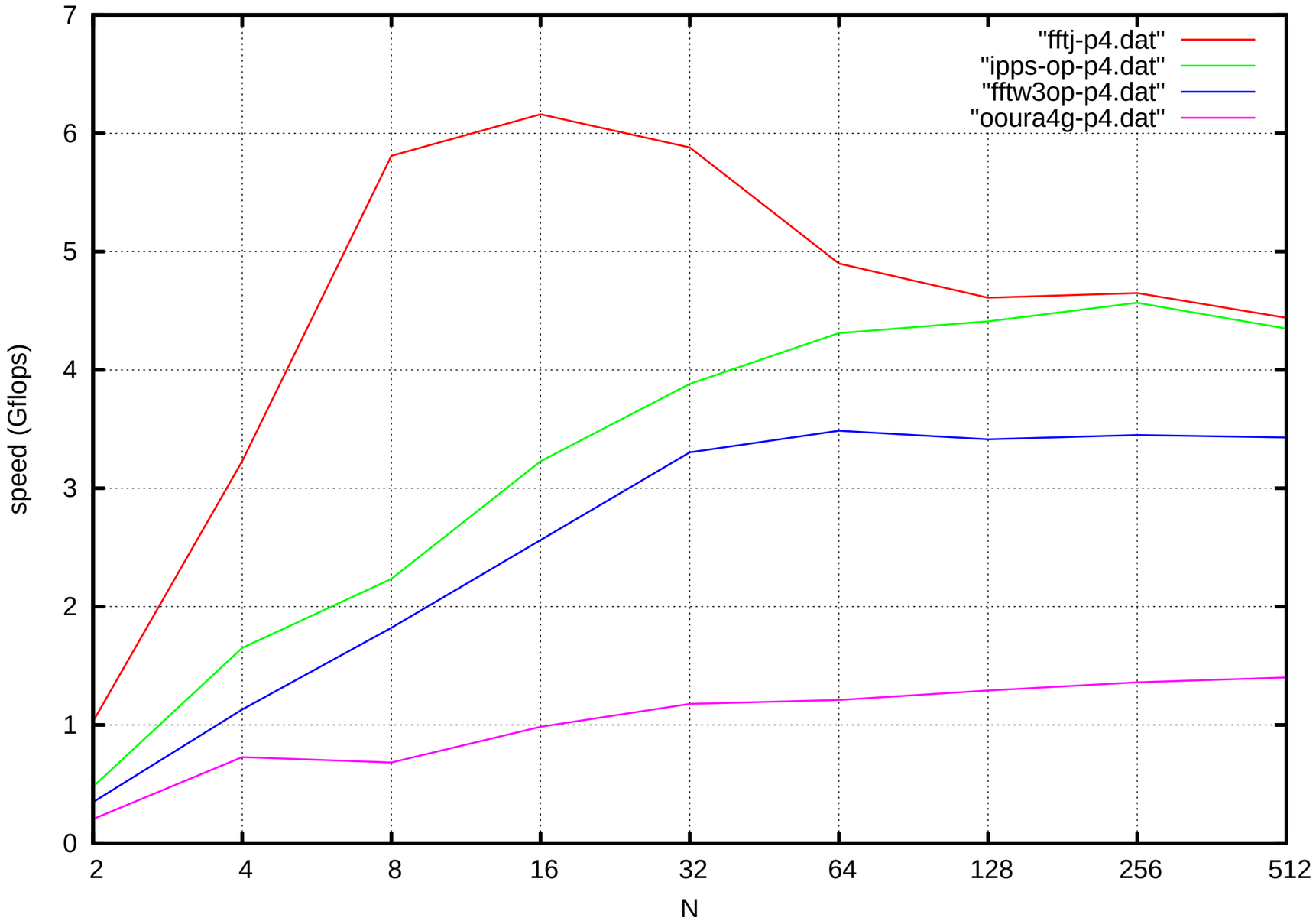
FORTRAN77:

```
      SUBROUTINE SUB(A,B)
      COMPLEX*16 A,B
      B=B+A
      END
```

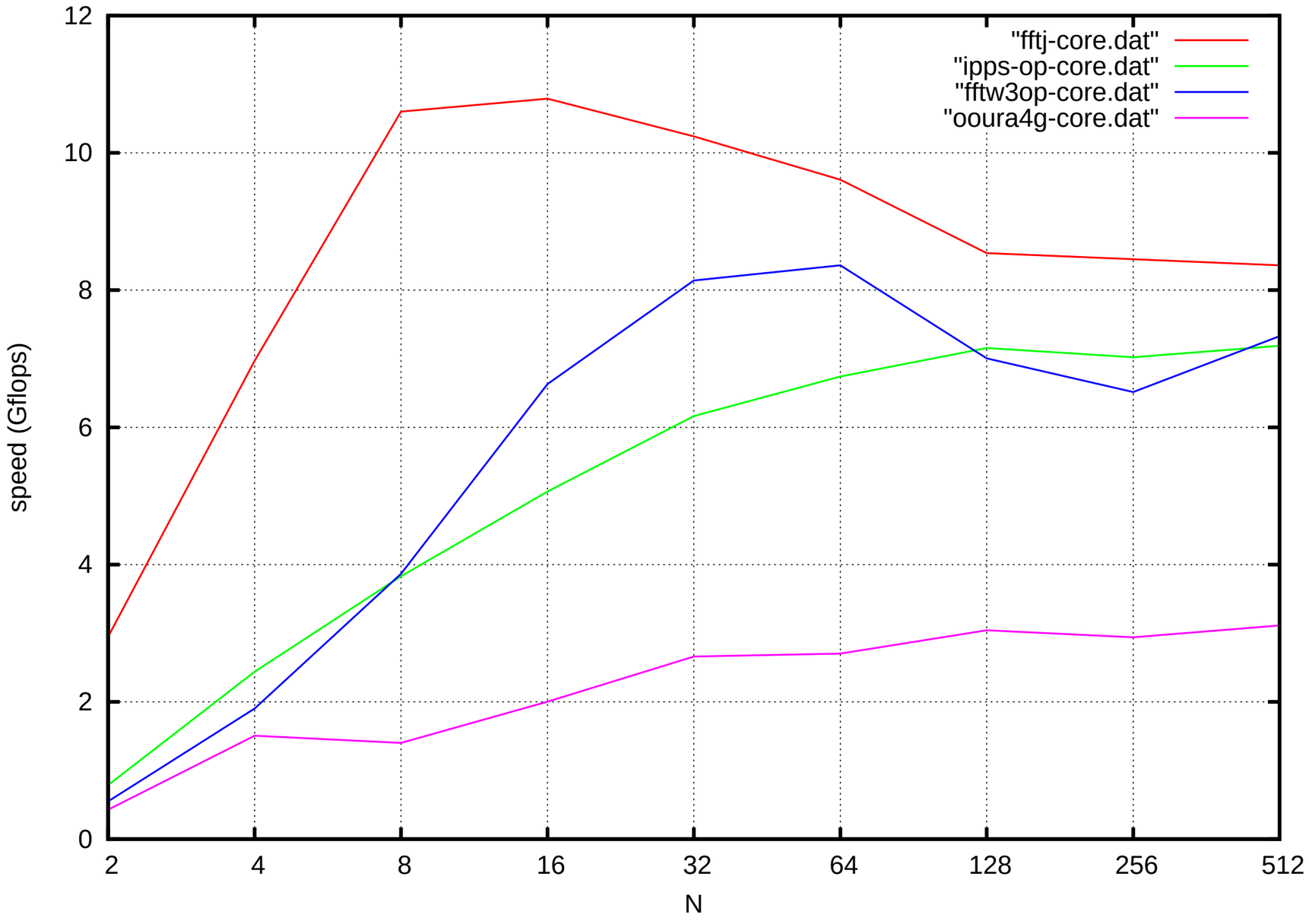
Pentium4, Core用アセンブリ:

```
.globl sub_
sub_:
 movl 4(%esp), %eax
 movl 8(%esp), %ecx
 movapd (%eax), %xmm0
 movapd (%ecx), %xmm1
 addpd %xmm0, %xmm1
 movapd %xmm1, (%ecx)
 ret
```

FFTの速度比較: Intel Pentium4 Xeon 3.06GHz (Prestonia)



FFTの速度比較: Intel Core2 Duo Xeon 3.0GHz (Woodcrest)



x86アセンブリのチュートリアル

自習するための資料

- 書籍
  - 蒲池・水越「はじめて読む Pentium マシン語入門編」(ASCII)
  - 林「高級言語プログラマのためのアセンブラ入門」(SoftBank Creative)
- インターネット上
  - IA32インテルアーキテクチャソフトウェアデベロッパーズマニュアル (ftp://download.intel.co.jp/jp/developer/jpdoc/)
  - Agner Fog氏のページ(x86アセンブリでの高速化について最大級の情報源) (http://www.agner.org/optimize/)

レジスタとは:

CPUにおける計算はレジスタと呼ばれる比較的少ないデータ格納領域に対する操作として行われる.

x86(Pentium4以降)におけるレジスタの分類:

汎用レジスタ: 32ビット(=4バイト)の大きさがあり, メモリアドレスの計算などに使われる.

　%eax, %ebx, %ecx, %edx, %esi, %edi, %ebp, %esp の計８個.　このうち，プログラマが自由に使えるのは %eax，%ecx, %edx の３個. その他については利用する際には値をメモリに退避しておいて後に復元しておかなければならない(細かい分類をするときは, “x” で終わるものだけを汎用レジスタと呼び, 残りはインデックスレジスタ, 特殊レジスタなどと呼ばれることもある).

XMMレジスタ: 128ビット(=16バイト)の大きさがあり，浮動小数点演算その他のSSE命令に使われる．%xmm0 − %xmm7 の計8個．このレジスタは自由に使える．

XMMレジスタは1つあたり単精度(4バイト)のデータを4つ，または倍精度(8バイト) のデータを2つ格納できる．SSE2命令では倍精度の2つのデータを同時に処理できることによって高速化につながる．

x86アセンブリについて:

アセンブリはマシン語を人間に分かりやすくしただけのものである． アセンブリをマシン語に変換するアセンブラの種類に依存して，アセンブリの記述方式は異なってくる． ここでは，Linux 上の標準的なアセンブラである GNU as が採用している AT&T Syntax を説明する(他にも MASM, NASM, Intel, ··· 等々，アセンブラそれぞれに固有の Syntax がある).

本日のチュートリアル用のサンプルプログラムは,
http://www.gfd-dennou.org/arch/ishioka/20080310/
以下に置いてあるので適宜参照されたい.

具体例その1:

FORTRAN77:

```
      SUBROUTINE SUB(A,B)
      COMPLEX*16 A,B
      B=B+A
      END
```

Pentium4, Core用アセンブリ:

```
.text
.globl sub_
sub_:
 movl 4(%esp), %eax
 movl 8(%esp), %ecx
movapd (%eax), %xmm0
movapd (%ecx), %xmm1
addpd %xmm0, %xmm1
movapd %xmm1, (%ecx)
ret
```

アセンブリの方の説明:

```
.text # 以下がコードであることを表す.
.globl sub_ # 他のプログラム単位から sub_ というラベルを
            # 参照可能にする.  なお,「#」以下はコメント.
sub_:  # メインプログラムで call sub した際にジャンプする先.
 movl 4(%esp), %eax
          # Aが格納されているメモリの先頭アドレスを %eax に.
 movl 8(%esp), %ecx
          # Bが格納されているメモリの先頭アドレスを %eax に.
#
# ここで, %esp はスタックポインタと呼ばれ, スタック領域という
# メモリ領域のうちで使用している部分の先頭アドレスを格納している.
# スタックに新たなデータが「積まれる」とその分 %esp は下位アドレス
# の方に下がっていく.  メインプログラムの CALL SUB(A,B) は,「B
# の先頭アドレスをスタックに積み, Aの先頭アドレスをスタックに積み,
# 戻り位置(CALL文の次の処理の位置)をスタックに積む」と実行される.
```

```
# なので，%esp を先頭とする4バイトには戻り位置が，%esp から
# 4バイト上位のアドレスを先頭とする4バイトにはAの先頭アドレスが，
# %esp から8バイト上位のアドレスを先頭とする4バイトにはBの
# 先頭アドレスが，それぞれ格納される．
#
# ここで，movl 8(%esp)，%ecx は，
#「%esp から8バイト上位のメモリアドレスから4バイト読み込んで
# %ecx レジスタに格納する」という動作になる．　従って，%ecx には
# Bの先頭アドレスが格納されることになる．　同様に %eax にはAの
# 先頭アドレスが格納される．
#
# ちなみに，movl 8(%esp)，%ecx において，
# movl が命令，8(%esp) をソースオペランド，
# %ecx をデスティネーションオペランド と言う．
# (NASM 等ではソースとデスティネーションの順が逆なので注意)
```

```
 movapd (%eax), %xmm0
# %eax に格納されたアドレスから16バイトを %xmm0 にロード.
# これで %xmm0 にAの値が格納される.
 movapd (%ecx), %xmm1
# %ecx に格納されたアドレスから16バイトを %xmm0 にロード.
# これで %xmm0 にBの値が格納される.
 addpd %xmm0, %xmm1
# %xmm0 に格納されている倍精度変数2つをそれぞれ %xmm1 に
# 格納されている倍精度変数2つに加える.
 movapd %xmm1, (%ecx)
# %xmm1 に格納されている倍精度変数2つ分を%ecx のアドレスにストア.
# これで %ecx のアドレスに A+B の値が格納される.
 ret # メインプログラムにリターン.
```

コンパイル:

メインプログラムと一緒にコンパイルする場合.

```
% g77 -O test.f sub-sse.s
```

オブジェクトファイルを作る場合.

```
% as sub-sse.s -o sub-sse.o
```

FORTRAN77からアセンブリへの変換.

```
% g77 -O -S sub.f
```

コンパイル(x86_64環境で32ビット用のアセンブリコードを使う場合):

メインプログラムと一緒にコンパイルする場合.

```
% g77 -m32 -O test.f sub-sse.s
```

ifort の場合.

```
% ifort -# test.f sub-sse.s
```

具体例その2(長さ2のFFT):

FORTRAN77:

```
      SUBROUTINE SUB2(Z)
      COMPLEX*16 Z(0:1),ZT
      ZT=Z0
      Z(0)=Z(0)+Z(1)
      Z(1)=ZT-Z(1)
      END
```

Pentium4, Core用アセンブリ:

```
.text
.globl sub2_
sub2_:
 movl 4(%esp), %eax
 movapd (%eax), %xmm0
 movapd 16(%eax), %xmm1
 movapd %xmm0, %xmm2
 subpd %xmm1, %xmm0
 addpd %xmm2, %xmm1
 movapd %xmm1, (%eax)
 movapd %xmm0, 16(%eax)
 ret
```

具体例その3(ループの例):

FORTRAN77:

```
  SUBROUTINE SUB3(N,Z,ZS)
  INTEGER I,N
  COMPLEX*16 Z(0:N-1),ZS
  ZS=0
  DO I=0,N-1
   ZS=ZS+Z(I)
  END DO
  END
```

Pentium4, Core用アセンブリ:

```
.text
.globl sub3_
sub3_:
    movl 12(%esp), %edx # ZS
    movl 8(%esp), %ecx # Z
    movl 4(%esp), %eax # N
    push %ebx
    push %edi
    movl $0,%ebx # I=0
    movl (%eax),%edi # N
    pxor %xmm0,%xmm0
.L1:  movaps (%ecx), %xmm1
    addpd %xmm1,%xmm0
    addl $16,%ecx
    addl $1,%ebx
    cmpl %ebx,%edi
    jnz .L1
    movaps %xmm0,(%edx)
    pop %edi
    pop %ebx
    ret
```