Protein Folding with the Pentium III

Michael Corbin Patrick Trinkle Zach Bennett

December 8, 2008

1 Introduction

Protein folding is the process of polypeptides folding into three-dimensional structures[6]. The Pande Lab at Stanford has developed simulation software called `Folding@home` (`FAH`)[2], which can be used to simulate this folding process. Understanding and simulating the folding process can lead to a greater understanding of the consequences of proteins folding incorrectly, which can lead to known diseases, like Alzheimer's, Mad Caw, Huntington's and more[2]. This paper proposes a design for an optimized Pentium III processor, designed specifically to optimally run the `FAH` protein folding simulation software, allowing scientist and researchers to perform more simulations in less time in order to further study the folding process.

Our approach included researching the code structure of `FAH` to discover where most of the processing occurred. We then modified Intel's Pentium III processor to improve the slow parts of the code, and provide quantifiable evidence of its improvement over a standard Pentium III chip. The use of the Pentium III as a base chip was based on some of the features of the Pentium III. These features are discussed in sections 2 and 3. The design improvements are discussed in section 4, while the results are discussed in section 5.

2 Characteristics of the Application

`Folding@home` uses several different cores depending on the type and configuration of the system running the simulation. There are different software cores for single core CPUs (Central Processing Unit), multi-cored CPUs, GPUs (Graphics Processing Unit), Sony Playstation 3, and many more. Most of the cores, and all of the fastest cores, are based off the Gromacs software[4]. Gromacs is a highly optimized software suite that can simulate molecular dynamics. It is broken up into various kernel pieces, each optimized to achieve fast calculations on different chip architectures. To narrow the focus of this paper we chose the nb (non-bonded) 302 kernel code for the IA-32 Architecture.

Gromacs can achieve speeds twenty to thirty times faster than the older cores used by `FAH`[3]. Gromacs achieves this speed up over other molecular dynamic simulators in a variety of different ways. It uses specialized instructions like `SSE` (Streaming SIMD[1] Extensions) and `3DNow!`[2] on Pentium and AMD chips. It moves all possible calculations from the algorithms out of what is labeled as the *inner loops* in order to speed up the code that is run most often. Gromacs also generats assembly code for entire inner loops, quad unrolled[3] and without conditional statements. The program then picks which inner loops are needed for each simulations depending on what is being simulated and which processor is being used.

Gromacs spends approximately 90% of the computation time for protein folding doing

1. SIMD Instructions (Single Instruction Multiple Data) are instructions that perform the same operation on multiple data elements at once[7]

2. Multimedia extension developed for AMD processors, adding SIMD support to the x86 instruction set[1].

3. Quad unrolled refers to a assembly loop being listed out 4 times to avoid the overhead of branching at the end of each iteration[12].

non-bonded force calculations inside these inner loops. These calculations are slow because of the number being performed and the complexity of the equations involved. Most atoms are only bonded to a handful of other atoms so non-bonded force calculations are required to figure out the majority of the forces involved in a simulation. Each non-bonded force calculation can involve the Coulomb force or Lennard-Jones calculations. These calculations involve square root and reciprocal operations that can be very slow if specific hardware is not provided. Gromacs is specifically optimized to speed up the non-force bounded calculations in several ways. It creates a neighbor list for each atom in order to reduce the number of operations needed. It also has software look-ups for the square root and reciprocal operations for processors without fast hardware support[11].

The Pentium III was chosen as the base processor because of its existing support for the operations involved in the simulations. The Pentium III provides fast square root and reciprocal operations, taking advantage of the speed up over Gromacs software solution. The Pentium III also supports SIMD operations through its `SSE` support. A processor with `SSE` support was chosen over one implementing `3DNow!` because of `SSE`'s ability to perform operations on four floats at once over the two float operations provided by `3DNow!`. The Pentium III was the first Pentium processor to support `SSE`[10], while retaining a level of simplicity (compare to its successor the Pentium 4) allowing application specific optimizations to be made.

3 Pentium III Background

The Pentium III is very similar to its predecessor, the Deschutes Pentium II processor, with the addition of `SSE` support[5]. Its first release (*Katmai*) was built on a 25μ process[5] and had an originally announced speed of 450 - 500 MHz. It was the first Pentium processor to implement `SSE`, which added 70 instructions and a new memory-streaming architecture[10]. Having this technology allows our application to take full advantage of SIMD instructions, which greatly increases performance, specifically for `FAH`, as approximately 86% of all its instructions in the inner loop (which is run 90% of the time) are SIMD instructions.

`SSE` is implemented on the Pentium III by adding eight 128 bit registers (`xmm[0-7]`). These registers are capable of holding four IEEE single-precision floating point data elements, which can then be performed on in parallel. While these registers contain 128 bits of data, the Pentium III did not widen its data path to 128 bits, but left it at 64 bits. It then implemented operations on the 128 bit registers by double-cycling the existing 64-bit data paths. While this limits the performance increase of the Pentium III over some of its SIMD enabled peers like the K6 III's 3DNow! implementation, an increase can be gained through its multiple instruction issue[10]. Since a SIMD instruction performs an operation on all of its elements in parallel, SSE allows four multiply or four addition operations to be started per cycle. However, since the Pentium III double-cycles the 64-bit data path, it must perform SIMD operations using two 64 bit μops[10] reducing the number of operations performed from a single instruction to only two per cycle. To compensate, it places the SIMD multiplier and SIMD adder units on different ports so they can be issued at the same time (see figure 1). A

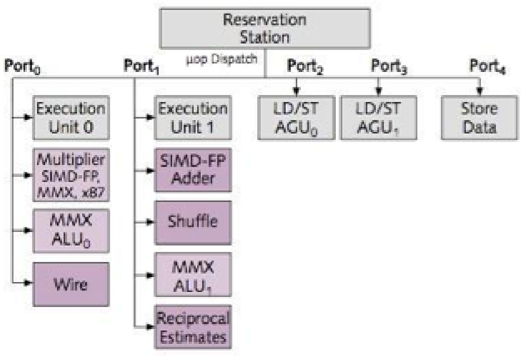

Figure 1. Original Pentium III Port/Functional Unit Arrangement [10].

single resource assignment can be made to each port per cycle. This means that any unit attached to Port x can be allocated at the same time as any unit attached to Port y as long as $x \neq y$. Only one unit per port can be assigned per cycle. This allows for alternating, data independent addition and multiplication operations to increase throughput back to four operation per cycle, even with the double-cycling of data paths. However, if the operations are dependent, then a latency is incurred to allow both μops to run. The Pentium III also allows for two simultaneous move operations to help throughput when instructions can not be scheduled to perform simultaneous, data independent multiply and addition operations[10].

The throughput of loads and stores is increased to allow for the throughput increase of FP operations to be utilized. The Pentium III includes prefetching of instructions so that load resources would retire earlier, and subsequent loads could use the memory unit. It also includes write back technology that could bypass cache on a miss, avoiding adding a line to the cache that would never be used. By overloading the use of load buffers, the Pentium III also increases the number of Write-Combining buffers to four, to allow for a greater level of optimization using write combining. Write combining is writing back multiple store data that is sequential in order to take advantage of bus speed. Adding extra buffers allows for partially filled Write-Combining buffers to wait for more data to be found while other store operations are issued[10].

4 Description

The original approach to the optimization of the Pentium III processor was adding instructions that would allow for faster processing of the Coulomb forces and Lennard-Jones equations. It soon became clear that this would not work because of the complexity of the equations and the fact that not all simulations use both equations. Instead, we decided to speed up the current `SSE` calculations. The Gromacs nb302 kernel code is not optimized to alternate independent floating point multiplications and additions each cycle so it can not run at full speed on the Pentium III processor. What we did find was that the assembly code for Gromacs had multiple instances of groups of three data independent SSE instructions. These instructions would all use the same operation, but none of the same registers. We refer to these sets of instructions as *triple-set* instructions.

4.1 Triple-Set Instructions

Table 1 provide the number of triple-set instructions that appear in the assembly code. On an unoptimized Pentium III these

Instruction	Triple Count
addss	1
xorps	2
movss	2
addps	3
mulps	6
subps	6
movaps	16

Table 1. Gives the Number of times like, data independent instructions are seen in sets of three in the .nb302_single_loop for IA32. All instructions are SIMD instructions except for `addss` and `movss` which are scalar, but act on 128 bit registers.

instructions can only be executed every other cycle because they all use the same function unit and the other instructions around them are data dependent. This is very limiting.

We decided to take advantage of this pattern by adding functional units to different ports on the Pentium III. This allows us to issue more instructions per cycle and avoid stalling because of lack of function units. In order to take full advantage of the structure of the code, we made the following additions: an Adder and Reciprocal unit to port 0, a Multiplier to port 1, a Store, Multiplier, ALU (arithmetic logic unit), and Adder to port 2, a Store to port 3, and a Load to port 4. Figure 2 illustrates all of the additions and there placement. While it may look like units

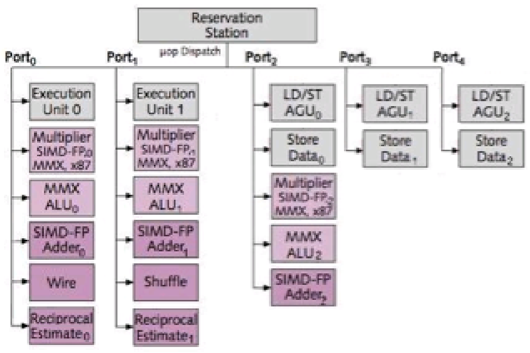

Figure 2. New arrangement of ports for the optimized Pentium III. Ports 2-4 now all contain load and store functional units and ports 0-2 all contain SIMD multiply and adder units. An additional Reciprocal unit was added as well to port 1.

were arbitrarily added to any port, there is a reason for each addition. The current configuration of units optimize Gromacs tendency these triple-set instructions. This allows multiple SIMD instruction to be issued and executed on the same cycle. We included three Multipliers, Adders, ALUs, Loads, and Stores because of the tendency of Gromacs to do three mulps, subps, addps, movss, and movaps at a time. In the single loop code alone there are 333 instructions. 108 of these instructions are data-independent triple-sets. This means that for the code that is run 90% of the time, 32.43% of it is a set of three like instructions, all needing access to the same function unit. Improving this section of code follows the important *Focus on the Common Case* principle of computer design, which states to concentrate and optimize the common case, or the code run the most[12]. The inner loop is overwhelmingly the common case. We also included two Reciprocal units because there are several instances of two sequential rsqrt instructions, again avoiding stalls waiting for functional units to become free.

Adding SIMD function units does not solve the problem that instructions can only be issued every other cycle (to the same port). This limitation is because of the 64 bit data paths and the fact that the Pentium III issues every SIMD instruction as two μops. To fix this we expand the data paths to 128 bits. This provides that when the two instructions are fetched to the reservation station, they can both be sent out to a port for execution, and they will complete in a single cycle, freeing up that unit for a second instructions instead of requiring an extra cycle for the second 64 bit μop to execute. This has the added benefit of reducing stalls when an instruction must wait for dependant data, for those instruction combinations that do contain dependencies.

4.2 Branch Prediction

Gromacs developers have spent a great deal of time developing branch free code for the the inner loop. Branches slow down processing by not allowing the CPU to know definitively which instructions are to be issued next. Chip designers spend large amounts of time and money on prediction techniques that might increase the accuracy of the CPU's guess as to which instruction to issue next. Since the inner loop is branch free (minus the single branch at the end of the loop sending it back to the top of the loop), our processor does not need much of this technology. The processor still needs to handle the other 10% of the processing, so we do not want to completely purge branch prediction. But it can be reduced. The optimized chip could contain a tournament style prediction method[12]. For the inner loop conditional branch, a single bit predictor will suffice. We can guarantee that it will only be wrong at most twice. It may be wrong initially, and it will be wrong on the exit. Making this a 2-bit predictor does not increase accuracy. Given that 90% of processing is done with approximately 300 instructions, we will get great performance from our branch prediction for the inner loop. The single bit predictor could be replaced by a single entry branch target buffer if address evaluation cannot be done in the first cycle. This would allow the CPU to know where the next branched instruction should be (and will be accurate an overwhelming amount time). The predictor to be used for the remaining 10% of the code can be a standard predictor. The Pentium III uses a Branch Target Buffer (BTB) for branch prediction[10], which could be reduced in size. The enhancement gained by simplifying branch prediction is not in performance, but in size. By reducing the BTB size, we allow more space for the additional units and increased data paths that we need to implement the previous enhancements. We justify these improvements in the next section.

5 Justification

To determine the possible improvement or speedup of the modifications to the processor we first needed to calculate the original CPI (cycles per instruction). Due to complications in the processor, we normalized many aspects to simplify the problem–while providing consistent results. Because the SIMD instructions are broken into 2 issue μops, we made the entire system dual issue; and maintained this relationship. On the original Pentium III, unless the sequence was a SSE multiply followed by a SSE add (or the other way around); the SSE instructions were issued on alternating cycles. Data dependencies were factored into the problem, as accounting for these is fairly straightforward. The Pentium III uses reservation stations; so a Tomasulu approach was required. All instructions were then set to require 1 cycle to execute; while the SSE instructions executed in 1 cycle for each 64-bit piece. The 333 instructions of the inner loop were manually calculated with the aforementioned assumptions to require 614 cycles to complete. This provides for an overall CPI of 1.8439. This is a result of most instructions stalling for any like arithmetic SIMD instruction which monopolized the function unit for two cycles.

Our improvements to the data path, and additional functional units allow for two SSE instructions per cycle, provided no data dependencies. Again, running through the code, it was calculated that the 333 instructions executed in 169 cycles, providing a CPI of 0.5075. This provided an overall speed up of

$$\text{Speedup}_{\text{overall}} = \frac{\text{CPI}_{\text{old}}}{\text{CPI}_{\text{new}}} = \frac{1.8439}{0.5075} = 3.633$$

This is a significant speed up over the native Pentium III. The increased datapath and extra functional units provide more opportunity for execution.

5.1 Space on Chip

In order to add functional units to a processor, and increase its data paths, there must be room on the chip to do so. This extra space can be found in several places. First, the Pentium III, being generations old, has not taken advantage of the current levels of miniaturization that can be achieved by today's technology. The Pentium III was originally manufactured at 0.18-micron on a 128 mm2 chip, while modern processors are manufactured at 65nm or less on dies over twice the size of the Pentium III[10]. Using the current manufacturing process we could easily fit the additional function units and data paths on the Pentium III. If that's not enough, room can be found by simplifying the branch prediction capabilities, and reducing its footprint on the chip. Needing only a very simple predictor for the inner loop, trade offs can be calculated for the size of a BTB, and the reduced performance of the other 10% of the cod versus the increased performance that is achieved by adding the functional units. Since our focus was primarily the 90% portion of the code, no exact analysis was performed on how the other 10% would be affected by decreasing the size of the BTB. However, a simple analysis could be done by analyzing the performance degradation in the worst case scenario.

Worst case is that the BTB must be done away with completely, minus the single entry needed for the inner loop. This would mean that 10% of the time, the code would suffer from poor branch performance. The worst case scenario for that 10% of code is that every other instruction is a branch (at least one cycle is needed to perform operations on the data being compared against to branch, as well as to do anything useful besides jumping around code). Assuming a two cycle stall (or penalty) for a branch, which would give a cycle for issuing, and a cycle for address calculation and reporting, this means that 50% of the code would have a CPI of 3, while the rest would have a CPI of 1, giving a total CPI of 2 for the 10% portion. Since every other instruction is a branch, we assume we cannot take advantage of dual issue, which is why the CPI is 1 for the remaining 50% even though the Pentium III has dual issue. Assuming the new CPI as calculated above, and that with branch prediction, the rest of the 10% portion somehow miraculously achieves a CPI of .5 with dual issuing (we are looking at worst case), we get the following speed up.

$$\begin{aligned}
\text{CPI}_{\text{new10}} &= \frac{(I \times .5)(1+3)}{I} = 2 \\
\text{CPI}_{\text{old10}} &= .5 \\
\text{CPI}_{\text{new90}} &= .5075 \\
\text{CPI}_{\text{old90}} &= 1.8439 \\
\text{Speedup} &= \frac{1.8439 \times .9 + .5 \times .1}{.5075 \times .9 + 2 \times .1} = 2.60
\end{aligned}$$

This means that even if the entire BTB had to be removed to make room for the extra units, we would still achieve a speed up of

2.6. This justifies decreasing the BTB to whatever size is necessary to fit the extra functional units on the chip, though we are confident that the simple miniaturizing of parts will be sufficient.

5.2 Instruction Cache

Another resource for additional room on the chip can be the instruction cache size. Present day chips have L1 instruction caches much larger than the Pentium III's L1 instruction cache. The Pentium III has a 16KB L1 instruction cache with 32B cache lines[10]. This is a small amount compared to today's standards, and the size of a 16KB cache is physically smaller now than it was when the Pentium III was developed. If we can show that we will never need an instruction cache larger than 16KB, then as the chip scales in size, instead of the cache getting bigger to fit the space, it will remain small to allow the space to be used for functional units (or other performance enhancing features). Since 90% of the time will be spent on the inner loop of our application, it is important that the instruction cache is large enough to hold the entire quad unrolled loop of the inner loop. Assuring that it does will guarantee that there will be no instruction cache misses save the initial compulsory misses. Seeing that these misses only occur once for the entire duration of the inner loop, adding extra hardware logic for prefetching the entire unrolled to avoid compulsory misses would not provide adequate speed up for its cost. Seeing that a single loop takes 614 cycles, and the loop takes up 90% of our processing, then speeding up a single loop will not provide a noticeable speed up. However, it is important to minimize the number of cache misses per iteration. If there is not enough cache for the entire unrolled loop, then every iteration we will experience an instruction cache miss. This would have devastating effects on performance. On the other hand, if there exists an extra large instruction cache, then most of the cache space will not be used and thus be wasted on the hardware. So while no changes are made to the Pentium III in terms of instruction cache size, choosing a chip with the right amount of cache space played into the decision of the base chip.

In order to take full advantage of the space on the die for the instruction cache, we need to know exactly how many instructions will be stored, and how large each instruction is. We know that a single loop is 333 instructions with approximately 24 distinct instructions and seven distinct instruction sizes as show in table 2[9][8]. The table only accounts for instructions with a limited number of addressing modes. Instructions like ADD with dozens of different addressing modes and instruction combinations were not considered as it is likely that a rough approximation will be acceptable. The instructions used were mostly the SIMD instructions which tend to have larger instruction sizes. As can be seen from table 2, a single loop contains 2678 bytes of instructions. In order to hold an entire unrolled loop, we will need four times a single loop requiring 10,712 bytes or approximately 10KB. This data points to 16KB being just the right amount of instruction cache. The extra 6KB will account for any extra instructions we did not consider, as well as provide some extra space for the other 10% of the program. Given 10KB of needed instruction cache leads to a 16KB cache size since cache size grows in powers of two, and the next size down, 8KB, is too small to fit all of the instructions and would have significant performance degradation. At the same

Instruction Size	**Number of Instructions**	**Total Size**
19	90	1710
8	14	112
7	29	203
6	2	12
5	18	90
4	137	548
3	1	3
Total	**291**	**2678**

Table 2. Instruction Sizes, and the number of instructions that have that size for a single iteration of the inner loop. Their products yields the total size needed for each of the instruction types and gives a total number of bytes needed for a single loop. Instruction sizes and total size are both given in bytes.

time, a 32KB L1 instruction cache would be too large, and would leave two thirds of the cache idle for 90% of execution. Therefore, keeping with the 16KB of the original Pentium III allows us to optimize the space used on the chip to make full use of the application running, whether this is to allow room for functional unit, wider data paths, more registers (for future iterations of the chip), or bigger L1 data caches, or L2 caches. This is again a performance gain in size, not in speed.

6 Conclusion

Our goal for this project was to speedup the `Folding@Home` protein folding application. While researching this project we found that currently `FAH` uses Gromacs to perform most of its protein simulations and that Gromacs spends around 90% of the time doing non-bonded force calculations on the atoms involved. We determined we could speed up the non-bonded force calculations by improving the `SSE` implementation of the Pentium III. It was calculated that 32.43% of the code is in triple-set form–and therefore if we found a way to optimize this it would have a significant impact. With this in mind we added functional units and expanded the data paths and we were able to improve the CPI by a factor of 3.63. We also looked at improving the cache of the Pentium III in order to prevent cache misses with the larger quad unrolled and single loop but calculated that the Pentium III's cache was not only sufficient to accomplish this, but also optimal in its allocation to the instruction cache. While this doesn't give us any quantifiable performance increase, it allows us to argue further that more units and larger data paths are feasible by decreasing the size of the features of the chip, and not needing to scale up the instruction cache.

We originally hoped to get at least a factor of 2 speed up for the inner loop of Gromacs application, by simply adding functional units. Since we were able to accomplish the goal we felt that our improvements turned out very well. The primary thing we would change about our approach if we had to repeat this project is to select an architecture that the code was not so closely tied to. Since the code was already extensively optimized for the x86 architecture we had a difficult time finding software improvements, and some of the improvements that were found were tied too closely to a specific set of assembly code and would not always be present in other similar code sets. As a result, we turned to improving and adding to the hardware, choosing a specific chip as a starting point, allowing us to quantify an expected speed up.

Bibliography

[1] 3dnow! `http://www.wikipedia.org/wiki/3dnow`.

[2] Folding@home. `http://folding.stanford.edu/`.

[3] Folding@home. folding@home gromacs faq. `http://folding.stanford.edu/English/FAQ-gromacs`.

[4] Gromacs: Fast, free and flexible md. `http://www.gromacs.org/`.

[5] Pentium iii. `http://www.wikipedia.org/wiki/Pentium_III`.

[6] Protien folding. `http://www.wikipedia.org/wiki/Protien_Folding`.

[7] Simd. `http://www.wikipedia.org/wiki/SIMD`.

[8] Intel architecture software developer's manual instruction set reference, 1999.

[9] Intel 64 and ia-32 architecture software developer's manual, instruction set reference, a-m, 2007.

[10] Keith Diefendorff. Pentium iii = pentium ii + sse internet sse architecture boosts multimedia performance. *Microprocessor Report the Insiders' Guide to Microprocessor Hardware*, Vol 13(No 3), 1999.

[11] David Van Der Spoel Erik Lindahl, Berk Hess. Gromacs 3.0: a package for molecular simulation and trajectory analysis. 2001.

[12] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

7 Glossary

Terms and acronyms used throughout the paper

`FAH`	Folding@Home
`SIMD`	Single Instruction Multiple Data
`SSE`	Streaming SIMD Extendions
`ALU`	Arithmetic Logic Unit
`FU`	Functional Unit
`FPU`	Floating Point Unit
`FP`	Floating Point
`nb`	Non-Bounded. Two atoms are non-bounded if they do not share a molecular bond.

8 Appendix A - Instruction Descriptions

Description of certain instructions used in the Gromacs assembly code [8][9].

Instruction	**Description**
MOVAPS	Move Aligned Packed Single-Precision Floating-Point Values Moves a double quadword (128 bits) to or from either an XMM register or memory
ADDPS	Add Packed Single-Precision Floating-Point Values Executes a SIMD add on the source and destination and stores the result in the destination
MULPS	Multiply Packed Single-Precision Floating-Point Values Executes a SIMD multiply on the source and destination and stores the result in the destination
SUBPS	Subtrace Packed Single-Precision Floating-Point Values Executes a SIMD subtraction from the destination by the source and stores the result in the destination
XORPS	Bitwise Logical XOR for Single-Precision Floating-Point Values Executes a bitwise logical XOR on the source and destination and stores the result in the destination
MOVSS	Move Scalar Single-Point Floating-Point Values Moves a scalar single-precision floating-point value from either a 32-bit address or an XMM register, but not between two memory locations
ADDSS	Add Scalar Single Precision Floating-Point Values Adds the low single-precision floating-point value from the source and destination. The destinations must be an XMM register, while the source can be either that or a 32-bit memory location

9 Appendix B - Assembly Source Code

The source we used to run our optimization calculations. The Gromacs simulation code uses specific kernels for specific non-bonded calculations; so we picked one of the kernels and examined it. The below code is strictly the single loop iteration; there are two primary sections to the kernel; the quad-unrolled loop and the single loop. The single loop optimizations can be extrapolated and applied to the quad unrolled loop code.

x86 32-bit nb302 single loop original code

```
.nb302_single_loop:
mov edx, [esp + nb302_innerjjnr] ;# pointer to jjnr[k]
mov eax, [edx]
add dword ptr [esp + nb302_innerjjnr], 4
mov esi, [ebp + nb302_pos]
lea eax, [eax + eax*2]
```

```
;# fetch j coordinates
xorps xmm3, xmm3
xorps xmm4, xmm4
xorps xmm5, xmm5
movss xmm3, [esi + eax*4]
movss xmm4, [esi + eax*4 + 4]
movss xmm5, [esi +eax*4 + 8]
movlps xmm6, [esi + eax*4 + 12]
movss xmm7, [esi + eax*4 + 20]
movhps xmm6, [esi + eax*4 + 24]
movss xmm2, [esi + eax*4 + 32]

shufps xmm6, xmm6, 216
unpcklps xmm7, xmm2
movaps xmm0, [esp + nb302_ixO]
movaps xmm1, [esp + nb302_iyO]
movaps xmm2, [esp + nb302_izO]
movlhps xmm3, xmm6
shufps xmm4, xmm6, 228
shufps xmm5, xmm7, 68

;# store all j coordinates in jO
movaps [esp + nb302_jxO], xmm3
movaps [esp + nb302_jyO], xmm4
movaps [esp + nb302_jzO], xmm5
subps xmm0, xmm3
subps xmm1, xmm4
subps xmm2, xmm5
movaps [esp + nb302_dxOO], xmm0
movaps [esp + nb302_dyOO], xmm1
movaps [esp + nb302_dzOO], xmm2
mulps xmm0, xmm0
mulps xmm1, xmm1
mulps xmm2, xmm2
addps xmm0, xmm1
addps xmm0, xmm2 ;# have rsq in xmm0

;# do invsqrt
rsqrtps xmm1, xmm0
movaps xmm2, xmm1
mulps xmm1, xmm1
movaps xmm3, [esp + nb302_three]
mulps xmm1, xmm0
subps xmm3, xmm1
mulps xmm3, xmm2
mulps xmm3, [esp + nb302_half]

movaps xmm1, xmm3
mulps xmm1, xmm0 ;# xmm1=r
movaps xmm0, xmm3 ;# xmm0=rinv
mulps xmm1, [esp + nb302_tsc]

movhlps xmm2, xmm1
cvttps2pi mm6, xmm1
cvttps2pi mm7, xmm2
cvtpi2ps xmm3, mm6
cvtpi2ps xmm2, mm7
movlhps xmm3, xmm2
subps xmm1, xmm3 ;# xmm1=eps
movaps xmm2, xmm1
mulps xmm2, xmm2 ;# xmm2=eps2
pslld mm6, 2
pslld mm7, 2

movd ebx, mm6
movd ecx, mm7
psrlq mm7, 32
movd edx, mm7
```

```
  mov esi, [ebp + nb302_VFtab]

  movlps xmm5, [esi + ebx*4]
  movlps xmm7, [esi + ecx*4]
  movhps xmm7, [esi + edx*4] ;# got half
coulomb table
  movaps xmm4, xmm5
  shufps xmm4, xmm7, 136
  shufps xmm5, xmm7, 221

  movlps xmm7, [esi + ebx*4 + 8]
  movlps xmm3, [esi + ecx*4 + 8]
  movhps xmm3, [esi + edx*4 + 8] ;# other
half of coulomb table
  movaps xmm6, xmm7
  shufps xmm6, xmm3, 136
  shufps xmm7, xmm3, 221
  ;# coulomb table ready, in xmm4-xmm7
  mulps xmm6, xmm1 ;# xmm6=Geps
  mulps xmm7, xmm2 ;# xmm7=Heps2
  addps xmm5, xmm6
  addps xmm5, xmm7 ;# xmm5=Fp
  mulps xmm7, [esp + nb302_two]

  xorps xmm3, xmm3
  ;# fetch charges to xmm3 (temporary)
  movss xmm3, [esp + nb302_qqOO]
  movhps xmm3, [esp + nb302_qqOH]
  addps xmm7, xmm6
  addps xmm7, xmm5 ;# xmm7=FF
  mulps xmm5, xmm1 ;# xmm5=eps*Fp
  addps xmm5, xmm4 ;# xmm5=VV
  mulps xmm5, xmm3 ;# vcoul=qq*VV
  mulps xmm3, xmm7 ;# fijC=FF*qq

  addps xmm5, [esp + nb302_vctot]
  movaps [esp + nb302_vctot], xmm5
  xorps xmm2, xmm2
  mulps xmm3, [esp + nb302_tsc]

  subps xmm2, xmm3
  mulps xmm0, xmm2

  movaps xmm1, xmm0
  movaps xmm2, xmm0

  mulps xmm0, [esp + nb302_dxOO]
  mulps xmm1, [esp + nb302_dyOO]
  mulps xmm2, [esp + nb302_dzOO]
  ;# initial update for j forces
  xorps xmm3, xmm3
  xorps xmm4, xmm4
  xorps xmm5, xmm5
  subps xmm3, xmm0
  subps xmm4, xmm1
  subps xmm5, xmm2
  movaps [esp + nb302_fjxO], xmm3
  movaps [esp + nb302_fjyO], xmm4
  movaps [esp + nb302_fjzO], xmm5
  addps xmm0, [esp + nb302_fixO]
  addps xmm1, [esp + nb302_fiyO]
  addps xmm2, [esp + nb302_fizO]
  movaps [esp + nb302_fixO], xmm0
  movaps [esp + nb302_fiyO], xmm1
  movaps [esp + nb302_fizO], xmm2

  movaps xmm0, [esp + nb302_ixH1]
  movaps xmm1, [esp + nb302_iyH1]
  movaps xmm2, [esp + nb302_izH1]
movaps xmm3, [esp + nb302_ixH2]
movaps xmm4, [esp + nb302_iyH2]
movaps xmm5, [esp + nb302_izH2]
subps xmm0, [esp + nb302_jxO]
subps xmm1, [esp + nb302_jyO]
subps xmm2, [esp + nb302_jzO]
subps xmm3, [esp + nb302_jxO]
subps xmm4, [esp + nb302_jyO]
subps xmm5, [esp + nb302_jzO]
movaps [esp + nb302_dxH1O], xmm0
movaps [esp + nb302_dyH1O], xmm1
movaps [esp + nb302_dzH1O], xmm2
movaps [esp + nb302_dxH2O], xmm3
movaps [esp + nb302_dyH2O], xmm4
movaps [esp + nb302_dzH2O], xmm5
mulps xmm0, xmm0
mulps xmm1, xmm1
mulps xmm2, xmm2
mulps xmm3, xmm3
mulps xmm4, xmm4
mulps xmm5, xmm5
addps xmm0, xmm1
addps xmm4, xmm3
addps xmm0, xmm2 ;# have rsqH1 in xmm0
addps xmm4, xmm5 ;# have rsqH2 in xmm4

;# start with H1, save H2 data
movaps [esp + nb302_rsqH2O], xmm4

;# do invsqrt
rsqrtps xmm1, xmm0
rsqrtps xmm5, xmm4
movaps xmm2, xmm1
movaps xmm6, xmm5
mulps xmm1, xmm1
mulps xmm5, xmm5
movaps xmm3, [esp + nb302_three]
movaps xmm7, xmm3
mulps xmm1, xmm0
mulps xmm5, xmm4
subps xmm3, xmm1
subps xmm7, xmm5
mulps xmm3, xmm2
mulps xmm7, xmm6
mulps xmm3, [esp + nb302_half]
mulps xmm7, [esp + nb302_half]

;# start with H1, save H2 data
movaps [esp + nb302_rinvH2O], xmm7

movaps xmm1, xmm3
mulps xmm1, xmm0 ;# xmm1=r
movaps xmm0, xmm3 ;# xmm0=rinv
mulps xmm1, [esp + nb302_tsc]

movhlps xmm2, xmm1
cvttps2pi mm6, xmm1
cvttps2pi mm7, xmm2
cvtpi2ps xmm3, mm6
cvtpi2ps xmm2, mm7
movlhps xmm3, xmm2
subps xmm1, xmm3 ;# xmm1=eps
movaps xmm2, xmm1
mulps xmm2, xmm2 ;# xmm2=eps2
pslld mm6, 2
pslld mm7, 2

movd ebx, mm6
movd ecx, mm7
```

```
  psrlq mm7, 32
  movd edx, mm7

  movlps xmm5, [esi + ebx*4]
  movlps xmm7, [esi + ecx*4]
  movhps xmm7, [esi + edx*4]
  shufps xmm4, xmm7, 136
  shufps xmm5, xmm7, 221

  movlps xmm7, [esi + ebx*4 + 8]
  movlps xmm3, [esi + ecx*4 + 8]
  movhps xmm3, [esi + edx*4 + 8] ;# other
half of coulomb table
  movaps xmm6, xmm7
  shufps xmm6, xmm3, 136
  shufps xmm7, xmm3, 221
  ;# coulomb table ready, in xmm4-xmm7
  mulps xmm6, xmm1 ;# xmm6=Geps
  mulps xmm7, xmm2 ;# xmm7=Heps2
  addps xmm5, xmm6
  addps xmm5, xmm7 ;# xmm5=Fp
  mulps xmm7, [esp + nb302_two]

  xorps xmm3, xmm3
  ;# fetch charges to xmm3 (temporary)
  movss xmm3, [esp + nb302_qqOH]
  movhps xmm3, [esp + nb302_qqHH]

  addps xmm7, xmm6
  addps xmm7, xmm5 ;# xmm7=FF
  mulps xmm5, xmm1 ;# xmm5=eps*Fp
  addps xmm5, xmm4 ;# xmm5=VV
  mulps xmm5, xmm3 ;# vcoul=qq*VV
  mulps xmm3, xmm7 ;# fijC=FF*qq

  addps xmm5, [esp + nb302_vctot]
  movaps [esp + nb302_vctot], xmm5

  xorps xmm1, xmm1

  mulps xmm3, [esp + nb302_tsc]
  mulps xmm3, xmm0
  subps xmm1, xmm3

  movaps xmm0, xmm1
  movaps xmm2, xmm1
  mulps xmm0, [esp + nb302_dxH1O]
  mulps xmm1, [esp + nb302_dyH1O]
  mulps xmm2, [esp + nb302_dzH1O]
  ;# update forces H1 - j water
  movaps xmm3, [esp + nb302_fjxO]
  movaps xmm4, [esp + nb302_fjyO]
  movaps xmm5, [esp + nb302_fjzO]
  subps xmm3, xmm0
  subps xmm4, xmm1
  subps xmm5, xmm2
  movaps [esp + nb302_fjxO], xmm3
  movaps [esp + nb302_fjyO], xmm4
  movaps [esp + nb302_fjzO], xmm5
  addps xmm0, [esp + nb302_fixH1]
  addps xmm1, [esp + nb302_fiyH1]
  addps xmm2, [esp + nb302_fizH1]
  movaps [esp + nb302_fixH1], xmm0
  movaps [esp + nb302_fiyH1], xmm1
  movaps [esp + nb302_fizH1], xmm2
  ;# do table for H2 - j water interaction

  movaps xmm0, [esp + nb302_rinvH2O]
  movaps xmm1, [esp + nb302_rsqH2O]
  mulps xmm1, xmm0 ;# xmm0=rinv, xmm1=r
  mulps xmm1, [esp + nb302_tsc]

  movhlps xmm2, xmm1
  cvttps2pi mm6, xmm1
  cvttps2pi mm7, xmm2
  cvtpi2ps xmm3, mm6
  cvtpi2ps xmm2, mm7
  movlhps xmm3, xmm2
  subps xmm1, xmm3 ;# xmm1=eps
  movaps xmm2, xmm1
  mulps xmm2, xmm2 ;# xmm2=eps2
  pslld mm6, 2
  pslld mm7, 2

  movd ebx, mm6
  movd ecx, mm7
  psrlq mm7, 32
  movd edx, mm7

  movlps xmm5, [esi + ebx*4]
  movlps xmm7, [esi + ecx*4]
  movhps xmm7, [esi + edx*4] ;# got half
coulomb table
  movaps xmm4, xmm5
  shufps xmm4, xmm7, 136
  shufps xmm5, xmm7, 221
  movlps xmm7, [esi + ebx*4 + 8]
  movlps xmm3, [esi + ecx*4 + 8]
  movhps xmm3, [esi + edx*4 + 8] ;# other
half of coulomb table
  movaps xmm6, xmm7
  shufps xmm6, xmm3, 136
  shufps xmm7, xmm3, 221
  ;# coulomb table ready, in xmm4-xmm7
  mulps xmm6, xmm1 ;# xmm6=Geps
  mulps xmm7, xmm2 ;# xmm7=Heps2
  addps xmm5, xmm6
  addps xmm5, xmm7 ;# xmm5=Fp
  mulps xmm7, [esp + nb302_two]

  xorps xmm3, xmm3
  ;# fetch charges to xmm3 (temporary)
  movss xmm3, [esp + nb302_qqOH]
  movhps xmm3, [esp + nb302_qqHH]
  addps xmm7, xmm6
  addps xmm7, xmm5 ;# xmm7=FF
  mulps xmm5, xmm1 ;# xmm5=eps*Fp
  addps xmm5, xmm4 ;# xmm5=VV
  mulps xmm5, xmm3 ;# vcoul=qq*VV
  mulps xmm3, xmm7 ;# fijC=FF*qq

  addps xmm5, [esp + nb302_vctot]
  movaps [esp + nb302_vctot], xmm5

  xorps xmm1, xmm1
```

```
  mulps xmm3, [esp + nb302_tsc]
  mulps xmm3, xmm0
  subps xmm1, xmm3

  movaps xmm0, xmm1
  movaps xmm2, xmm1

  mulps xmm0, [esp + nb302_dxH2O]
  mulps xmm1, [esp + nb302_dyH2O]
  mulps xmm2, [esp + nb302_dzH2O]
  movaps xmm3, [esp + nb302_fjxO]
  movaps xmm4, [esp + nb302_fjyO]
  movaps xmm5, [esp + nb302_fjzO]
  subps xmm3, xmm0
  subps xmm4, xmm1
  subps xmm5, xmm2
  mov esi, [ebp + nb302_faction]
  movaps [esp + nb302_fjxO], xmm3
  movaps [esp + nb302_fjyO], xmm4
  movaps [esp + nb302_fjzO], xmm5
  addps xmm0, [esp + nb302_fixH2]
  addps xmm1, [esp + nb302_fiyH2]
  addps xmm2, [esp + nb302_fizH2]
  movaps [esp + nb302_fixH2], xmm0
  movaps [esp + nb302_fiyH2], xmm1
  movaps [esp + nb302_fizH2], xmm2

  movlps xmm0, [esi + eax*4]
  movlps xmm1, [esi + eax*4 + 12]
  movhps xmm1, [esi + eax*4 + 24]
  movaps xmm3, [esp + nb302_fjxO]
  movaps xmm4, [esp + nb302_fjyO]
  movaps xmm5, [esp + nb302_fjzO]
  movaps xmm6, xmm5
  movaps xmm7, xmm5
  shufps xmm6, xmm6, 2
  shufps xmm7, xmm7, 3
  addss xmm5, [esi + eax*4 + 8]
  addss xmm6, [esi + eax*4 + 20]
  addss xmm7, [esi + eax*4 + 32]
  movss [esi + eax*4 + 8], xmm5
  movss [esi + eax*4 + 20], xmm6
  movss [esi + eax*4 + 32], xmm7
  movaps xmm5, xmm3
  unpcklps xmm3, xmm4
  unpckhps xmm5, xmm4
  addps xmm0, xmm3
  addps xmm1, xmm5
  movlps [esi + eax*4], xmm0
  movlps [esi + eax*4 + 12], xmm1
  movhps [esi + eax*4 + 24], xmm1

  dec dword ptr [esp + nb302_innerk]
  jz .nb302_updateouterdata
  jmp .nb302_single_loop
```