Otimizações com Assembly x86

Visão Geral

1a. Regra de Otimização Manual

- **Não faça**
  - A maior parte do código não terá ganhos perceptíveis ao se fazer otimizações em nível de assembly
  - Os maiores ganhos normalmente vêm de utilizar as estruturas de dados e algoritmos mais apropriados para cada problema
    - Complexidade de tempo, uso de memória
  - Otimizações prematuras podem complicar sobremaneira o entendimento do código e dificultar a sua manutenção
    - Por vezes capacidade de processamento é mais barato que horas de trabalho

2a. Regra de Otimização Manual

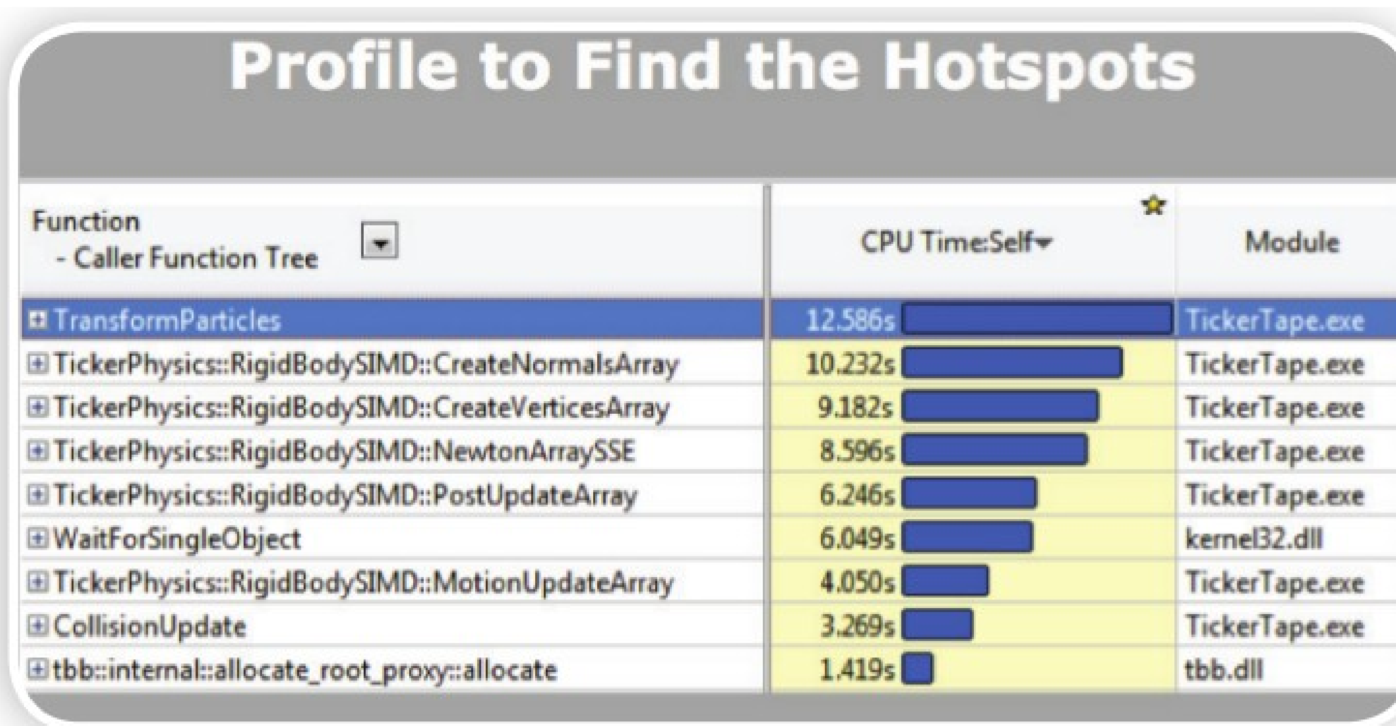
- **Não faça <u>ainda</u>**
  - Uma vez que você tenha um código que use intensivamente o processador e que creia que pode se beneficiar de otimizações manuais, não comece otimizando baseado em “achismos”:
    - “Acho que é importante otimizar essa função”...
  - Use ferramentas que permitam identificar que trechos de código seriam mais beneficiados
    - Normalmente você precisará de um Profiler
  - Depois que você identificar os “hot spots” do código, você pode passar a otimizá-los.
  - “Don't waste your time trying to fix what isn't broken”!

Exemplo: TickerTape

- Simula o movimento de folhas ao vento usando cálculos complexos aerodinâmicos
  - http://software.intel.com/en-us/vcsource/samples/ticker-tape
  - Milhares de cálculos de ponto flutuante são necessários a cada instante
- A arquitetura da aplicação já foi pensada para ter uma implementação apropriada para multi-core
  - Múltiplos threads
- Depois foi tratado como obter o desempenho máximo possível em cada core

Exemplo: TickerTape

- Uso de um profiler para identificar funções que estão consumindo mais tempo:


Profile to Find the Hotspots

Function - Caller Function Tree	CPU Time:Self	Module
TransformParticles	12.586s	TickerTape.exe
TickerPhysics::RigidBodySIMD::CreateNormalsArray	10.232s	TickerTape.exe
TickerPhysics::RigidBodySIMD::CreateVerticesArray	9.182s	TickerTape.exe
TickerPhysics::RigidBodySIMD::NewtonArraySSE	8.596s	TickerTape.exe
TickerPhysics::RigidBodySIMD::PostUpdateArray	6.246s	TickerTape.exe
WaitForSingleObject	6.049s	kernel32.dll
TickerPhysics::RigidBodySIMD::MotionUpdateArray	4.050s	TickerTape.exe
CollisionUpdate	3.269s	TickerTape.exe
tbb::internal::allocate_root_proxy::allocate	1.419s	tbb.dll

Figure 16. Intel® Parallel Studio presents an analysis of where functions are spending time.

TransformParticles é a função que está consumindo mais tempo

Exemplo: TickerTape

- O profiler ainda permite fazer um “drill-down” nessa função, mostrando uma análise linha por linha

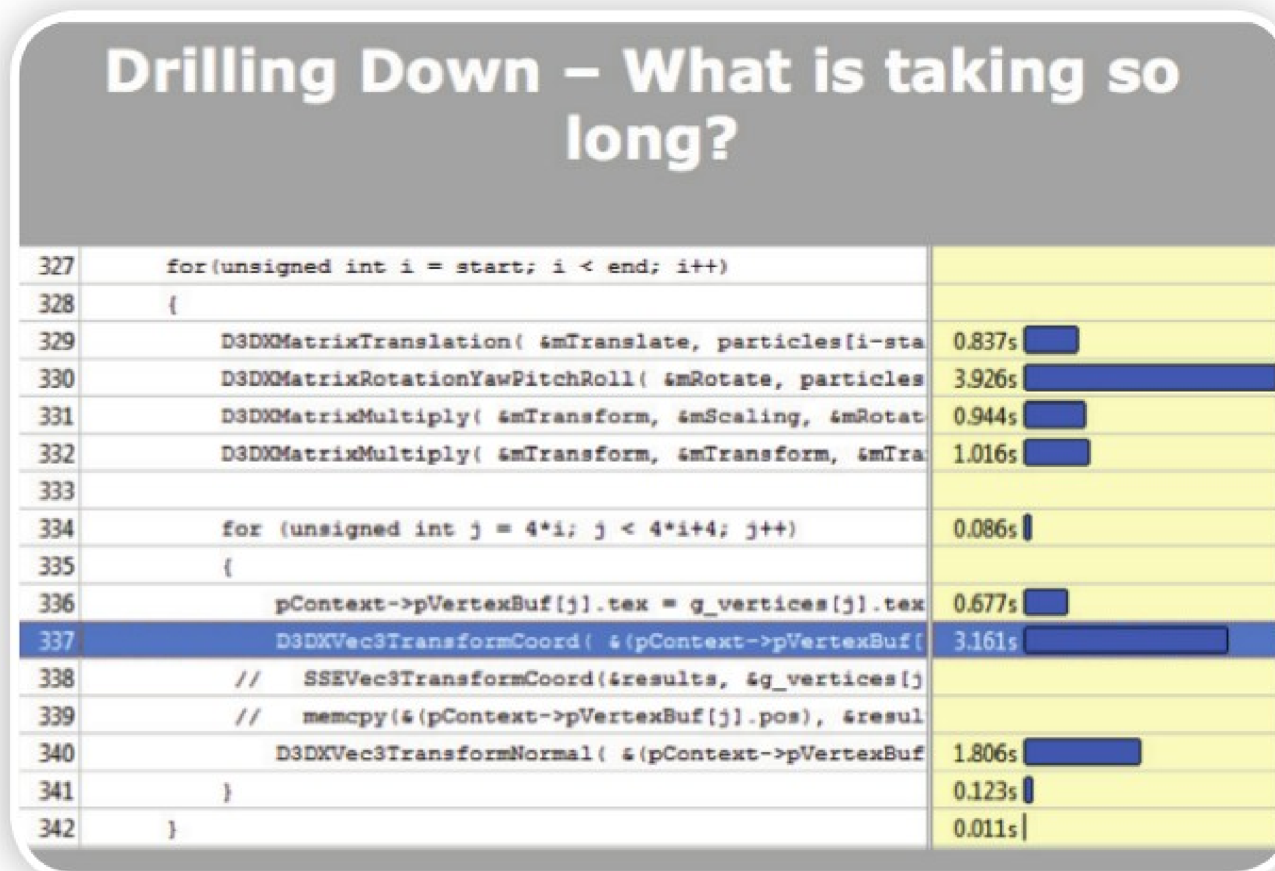


Figure 17. Drilling into slow-performing functions reveals the underlying code.

Neste caso vê-se que as funções `D3DXMatrixRotationYawPitchRoll` e `D3DXVec3TransformCoord` são as mais custosas. Decidiu-se otimizar a última, reimplementando-a com uso de instruções SSE

x86: Diretrizes gerais para otimização

- O efeito de otimizações pode depender da microarquitetura do processador visado
  - Por exemplo, vimos que arquiteturas Nehalem têm otimizações para trabalhar com dados não alinhados na memória, enquanto que as anteriores não as tinham.
  - Para ter um código rápido em qualquer situação, é recomendável então alinhar
    - Mas se o código for rodar somente em arquiteturas Nehalem isso não seria necessário...

x86: Diretrizes gerais para otimização

- For maximum performance, take advantage of the best available features on each processor such as Hyper-Threading Technology, streaming SIMD extensions, and increased cache size. Where optimum performance on all processor generations is desired, applications can use the CPUID instruction to identify those features and allow you to integrate processor-specific instructions (such as SSE2) or additional threads in the case of Hyper-Threading Technology enabled processors into the source code where appropriate.

x86: Diretrizes gerais para otimização

- For some compilers, there is also the capability for the compiler to help you transparently use available features on the latest x86 processors.
  - For example, the Intel® C++ Compiler supports the integration of different versions of the code (SSE, SSE2) for each target processor within the same binary library or executable instructions. The selection of which code to execute at runtime is made based on the CPU identifier that is read with the CPUID instruction. Binary code targeted for different processor generations can either be generated under the control of the programmer or automatically by the compiler.
  - But be aware that if you use automatic code targeting in the Intel Compiler, for instance, the optimized code may be executed only on Intel CPUs, even if the Other Manufacturer's CPU supports it too.

What about optimizing for Power?

- Em geral compiladores fornecem opções de otimização que priorizarão:
  - Velocidade ou
  - Tamanho do código
- É possível em tese calcular o quanto de potência determinado conjunto de instruções irá consumir, e assim otimizar para eficiência energética
  - Isso é um campo de pesquisa em florescimento, encontra-se vários artigos na área nos últimos anos

What about optimizing for Power?

- No entanto, isso ainda é muito complexo para ser levado em conta pelos compiladores atuais, então hoje, na prática, assume-se que otimizar para velocidade também é otimizar para energia, uma vez que o processador vai terminar o trabalho mais rápido

> Typically, code optimized for performance is also good for power since you may complete a task faster and enter an idle or low power state earlier or be able to run the same application at a lower voltage and frequency for the same performance level. In general, performance optimizations tend to reduce power consumption as long as the overall runtime or CPU utilization is reduced allowing a mobile processor to run in a lower power state. Keep in mind however that the processor is only one part of overall power consumption in a mobile PC and there are other ways to reduce overall power consumption by involving other system components such the hard drive, CD-DVD drives and LCD display
>
> http://software.intel.com/en-us/articles/optimizing-software-for-intel-centrinor-mobile-technology-and-intel-netburstt-microarchitecture

A maneira mais fácil de otimizar...

- Usar os switches que ligam otimizações do compilador adequadas à microarquitetura em que o código vai rodar e "backwards-compatible"
  - Exemplo: Switch G7 do compilador Intel otimiza para microarquiteturas P6, P7 e posteriores, sem utilizar nenhuma instrução específica delas
    - Possivelmente adota esquema 4-1-1 de µ-ops...

A maneira mais fácil de otimizar...

- Usar os switches que habilitam vetorização automática do código
  - Exemplos (Intel Compiler):
    - -Qax{M|K|W} option produces optimized code with a generic code path for non-compatible processors (M = MMX, K = SSE, W = SSE2)
    - -Qx{M|K|W} option switch which produces optimized code only for the selected target instruction set as indicated above

Maneiras menos fáceis...

Algumas dicas práticas

Optimize Branch Predictability and Performance

- If your program has a significant amount of mispredicted branches, branch optimizations may offer a great deal of potential performance improvement. Understanding the flow of branches and improving the predictability of branches can improve performance significantly. Using a Profiler (such as VTune), try to identify areas of your code that may have a larger percentage of mispredicted branches so that you can focus on the real problem areas in your code.

Optimize Branch Predictability and Performance

- Eliminate Unnecessary Branches
  - In modern x86 processors, eliminating unnecessary branches improves performance by reducing mispredictions and reduces impact on hardware branch prediction resources. Note that every branch affects performance since even correctly predicted branches reduce the amount of useful code delivered to the processor.
  - Possible ways to eliminate branches include optimizing code flow, making basic blocks contiguous, enabling compiler switches to make use of the cmov and setcc instructions and with other code optimizations.

Optimize Branch Predictability and Performance

- Eliminate Unnecessary Branches (exemplo 1)

```
cmp a, b    ; Condition
jbe L30     ; Conditional branch
mov ebx const1 ; ebx holds X
jmp L31     ; Unconditional branch
L30:
mov ebx, const2
L31:
```

Código com branches

Se a<=b, entao ebx <- const2
Se não, ebx <- const1

Optimize Branch Predictability and Performance

- Eliminate Unnecessary Branches (exemplo 1)

```
xor ebx, ebx   ; Clear ebx (X in the C code)
cmp A, B
setge bl   ; When ebx = 0 or 1
        ; OR the complement condition
sub ebx, 1 ; ebx=11...11 or 00...00
and ebx, CONST3; CONST3 = CONST1-CONST2
add ebx, CONST2; ebx=CONST1 or CONST2
```

Se a<=b, entao ebx <- const2
Se não, ebx <- const1

Código ninja :-)

The optimized code sets EBX to zero, then compares A and B. If A is greater than or equal to B, EBX is set to one. Then EBX is decreased and AND'd with the difference of the constant values. This sets EBX to either zero or the difference of the values. By adding CONST2 back to EBX, the correct value is written to EBX. When CONST2 is equal to zero, the last instruction can be deleted.

Optimize Branch Predictability and Performance

- Eliminate Unnecessary Branches (exemplo 2)

```
test ecx, ebx
jne label
mov eax, ebx
label:
```

Código com branches

Se ecx = ebx, entao eax <- ebx

Optimize Branch Predictability and Performance

- Eliminate Unnecessary Branches (exemplo 2)

```
test ecx, ebx
; Test the flags
cmoveq eax, ebx
; If the equal flag is set, move
; ebx to eax- the label: tag no longer needed
```

Código sem branches

Se ecx = ebx, entao eax <- ebx

The CMOV and FCMOV instructions are available on the Pentium II and subsequent processors, but not on Pentium processors and earlier IA-32 processors. Be sure to check whether a processor supports these instructions with the CPUID instruction.

Optimize Branch Predictability and Performance

- Arrange Code to Improve Branch Predictability
  - Sometimes the most common code path of conditional expressions as they are initially written in code doesn't match up very well with processor branch prediction algorithms. **When a new branch instruction is encountered for the first time in the processor, it is assumed that backward branches will be taken and forward branches will not be taken**. You can improve branch predictability and optimize instruction prefetching by arranging code to be consistent with these static branch prediction assumptions. In the case of a if–else statement for example, this would involve rewriting the statement so that the if and else portion are switched so that the if condition is the most common case.

Optimize Branch Predictability and Performance

- Arrange Code to Improve Branch Predictability
  - Sometimes the most common code path of conditional expressions as they are initially written in code doesn't match up very well with processor branch prediction algorithms. **When a new branch instruction is encountered for the first time in the processor, it is assumed that backward branches will be taken and forward branches will not be taken**. You can improve branch predictability and optimize instruction prefetching by arranging code to be consistent with these static branch prediction assumptions. In the case of a if–else statement for example, this would involve rewriting the statement so that the if and else portion are switched so that the if condition is the most common case.

Arrange Code to Improve Branch Predictability (Exemplos)

```
01 // forward conditional branches not taken (fall through)
02
03 // Branch statically predicted to run if condition and not else
04
05 if <condition> {
06
07 ...
08
09 }
10
11 else
12
13 {
14
15 }
16
17
18 // For loop fall through predicted as taken
19
20 for <condition> {
21
22 ...
23
24 }
25
26
27 // Backward Conditional Branches are predicted taken.
28
29 // In each of these cases, the backward branches at the bottom of
30
31 // the loop are predicted to be taken
32
33
34 // For loop bottom predicted as taken to the top
35
36 for <condition> {
37
38 ...
39
40 }
41
42 loop {
43
44 ...
45
46 } <condition>
```

Inline Functions According to Coding Recommendations

- Generally, small functions that would suffer from an excess amount of call overhead if they weren't inlined or C++ Get/Set methods should continue to be declared with the a inline qualifier.
- Current generation compilers are best able to determine when performance is improved due to function inlining. By compiling with the recommended settings, automatic function inlining is likely to improve rather than degrade performance as excessive manual inlining can sometimes do. Where possible, also avoid indirect calls and virtual functions in C++ since these calls incur greater overhead at function call time.

Optimizing Memory Accesses

Memory optimizations can also improve performace significantly...

Optimize for Increased Cache Size

- In high performance applications, take advantage of increased cache sizes where possible by dynamically detecting cache size at runtime with the cupid instruction and adjusting performance critical code accordingly. Optimize data structures to either fit in one-half of the first-level cache or in the second-level cache. Optimizing for one-half of the first-level cache will bring the greatest performance benefit. If one-half of the first-level cache is too small to be practical, optimize for the second-level cache. Optimizing for a point in between (for example, for the entire first-level cache) will likely not bring a substantial improvement over optimizing for the second-level cache.
- Although current compilers often do a good job enhancing locality, also consider using manual techniques

Prefetch Data

- Enable the prefetch generation in your compiler by using the /QxW or /QaxW switch in the Intel C++ compiler or the /arch:SSE2 switch in the .Net 2003 C++ compiler. As the compiler's prefetch implementation improves, automatic prefetch insertion by the compiler may outperform manual insertion except for code tuning experts. If you are using a compiler that does not support software prefetching, intrinsics or inline assembly may be used to manually insert prefetch instructions.
- If a load is found to miss frequently with significant negative performance impact, first try moving the load up to execute either. If that change doesn't reduce the amount of load misses, insert a prefetch before the load of the data. Be aware that manual prefetch is independent of the hardware prefetching capabilities in both the Netburst microarchitecture and Pentium M processors. These mechanisms are separate and hardware prefetch is not improved by manual prefetching and excessive manual prefetching can degrade performance.

Align and Organize Data for Better Performance

- Unaligned data can be another potentially serious performance problem. The guidelines in this section helps you minimize performance losses due to unaligned data. It is important to remember to focus on data elements in the most CPU intensive parts of your program.
- Align data on natural operand size address boundaries. If the data will be accessed with vector instruction loads and stores, align the data on 16 byte boundaries. For best performance, align data as follows:
  - Align 8-bit data at any address.
  - Align 16-bit data to be contained within an aligned four byte word.
  - Align 32-bit data so that its base address is a multiple of four.
  - Align 64-bit data so that its base address is a multiple of eight.
  - Align 80-bit data so that its base address is a multiple of sixteen.
  - Align 128-bit data so that its base address is a multiple of sixteen.

Align and Organize Data for Better Performance

- Also, pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary. If the operands are packed in a SIMD instruction, align to the packed element size (64- or 128-bit). Align data by providing padding inside structures and arrays. Programmers can reorganize structures and arrays to minimize the amount of memory wasted by padding.
- The __declspec(align(sizeInBytes)) pragma is supported in both the Intel and Microsoft compilers, and causes the linker to align variables with the specified alignment. For example, the following declaration ensures that the variable signMask is aligned on a 16 byte boundary suitable for vector instruction processing.
  - __declspec(align(16)) static const int signMask[4] = {-1,-1,-1,-1};
    - No gcc: static const int signMask[4] __attribute__ ((aligned (16))) = {-1,-1,-1,-1} ;

Floating Point/SIMD Tips

How can you make sure that your application has great floating point performance?

1st: Use the compiler!

- As mentioned earlier, the quickest way to optimize floating point intensive programs is to enable the compiler's use of SIMD instructions with appropriate switches. These switches help your application take advantage of the SIMD capabilities of Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2) instructions etc.

Small is better

- Use the smallest possible floating-point or SIMD data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible and short integers instead of long integers.
- Do not use double precision unless necessary. Set the precision control (PC) field in the x87 FPU control word to "Single Precision". This allows single precision (32-bit) computation to complete faster on some operations (for example, divides due to early out)

Dependencies are BAD

- Dependence chains can sometimes impact performance by introducing artificial dependencies that are an artifact of how an expression is written and not true data dependencies. For best performance, break dependence chains where possible. The following example shows an example dependence chain and a simple rewrite that helps overall performance and parallelism.

To calculate z = a + b + c + d, instead of

```
x = a + b;
y = x + c;
z = y + d;
```

use

```
x = a + b;
y = c + d;
z = x + y;
```

For More Information:

- Intel® 64 and IA-32 Architectures Optimization Reference Manual (google for it)
- Optimizing Software for Intel® Centrino® Mobile Technology and Intel® NetBurst™ Microarchitecture: http://goo.gl/nLVyS