Assembly Questions

Question Responses for Assignment 3

Chris Klassen & Alex Kabak

2016

# Question A

When comparing the NEON intrinsics code to the C++ implementation, the average times for processing are as follows:

**C++**: 31.280 milliseconds

**NEON intrinsics**: 13.500 milliseconds

The NEON intrinsics implementation outperforms the C++ implementation by nearly 3x, making it a much better choice.

# Question B

Compared to the C++ and NEON intrinsics implementations, the NEON assembly implementation’s average processing time was:

**NEON assembly**: 4.500 milliseconds

Clearly, implementing NEON via raw assembly code bypasses some of the inefficiencies of the compiler, allowing for an even better performance increase and making NEON assembly the best choice in this case.

# Question C

The ARM neon code implemented is slower than the OpenGL ES function for matrix multiplication. This is more than likely the case because OpenGL is far better optimized than my hand-coded neon assembly intrinsics; additionally, it’s also possible the compiler is generating more optimized code. I believe there is still an opportunity to achieve better performance through hand-coded neon instructions, and this would be done by reworking the current solution.

# Question G

## Pass by Value Function

mov eax, DWORD PTR \_number$[ebp]

push eax

call ?plusOne@@YAHH@Z

add esp, 4

mov DWORD PTR \_number$[ebp], eax

## Pass by Reference Function

lea eax, DWORD PTR \_number$[ebp]

push eax

call ?plusOneRef@@YAHAAH@Z

add esp, 4

mov DWORD PTR \_number$[ebp], eax

## Differences

When passing by value, the assembler moves the value **number** to eax and then pushes it to the stack so that it can be used by the function. After the function returns, the number is pulled out of eax (where it was returned to) and assigned to **number** once again.

When passing by reference, the assembler retrieves the address of **number** and pushes it to the stack. This address is then pushed to the stack to be used by the function in retrieving the actual value of the number. After the function returns, the return value is assigned to **number** normally.

# Question H

## Function Code

?timesTwo@@YAHXZ PROC

push ebp

mov ebp, esp

sub esp, 192 push ebx

push esi

push edi

lea edi, DWORD PTR [ebp-192]

mov ecx, 48

mov eax, -858993460

rep stosd

mov eax, 12

pop edi

pop esi

pop ebx

mov esp, ebp

pop ebp

ret 0

?timesTwo@@YAHXZ ENDP

## Inline Code

mov DWORD PTR \_numberB$[ebp], 12

## Differences

In the function code, quite a bit of overhead work must be done besides the very simple mathematical calculation. The function must store the existing base pointer, and also update the base pointer and stack pointer and other registers on the CPU. Then, it can calculate the result and put it in eax to be returned. After this is done, cleanup must be done to ensure that when the function returns, the program can continue properly.

In the inline code, the assembler simply puts the mathematical result into **numberB** and continues on its way, with far less overhead.