HARDWARE (COMPILERS, OS, RUNTIME) AND C, C++, TO CUDA

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

Abstract. I review what, how, and why Segmentation Fault is and occurs in the specific context of C and hopefully to CUDA, virtual memory addresses in C++, and to CUDA.

1. Introduction; why I'm writing this

I didn't realize the importance of having a profound understanding of C/C++ and its relation to hardware, in particular memory (addresses), until I learned more specifically about the work done at NVIDIA from its news and job postings. Coming from a theoretical and mathematical physics background, I wanted to get myself up to speed as fast as possible, provide good textbook and online references, and directly relate these topics, which seem to be classic topics in computer science (at hardware level) and electrical engineering, to CUDA, to specifically GPU parallel programming with direct CUDA examples.

Note that there is a version of this LaTeX/pdf file in a "grande" format (not letter format but "landscape" format) that is exactly the same as the content here, but with different size dimensions.

Meyer (1997) [1]

Date: 1 Nov 2017.

Key words and phrases. C, C++, CUDA, CUDA C/C++, Compilers, OS, runtime, classes, Object Oriented Programming, Segmentation Fault, memory, memory addresses.

> "When programmers first began giving up assembly language in favor of using HLLs, they generally understood the low-level ramifications of the HLL statements they were using and could choose their HLL statements appropriately.Unfortunately, the generation of computer programmers that followed them did not have the benefit of mastering assembly language. As such, they were not in a position to wisely choose statements and data structures that HLLs could efficiently translate into machine code."

pp. 2, Ch. 1 of Hyde (2006) [3], where HLL stands for high-level language. I was part of that generation and knew of nothing of assembly.

2. 80x86 Assembly; 80x86 Architecture

cf. Ch. 3 of Hyde (2006) [3]

Main difference between complex instruction set computer (CISC) architectures such as 80x86 and reduced instruction set computer (RISC) architectures like PowerPC is the way they use memory. RISC provde relatively clumsy memory access, and applications avoid accessing memory. 80x86 access memory in many different ways and applications take advantage of these facilities.

2.1. Basic 80x86 Architecture.

Intel CPU generally classified as a *Von Neumann machine*, containing 3 main building blocks:

(1) CPU
(2) memory
(3) input/output (I/O) devices

These 3 components are connected together using the *system bus*. System bus consists of

- address bus
- data bus
- control bus

CPU communicates with memory and I/O devices by placing a numeric value on the address bus to select 1 of the memory locations or I/O device port locations, each of which has a unique binary numeric address.
Then the CPU, I/O, and memory devices pass data among themselves by placing data on the data bus.
Control bus contains signals that determine the direction of data transfer (to or from memory, and to or from an I/O device).

So

$$\textbf{Memory}$$

$$\text{Obj}(\textbf{Memory}) = \text{ memory locations}$$

$$\textbf{I/O}$$

$$\text{Obj}(\textbf{I/O}) = \text{ I/O device port locations}$$

and

$$\textbf{address bus}$$

$$\text{Obj}(\textbf{address bus}) = \text{ unique binary numeric addresses}$$

Likewise,

$$\textbf{data bus}$$

$$\text{Obj}(\textbf{data bus}) = \text{ data}$$

And so

$$\text{CPU} : \textbf{Memory} \times \textbf{address bus} \to \text{Hom}(\textbf{Memory}, \textbf{address bus})$$

$$\text{CPU} : (\text{memory location}, \text{unique binary numeric address}) \mapsto \&$$

in the language of category theory.

2.2. **Registers (for 80x86).** cf. 3.3 Basic 80x86 Architecture, from pp. 23 of Hyde (2006) [3].

Example: to add 2 variables, x, y and store $x + y$ into z, you must load 1 of the variables into a register, add the 2nd. operand to the register, and then store register's value into destination variable. "Registers are middlemen in almost every calculation." Hyde (2006) [3].

There are 4 categories of 80x86 CPU registers:

- general-purpose
- special-purpose application-accessible
- segment
- special-purpose kernel-mode

Segment registers not used very much in modern 32-bit operating systems (e.g. Windows, Linux; what about 64-bit?) and special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools.

2.2.1. *80x86 General-Purpose Registers.* 80x86 CPUs provide several general-purpose registers, including 8 32-bit registers having the following names: (8 bits in 1 byte, 32-bit is 4 bytes)

$$EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP$$

where E is *extended*; 32-bit registers distinguished from 8 16-bit registers with following names

$$AX, BX, CX, DX, SI, DI, BP, SP$$

and finally

$$AL, AH, BL, BH, CL, CH, DL, DH$$

Hyde says that it's important to note about the general-purpose registers is they're not independent. 80x86 doesn't provide 24 separate registers, but overlaps the 32-bit registers with 16-bit registers, and overlaps 16-bit registers with 8-bit registers. e.g.

$$\&AL \neq \&AH, \text{ but}$$
$$\&AL, \&AH \in \&AX \text{and } \&AX \in \&EAX, \text{ so that}$$
$$\&AL, \&AH, \&AX \in \&EAX$$

3. Compilers; Compiler Operation and Code Generation

cf. Ch. 5 Compiler Operation and Code Generation, from pp. 62 and pp. 72- of Hyde (2006) [3]

4. gdp; good debugging processes (in C/C++/CUDA)

Learn C the hard way Lecture 4, Using Debugger (GDB)

5. Pointers in C; Pointers in C categorified (interpreted in Category Theory) and its relation to actual, physical, computer memory and (memory) addresses ((address) bus; pointers, structs, arrays in C

From Shaw (2015) [4], Exercise 15,

e.g. `ages[i]`, You're indexing into array `ages`, and you're using the number that's held in i to do it:

$$\begin{array}{ll} a : \mathbb{Z} \to \text{Type} \in \mathbf{Type} & \\ a : i \mapsto a[i] & \text{e.g.} \end{array} \quad \begin{array}{l} a : \mathbb{Z} \to \mathbb{R} \text{ or } \mathbb{Z} \\ a : i \mapsto a[i] \end{array}$$

Index $i \in \mathbb{Z}$ is a location *inside* `ages` or a, which can also be called *address*. Thus, $a[i]$.

Indeed, from cppreference for Member access operators,

Built-in *subscript* operator provides access to an object pointed-to by pointer or array operand. And so `E1[E2]` is exactly identical to `*(E1+E2)`.

To C, e.g. `ages`, or a, is a location in computer's memory where, e.g., all these integers (of `ages`) start, i.e. where a starts.

$\mathbf{Memory}$, $\quad \text{Obj}(\mathbf{Memory}) \ni$ memory location. Also, to specify CPU,

$\mathbf{Memory}_{CPU}$, $\quad \text{Obj}(\mathbf{Memory}_{CPU}) \ni$ computer memory location

It's *also* an address, and C compiler will replace e.g. `ages` or array a, anywhere you type it, with address of very 1st integer (or 1st element) in, e.g. `ages`, or array a.

$$\mathbf{Arrays} \overset{\cong}{\longleftrightarrow} \mathbf{address}$$

$$\text{Obj}(\mathbf{Arrays}) \overset{\cong}{\longleftrightarrow} \text{Obj}(\mathbf{address})$$

$$a \overset{\cong}{\longleftrightarrow} \texttt{0x17}$$

"But here's the trick": e.g. "`ages` is an address inside the *entire computer*." (Shaw (2015) [4]).

It's not like i that's just an address inside `ages`. `ages` array name is actually an address in the computer.

"This leads to a certain realization: C thinks your whole computer is 1 massive array of bytes."

"What C does is layer on top of this massive array of bytes the concept of types and sizes of those types." (Shaw (2015) [4]).

Let

$\mathbf{Memory}_{CPU} :=$ 1 massive array of bytes

$\text{Obj}(\mathbf{Memory}_{CPU})$

$\mathbf{Type}$

$\text{Obj}(\mathbf{Type}) \ni \texttt{int}, \texttt{char}, \texttt{float}$

$\text{Obj}(\mathbf{Type}) \xrightarrow{\texttt{sizeof}} \mathbb{Z}^+$

$T \xmapsto{\texttt{sizeof}} \texttt{sizeof}(T)$

$\texttt{float} \xmapsto{\texttt{sizeof}} \texttt{sizeof}(\texttt{float})$

How C is doing the following with your arrays:

- *Create* a block of memory inside your computer:

$$\text{Obj}(\mathbf{Memory}_{CPU}) \supset \text{ Memory block}$$

  Let $\text{Obj}(\mathbf{Memory}_{CPU})$ be an ordered set. Clearly, then memory can be indexed. Let $b \in \mathbb{Z}^+$ be this index. Then $\text{Memory block}(0) = \text{Obj}(\mathbf{Memory}_{CPU})(b)$.
- *Pointing* the name `ages`, or a, to beginning of that (memory) block.
  Entertain, possibly, a category of pointers, **Pointers** $\equiv$ **ptrs**.

$$\mathbf{ptrs}$$

$$\text{Obj}(\mathbf{ptrs}) \ni a, \text{ e.g. } \texttt{ages}$$

$$a \mapsto \text{Memory block}(0)$$

$$\text{Obj}(\mathbf{ptrs}) \to \text{Obj}(\mathbf{Memory}_{CPU})$$

- *indexing* into the block, by taking the base address of `ages`, or a

$a \overset{\cong}{\mapsto}$ base address $0x17$

$\text{Obj}((T)\mathbf{array}) \overset{\cong}{\to} \text{Obj}(\mathbf{addresses})$

$a[i] \equiv a + i \overset{\cong}{\mapsto}$ base address + i * `sizeof`$(T) \overset{*}{\mapsto} a[i] \in T$ where T, e.g. $T = \mathbb{Z}, \mathbb{R}$

$\text{Obj}((T)\mathbf{array} \overset{\cong}{\to} \text{Obj}(\mathbf{addresses}) \to T$

"A pointer is simply an address pointing somewhere inside computer's memory with a type specifier." Shaw (2015) [4]

C knows where pointers are pointing, data type they point at, size of those types, and how to get the data for you.

5.0.1. *Practical Pointer Usage.*

- Ask OS for memory block (chunk of memory) and use pointer to work with it. This includes strings and `structs`.
- Pass by reference - pass large memory blocks (like large structs) to functions with a pointer, so you don't have to pass the entire thing to the function.
- Take the address of a function, for dynamic callback. (function of functions)

"You should go with arrays whenever you can, and then only use pointers as performance optimization if you absolutely have to." Shaw (2015) [4]

5.0.2. *Pointers are not Arrays.* No matter what, pointers and arrays are not the same thing, even though C lets you work with them in many of the same ways.

From Eli Bendersky's website, Are pointers and arrays equivalent in C?

He also emphasizes that

5.0.3. *Variable names in C are just labels.* "A variable in C is just a convenient, alphanumeric pseudonym of a memory location." (Bendersky, [5]). What a compiler does is, create label in some memory location, and then access this label instead of always hardcoding the memory value.

"Well, actually the address is not hard-coded in an absolute way because of loading and relocation issues, but for the sake of this discussion we don't have to get into these details." (Bendersky, [5]) (EY : 20171109 so it's on the address bus?)

Compiler assigns label *at compile time.* Thus, the great difference between arrays and pointers in C.

5.0.4. *Arrays passed to functions are converted to pointers.* cf. Bendersky, [5].

Arrays passed into functions are always converted into pointers. The argument declaration `char arr_place[]` in

```
void foo(char arr_arg[], char* ptr_arg)
{
char a = arr_arg[7];
char b = ptr_arg[7];
}
```

is just syntactic sugar to stand for `char* arr_place`.

From Kernighan and Ritchie (1988) [6], pp. 99 of Sec. 5.3, Ch. 5 Pointers and Arrays,

> When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.

Why?

The C compiler has no choice here, since,
array name is a label the C compiler replaces *at compile time* with the address it represents (which for arrays is the address of the 1st element of the array).

But function isn't called at compile time; it's called *at run time*.
At run time, (where) something should be placed on the stack to be considered as an argument.

Compiler cannot treat array references inside a function as labels and replace them with addresses, because it has no idea what actual array will be passed in at run time.

EY : 20171109 It can't anticipate the exact arguments that'll it be given *at run-time*; at the very least, my guess is, it's given instructions.

Bendersky [5] concludes by saying the difference between arrays and pointers does affect you. One way is how arrays can't be manipulated the way pointers can. Pointer arithmetic isn't allowed for arrays and assignment to an array of a pointer isn't allowed. cf. van der Linden (1994) [7]. Ch. 4, 9, 10.

Bendersky [5] has this one difference example, "actually a common C gotcha":

"Suppose one file contains a global array:"

```
char my_Arr[256]
```

Programmer wants to use it in another file, *mistakingly* declares as

```
extern char* my_arr;
```

When he tries to access some element of the array using this pointer, he'll most likely get a segmentation fault or a fatal exception (nomenclature is OS-dependent).

To understand why, Bendersky [5] gave this hint: look at the assembly listing

```
char a = array_place[7];

0041137E  mov  al,byte ptr [_array_place+7 (417007h)]
00411383  mov  byte ptr [a],al

char b = ptr_place[7];

00411386  mov  eax,dword ptr [_ptr_place (417064h)]
0041138B  mov  cl,byte ptr [eax+7]
0041138E  mov  byte ptr [b],cl
```

or my own, generated from `gdb` on Fedora 25 Workstation Linux:

```
0x00000000004004b1 <main+11>:    movzbl 0x200b8f(%rip),%eax        # 0x601047 <array_place+7>
0x00000000004004b8 <main+18>:    mov    %al , 0x1(%rbp)

0x00000000004004bb <main+21>:    mov    0x200be6(%rip),%rax        # 0x6010a8 <ptr_place>
0x00000000004004c2 <main+28>:    movzbl 0x7(%rax),%eax
0x00000000004004c6 <main+32>:    mov    %al , 0x2(%rbp)
```

"How will the element be accessed via the pointer? What's going to happen if it's not actually a pointer but an array?" Bendersky [5]

EY : 20171106. Instruction-level, the pointer has to

- `mov 0x200be6(%rip),%rax` - 1st., copy value of the pointer (which holds an address), into `%rax` register.
- `movzbl 0x7(%rax),%eax` - off that address in register ‘
- `mov %al,-0x2(%rbp)` - `mov` contents `-0x2(%rbp)`into register `%al`

If it's not actually a pointer, but an array, the value is copied into the `%rax` register is an actual `char` (or `float`, some type). *Not* an address that the registers may have been expecting!

5.1. **Structs in C.** From Shaw (2015) [4], Exercise 16,

`struct` in C is a collection of other data types (variables) that are stored in 1 block of memory. You can access each variable independently by name.

- The `struct` you make, i.e.g `struct Person` is now a *compound data type*, meaning you can refer to `struct Person` using the same kinds of expressions you would for other (data) types.
- This lets you pass the whole `struct` to other functions
- You can access individual members of `struct` by their names using `x->y` if dealing with a ptr.

If you didn't have `struct`, you'd have to figure out the size, packing, and location of memory of the contents. In C, you'd let it handle the memory structure and structuring of these compound data types, `struct`s. (Shaw (2015) [4])

Part 1. C, Stack, Heap Memory Management in C

6. C, Stack and Heap Memory Management, Heap and Stack Memory Allocation

cf. Ex. 17 of Shaw (2015) [4]

Consider chunk of RAM called stack, another chunk of RAM called heap. Difference between heap and stack depends on where you get the storage.

Heap is all the remaining memory on computer. Access it with `malloc` to get more.

Each time you call `malloc`, the OS uses internal functions (EY : 20171110 address bus or overall system bus?) to register that piece of memory to you, then returns ptr to it.

When done, use `free` to return it to OS so OS can use it for other programs. Failing to do so will cause program to *leak* memory. (EY: 20171110, meaning this memory is unavailable to the OS?)

Stack, on a special region of memory, stores temporary variables, which each function creates as locals to that function. How stack works is that each argument to function is *pushed* onto stack and then used inside the function. Stack is really

a stack data structure, LIFO (last in, first out). This also happens with all local variables in `main`, such as `char action`, `int id`. The advantage of using stack is when function exits, *C compiler* pops these variables off of stack to clean up.

Shaw's mantra: If you didn't get it from `malloc`, or a function that got it from `malloc`, then it's on the stack.

3 primary problems with stacks and heaps:

- If you get a memory block from `malloc`, and have that ptr on the stack, then when function exits, ptr will get popped off and lost.
- If you put too much data on the stack (like large structs and arrays), then you can cause a *stack overflow* and program will abort. Use the heap with `malloc`.
- If you take a ptr, to something on stack, and then pass or return it from your function, then the function receiving it will *segmentation fault*, because actual data will get popped off and disappear. You'll be pointing at dead space.

cf. Ex. 17 of Shaw (2015) [4]

References

[1] Bertrand Meyer. **Object-Oriented Software Construction** (Book/CD-ROM) (2nd Edition) 2nd Edition. Prentice Hall; 2 edition (April 13, 1997). ISBN-13: 978-0136291558

[2] Randall Hyde. **Write Great Code: Volume 1: Understanding the Machine** October 25, 2004. No Starch Press; 1st edition (October 25, 2004). ISBN-13: 978-1593270032

[3] Randall Hyde. **Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level**. 1st Edition. No Starch Press; 1 edition (March 18, 2006). ISBN-13: 978-1593270650

[4] Zed A. Shaw. **Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)** (Zed Shaw's Hard Way Series) 1st Edition. Addison-Wesley Professional; 1 edition (September 14, 2015) ISBN-13: 978-0321884923.

[5] Eli Bendersky. Are pointers and arrays equivalent in C?

[6] Brian W. Kernighan, Dennis M. Ritchie. **C Programming Language**, 2nd Ed. 1988.

[7] Peter van der Linden. **Expert C Programming: Deep C Secrets** 1st Edition. Prentice Hall; 1st edition (June 24, 1994) ISBN-13: 978-0131774292