Ingegneria degli Algoritmi (A.A. 2012-2013)

Corsi di Laurea in Ingegneria Informatica e Automatica, Ingegneria dei Sistemi Informatici, e Laurea Magistrale in Ingegneria Informatica

Sapienza Università di Roma

Appello del 12/07/2013 – Durata 60'

Cognome: _____ Nome: _____ Matricola: _____	Autorizzo la pubblicazione del voto di questo esame sul sito web http://www.dis.uniroma1.it/~demetres/didattica/asd, secondo quanto prevede il decreto legislativo 196/2003 (codice in materia di protezione dei dati personali) che dichiaro di conoscere. In fede, _____

Domanda 1 (10 punti su 32)

In riferimento al framework OpenCL, si discutano:

1. i concetti di: host, device, NDRange, work group e work item
2. il modello di memoria, illustrando:
   - i tipi di memoria presenti
   - la loro accessibilità da parte di host, device, NDRange, work group e work item
   - in che modo il programmatore può trasferire dati tra le diverse memorie

---

Domanda 2 (6 punti su 32)

Si consideri il seguente profilo di esecuzione collezionato da perf per un programma:

```
   9256.867184 task-clock                #    0.982 CPUs utilized
         2,782 context-switches          #    0.000 M/sec
             0 CPU-migrations            #    0.000 M/sec
       289,885 page-faults               #    0.031 M/sec
18,430,867,389 cycles                    #    1.991 GHz
17,315,784,258 stalled-cycles-frontend   #   93.95% frontend cycles idle
 6,280,843,367 stalled-cycles-backend    #   34.08% backend  cycles idle
 3,530,852,885 instructions              #    0.19  insns per cycle
                                         #    4.90  stalled cycles per insn
   570,441,397 branches                  #   61.624 M/sec
     2,297,608 branch-misses             #    0.40% of all branches
```

Si risponda alle seguenti domande, motivando le risposte:

1. La potenzialità della macchina sono sfruttate appieno?
2. Se ci sono colli di bottiglia prestazionali, dove vi aspettate che siano?

Domanda 3 (8 punti su 32)

Data una variabile `x` di tipo `int` e una variabile `p` di tipo `int*`, si considerino le seguenti espressioni C e si dica per ciascuna se è valida, se è un Lvalue o Rvalue, e di

che tipo è:

1. `*(&x+*p)+x`
2. `&p+x`
3. `*(&p-(int**)&x)+x`
4. `*(&x+*(p+7))`

Domanda 4 (8 punti su 32)

Si consideri il seguente frammento di programma C:

```
void swap(int* a, int* b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}

void pairwise_swap(int* v, unsigned long n) {
        unsigned long i;
        for (i=0; i<n; i+=2) swap(&v[i], &v[i+1]);
}
```

E la seguente traduzione in codice x86-64 effettuata da `gcc 4.7.2`:

```
swap:
        movl    (%rdi), %eax
        movl    (%rsi), %edx
        movl    %edx, (%rdi)
        movl    %eax, (%rsi)
        ret

pairwise_swap:
        xorl    %eax, %eax
        testq   %rsi, %rsi
        je      .L2
.L6:
        movl    (%rdi,%rax,4), %edx
        movl    4(%rdi,%rax,4), %ecx
        movl    %edx, 4(%rdi,%rax,4)
        movl    %ecx, (%rdi,%rax,4)
        addq    $2, %rax
        cmpq    %rax, %rsi
        ja      .L6
.L2:
        ret
```

Rispondere alle seguenti domande, motivando le risposte (si consulti la tabella allegata):

1. Quale rilevanti ottimizzazioni sono state effettuate da gcc nella traduzione da C a codice assembly?
2. Relativamente alla sola funzione `pairwise_swap`, a cosa corrispondono i registri usati dal codice x86-64 rispetto alle variabili del programma originale?

prefix	description	example	C analog
`add`	add source to destination	`addl $5,%ecx`	`ecx += 5`
`call`	procedure call	`call _foo`	`foo()`
`cltq`	sign-extend `eax` to `rax`	–	–
`dec`	decrement destination	`decq %rcx`	`rcx--`
`imul`	multiply destination with source	`imull %esi,%eax`	`eax *= esi`
`inc`	increment destination	`incl %ecx`	`ecx++`
`ja`	jump if above *(unsigned comparison)*	`cmpl %eax,%ebx` `ja L2`	`if ((unsigned)eax < (unsigned)ebx) goto L2`
`jae`	jump if above or equal *(unsigned comparison)*	`cmpl %eax,%ebx` `jae L2`	`if ((unsigned)eax <= (unsigned)ebx) goto L2`
`jb`	jump if below *(unsigned comparison)*	`cmpl %eax,%ebx` `jb L2`	`if ((unsigned)eax > (unsigned)ebx) goto L2`
`jbe`	jump if below or equal *(unsigned comparison)*	`cmpl %eax,%ebx` `jbe L2`	`if ((unsigned)eax >= (unsigned)ebx) goto L2`
`je`	jump if equal	`cmpq %rax,%rbx` `je L2`	`if (rax == rbx)` `goto L2`
`jg`	jump if greater *(signed comparison)*	`cmpq %rax,%rbx` `jg L2`	`if (rax < rbx)` `goto L2`
`jge`	jump if greater or equal *(signed comparison)*	`cmpq %rax,%rbx` `jge L2`	`if (rax <= rbx)` `goto L2`
`jl`	jump if less *(signed comparison)*	`cmpq %rax,%rbx` `jl L2`	`if (rax > rbx)` `goto L2`
`jle`	jump if less or equal *(signed comparison)*	`cmpq %rax,%rbx` `jle L2`	`if (rax <= rbx)` `goto L2`
`jmp`	unconditional jump	`jmp L2`	`goto L2`
`jne`	jump if not equal	`cmpq %rax,%rbx` `jne L2`	`if (rax != rbx)` `goto L2`
`jnz`	identical to `jne`	–	–
`jz`	identical to `je`	–	–
`lea`	copy address to destination (load effective address)	`leaq -12(%rax),%rcx`	`rcx=rax-12`
`leave`	pop the current stack frame, and restore the caller's frame	–	–
`mov`	copy data from source to destination	`movq $7,(%rax)`	`*(long*)rax=7`
`movabs`	copy 64-bit immediate to destination register	`movabsq $-7,%rax`	`rax=-7`
`movsl`	copy sign-extended word to destination register	`movslq %bx,%rax`	`rax=bx`
`movzl`	copy zero-extended word to destination register	`movzlw %bx,%eax`	`eax=(unsigned)bx`
`movsb`	copy sign-extended byte to destination register	`movsbq %bl,%rax`	`rax=bl`
`movzb`	copy zero-extended byte to destination register	`movzbq %bl,%rax`	`rax=(unsigned)bl`
`pop`	pop value from stack and write it to destination	`popq %rbx`	–
`push`	push value on stack	`pushq %rbx`	–
`ret`	return from procedure call	`ret`	`return`
`sub`	subtract source from destination	`subl $5,%ecx`	`ecx -= 5`

Tabella 1: Istruzioni x86-64 più comunemente utilizzate.