嵌入式重难点总结

软中断流程

1. 通过`__swi(中断号) function`给中断函数指定中断号（一般在C代码中撰写）
2. 调用function时，相当于执行了`SWI 中断号`指令
3. 再执行`SWI`之前需要先注册中断程序，即把处理软中断的程序（SWI_handler）地址放到`0x08`软中断向量表地址上
4. 注册之后执行`SWI`的时候，会执行`0x08`中断向量地址上存放的跳转指令（`B SWI_handler`），即跳转到SWI_handler；同时将function的参数传递到寄存器内（如果参数不超过4个），超过4个应该存放到内存

```
// 中断注册程序
unsigned Install_Handler (unsigned *handlerloc, unsigned *vector)
{
    unsigned vec, oldvec;
    vec = ((unsigned)handlerloc - (unsigned)vector - 0x8)>>2;    // 确保中断处理函数地址偏移在26位以内：正负32M
    if ((vec & 0xFF000000) != 0)     // 取低24位，即中断服务程序地址
     { return 0;}
    vec = 0xEa000000 | vec;          // 0xEa 应该时跳转指令B的操作码，这里表示跳转指令：”B handlerloc“
    oldvec = *vector;
    *vector = vec;                   // 将中断向量表中断向量内写入跳转指令的编码，即vec
    return (oldvec);
}
```

5. 跳转后，`SWI_handler`需要干五件事：
   - 1）将保存函数参数的寄存器以及lr寄存器存放到堆栈里保存，
   - 2）通过`BIC`指令获取中断号放置在r0寄存器，
   - 3）将堆栈指针保存到r1，
   - 4）跳转到要执行function的处理函数中（C_SWI_handle）加密，并将r0，r1寄存器作为函数参数传递
   - 5）处理完中断函数后恢复现场

```
SWI_Handler
        STMFD       sp!,{r0-r12,lr}       ;保存现场
        LDR         r0,[lr,#-4]           ;获取 SWI 指令
        BIC         r0,r0,#0xff000000     ;参数1，NUM
        MOV         R1, SP                ;参数2，传递堆栈指针
        BL C_SWI_Handler                  ;To Function
        LDMFD       sp!, {r0-r12,pc}^     ;处理完中断恢复现场，将最初的lr->pc，继续执行SWI指令的下一条指令
        END
```

6. `C_SWI_handle`接受中断号和堆栈指针，通过堆栈指针开始从堆栈中获取参数数据，根据中断号选择计算的程序逻辑，计算完成将结果再次写入堆栈

```
void C_SWI_handler (int swi_num, int *reg )
{   switch (swi_num)
    {
        case 0 :
            ……            /* SWI number 0 code */
                    break;
            case 1 :
            ……            /* SWI number 1 code */
                    break;
        ……
            default :
            break;       /* Unknown SWI - report error */
    }
    return;
}
```

或

```
C_SWI_Handler
        STMFD   sp!,{r0-r12,lr}
        CMP     r0,#MaxSWI          ; Range check
        LDRLE   pc, [pc,r0,LSL #2]  ;(PC -> DCD SWInum0)
        B       SWIOutOfRange
SWIJumpTable
        DCD    SWInum0
        DCD    SWInum1
SWInum0   ; SWI number 0 code
        B    EndofSWI
SWInum1   ; SWI number 1 code
        B    EndofSW
EndofSW
        SUB     lr, lr, #4
        LDMFD   sp!, {r0-r12,pc}^
        END
```

7. 最后从堆栈中取出计算结果放到寄存器r0中
8. 中断结束，从堆栈中恢复现场，继续执行`SWI`指令后的指令

系统启动流程

以`S3C2410A`处理器的启动文件startup.s为例.

1. 系统上电后，先进行复位Reset
2. 设置PC的初始值0，执行Reset_Handler
3. Reset_Handler初始化需要干以下几件事
    - 1）设置WT_Setup（Watchdog Timer）、CLK_Setup（Clock）、MC_Setup（Memory Controller）、PIO_Setup（IO port）、Stack_Setup（Stack/Heap）
    - 2）设置异常中断的模式的栈指针，包括Undefined、Abort、FIQ、IRQ、Supervisor、User
    - 3）设置SRAM

```
                BL INISDRAM
```

    - 4）堆栈空间的设置
    - 5）进入C程序代码(__main -> init -> main)

```
IMPORT  __main
                LDR     R0, =__main
                BX      R0
```

- 如果调试时无法进入main，可能的原因是什么?
  - main单词拼写错误
  - 启动代码不对
  - 没有将main.c文件添加到工程文件中
- Startup.s文件的功能和作用?
  - 设置中断向量与终端服务程序地址
  - 分配堆栈空间
  - 设置时钟，看门狗Timer，内存控制，IO端口
  - 设置中断入口IRQ_Entry
  - 实现Reset_Handler

OpenMP多核处理

- #pragma
  - parallel：表示这段代码将被并行执行；
  - for：表示将循环计算任务分配到多个线程中并行执行；
  - sections：用于实现多个结构块语句的任务分担；
  - parallel sections：类似于parallel for；
  - single：表示一段只被单个线程执行的代码；
  - critical：保证每次只有一个OpenMP线程进入；
  - flush：保证各个OpenMP线程的数据映像的一致性；
  - barrier：用于并行域内代码的线程同步，线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行；
  - atomic：用于指定一个数据操作需要原子性地完成；
  - master：用于指定一段代码由主线程执行；
  - Thread private：用于指定一个或多个变量是线程专用。
- 基本使用
  - `#pragma omp parallel for` ☆☆☆☆☆

```
int step = 100;
int _tmain(int argc, _TCHAR* argv[])
{
   int i;                          // 注意int i 不要再for循环里面写int i = 0
   #pragma omp parallel for
   for (i = 0; i< step; i++)
   {
       printf("i = %d\n", i);
   }
   return 0;
}
```

- 为了防止共享数据导致并行抢占改写，需要使用:
  - `#pragma omp parallel for private(共享变量)` ☆☆☆

```
ifirst = 10;
int k[600];
for(j = 0; j <= 60; j++)
{
#pragma omp parallel for private(i2)
    for(i=0;i<6000000;i++){
     i2 = i-(i/600)*600;
     k[i2] = ifirst + i;
    }
}
```

- `share`共享不能用于类似sum求和的方式

```
float sum = 0.0;
float a[10000],b[10000];
for(int j=0;j<10;j++)
{
    sum=0.0;
    #pragma omp parallel for shared(sum)
   for(int i=0; i<10000; i++) { // 注意int不要写在for循环，”int i;“在for外面声明
     sum += a[i] * b[i];
   }
    printf(“j=%d, sum=%f\n”,j,sum);
}
  return sum;
```

- `critical`每次让一个线程去操作，即让下一行代码串行

```
float sum = 0.0;
flaot a[10000],b[10000];
for(int j=0;j<10;j++)
{
    sum=0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
    #pragma omp critical
        sum += a[i] * b[i];
    }
    printf(“j=%d, sum=%f\n”,j,sum);
}
return sum;
```

- `reduction`让所有线程在最后统一执行某一操作，例如：求和”+：“ ☆

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

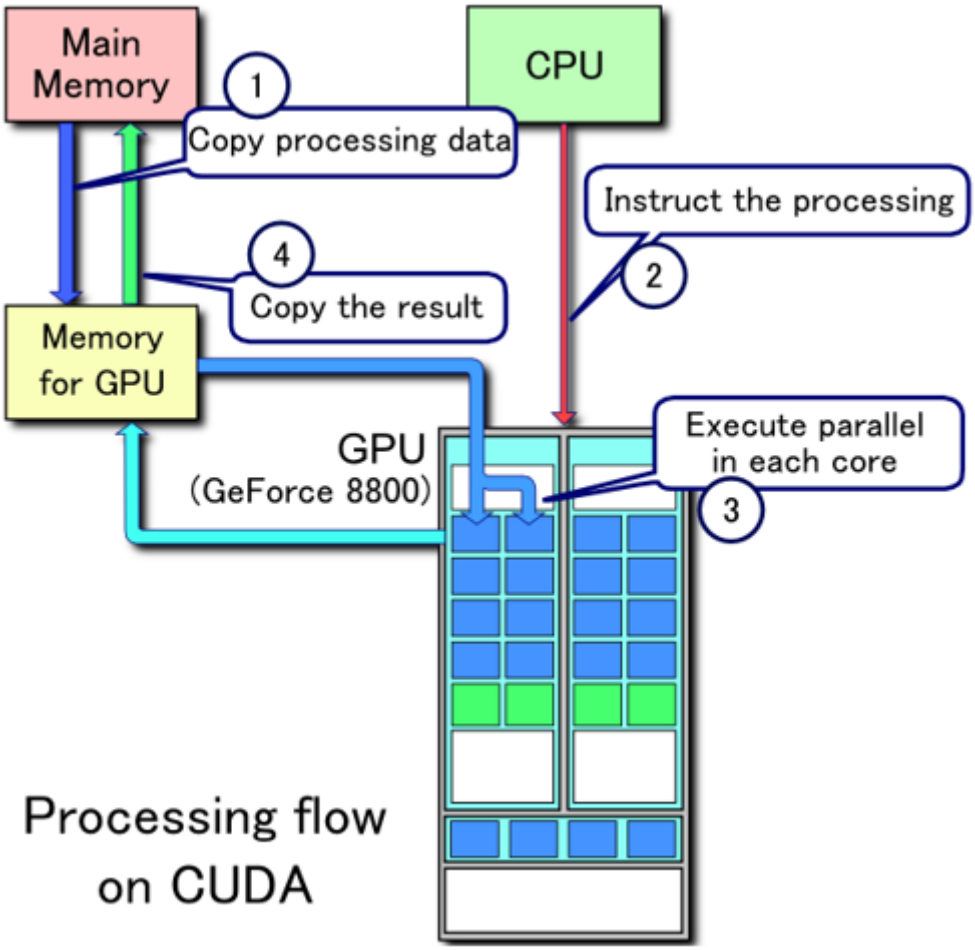
- `barrier`用于并行域内代码的线程同步，线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行

```
#pragma omp parallel private(myid,istart,iend)
myrange(myid,istart,iend);
for(i=istart; i<=iend; i++){
   a[i] = a[i] - b[i];
}
#pragma omp barrier
myval[myid] = a[istart] + a[0]
```

- `schedule`分配线程执行
  - `static`：线程按顺序分配，线程内数据处理顺序也按照for循环顺序分配
  - `dynamic`：线程随机分配，线程内数据处理顺序也按照for循环顺序分配
  - `guided`：线程和线程内数据处理顺序都是随机分配

CUDA

- CUDA处理数据过程
  - Copy data from main mem to GPU mem.
  - CPU instructs the process to GPU.
  - GPU execute parallel in each core.
  - Copy the result from GPU mem to main mem.



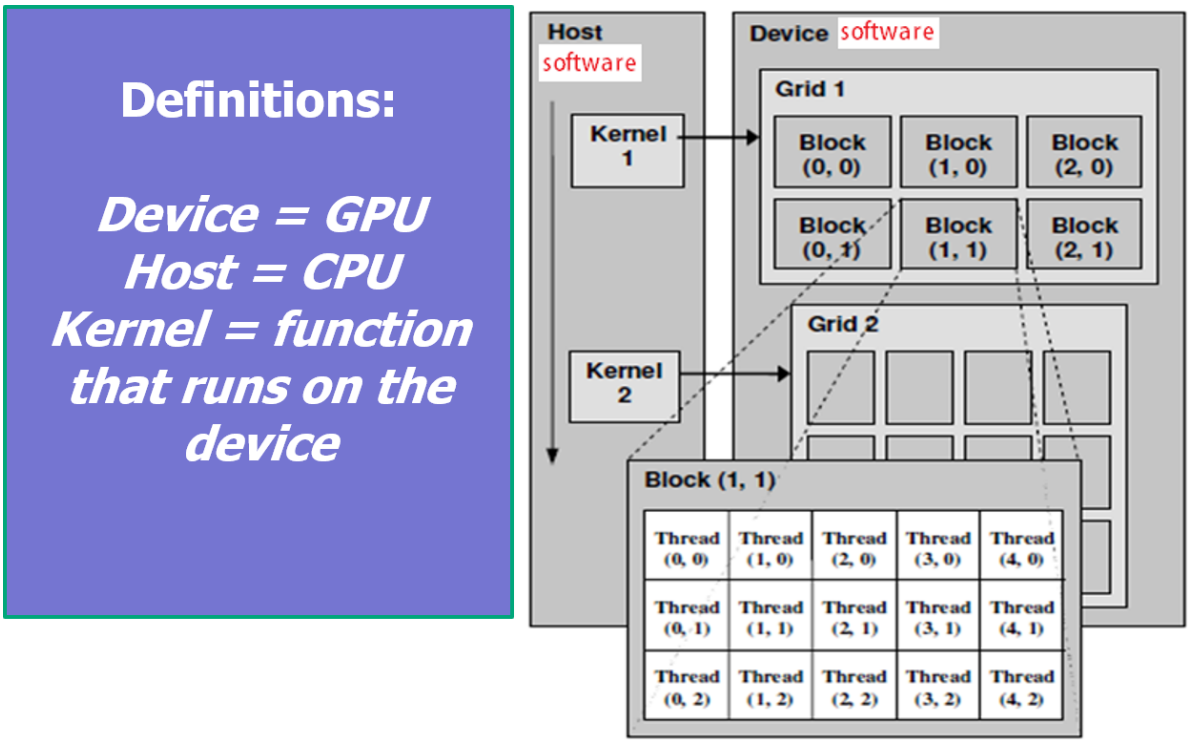
- CUDA编程基础代码
  - `<<<blk,thr>>>`中，`blk`表示需要使用的block数目，`thr`表示每个使用的block用多少个线程使用

```
//hello_world.cu:
#include <stdio.h>

__global__ void hello_world_kernel()     // CUDA执行的核（kernel）函数需要加__global__
{
    printf(“Hello World\n”);
}

int main()
{
     hello_world_kernel<<<1,1>>>();      // 函数调用需要指定<<<Blk, Tid>>>
}
```

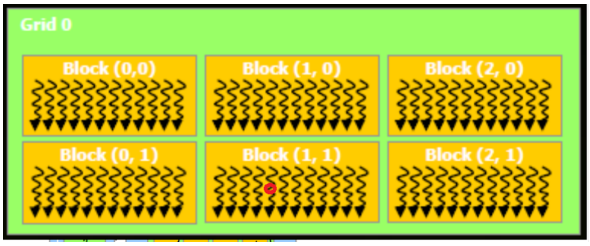
- 基本概念




```
BlockID= blockIdx.y*gridDim.x+blockIdx.x

ThreadID=BlockID*blockDim.x*blockDim.y*blockDim.z
    +threadIdx.z*blockdIm.x*blockDim.y*
    +threadIdx.y*blockDim.x
    +threadIdx.x
```

- 例子



dim3 grid(3,2)；dim3 block(12,1,1)；
gridDim.x=3; gridDim.y=2;
blockDim.x=12; blockDim.y=1; blockDim.z=1;
blockIdx.x=1; blockIdx.y=1;
threadIdx.x=4, threadIdx.y=0; threadIdx.z=0;
threadID=(1*3+1)*12+4=52

- 一维CUDA模板代码

```
// Kernel definition
```

```
__global__ void VecAdd(float* A, float* B, float* C)    // 注意数组参数用指针形式传递
{
    int i = threadIdx.x;    // 线程只需要用x维度
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);  // 使用一个block和N的线程
    ...
}
```

- 二维CUDA一个Block模板代码

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) // 注意二维数组的参数
{
    int i = threadIdx.x;
    int j = threadIdx.y;    // 需要指定线程的两个维度
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;              // 使用的Block数目
    dim3 threadsPerBlock(N, N);     // 注意声明dim3类型，每个block采用N*N个线程
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

- 二维CUDA多个Block模板代码 ☆☆☆☆☆

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;  // 采用blockIDx * blockDim + threadIDx 的方式赋值变量
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);       // 数据类型都采用dim3，每个Block使用后16*16个线程
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);   // 计算每个维度需要多少个Block，返回Block数目
```

```
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

- Host Call Device