Relatório EP2 MAC0219

João Gabriel Basi Nº USP 9793801
Juliano Garcia de Oliveira Nº USP: 9277086

Neste exercício programa, nossa tarefa foi implementar uma operação de redução em GPU (Nvidia), utilizando CUDA, e utilizamos a linguagem C++. Como cada matriz possui 9 elementos (é uma matriz 3×3), representamos as N matrizes em um único vetor, de tamanho $9 * N$, para minimizar a quantidade de alocação de memória, tanto no *Host* quanto no *Device* (através da função *cudaMemcpy()*). A matriz i está armazenada na posição $9 \cdot i$ deste vetor.

A nossa implementação do *Kernel* CUDA para fazer a redução de S com respeito a $\oplus$ utiliza algumas técnicas que vimos em aula. O código do nosso *Kernel* é o seguinte:

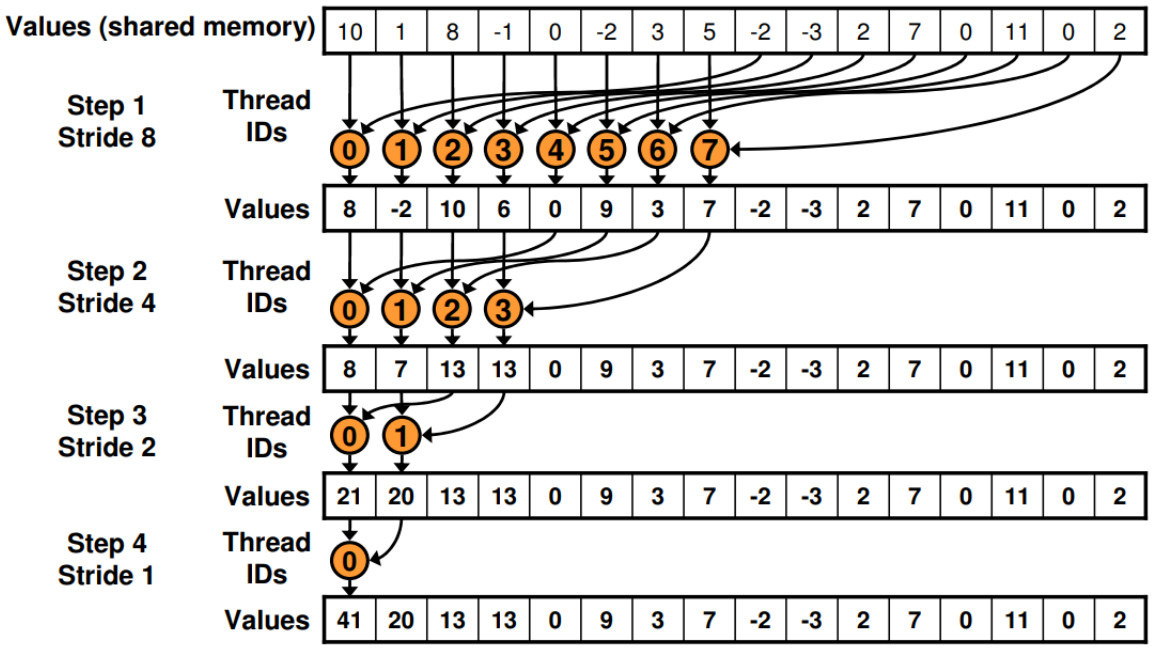
```
__global__ void reduce_min( int32_t *list_m, int32_t N ) {
    extern __shared__ int32_t cache[];
    int tid = 9*(threadIdx.x + blockIdx.x * blockDim.x);
    int cid = 9*threadIdx.x;

    for (int32_t i = 0; i < 9; i++)
        cache[cid + i] = list_m[tid + i];
    __syncthreads();

    for (int32_t i = blockDim.x/2; i != 0; i >>= 1) {
        if (threadIdx.x < i)
            for (int32_t j = 0; j < 9; j++)
                cache[cid + j] = min(cache[cid + 9*i + j], cache[cid + j]);
        __syncthreads();
    }

    if (cid == 0)
        for (int32_t i = 0; i < 9; i++)
            list_m[9*blockIdx.x + i] = cache[i];

}
```

Para melhorar o desempenho, cada bloco copia o conteúdo pelo qual é responsável para a memória compartilhada daquele bloco (no nosso caso, o *array cache*). Assim, cada *thread* não precisa fazer acessos constantes à memória global, apenas no final da redução. Cada bloco faz a redução apenas na memória compartilhada daquele bloco. Utilizamos a técnica de *Sequential Addressing* para fazer a redução em paralelo.

Com esta técnica, não há conflito na escrita da memória, e mantemos a corretude do algoritmo de redução. Cada *thread* reduz o valor pelo qual é responsável com o valor correspondente, e armazena na posição correta (no nosso *Kernel*, o *array* da *thread* começa em *cache[cid]*, e *cid* é uma variável específica de cada *thread*).

Em cada iteração (também chamada de *stride*), a quantidade de *threads* que irão fazer a redução é dividida por 2. As *threads* devem ser sincronizadas ao final de cada iteração para manter a corretude do algoritmo (i.e., uma redução poderia ainda não ter acontecido, mas uma *thread* da próxima iteração reduzir outros elementos e sobrescrever o valor que seria calculado por outra *thread* na iteração anterior, por exemplo). Ao final, apenas a primeira *thread* do bloco, que possui o resultado da redução correto, vai escrever na memória global.

Abaixo há um diagrama ilustrando como funciona o *Sequential Addressing*. No nosso caso, cada *thread* é responsável por 9 valores (cada entrada da matriz), não apenas por 1, como mostrado na figura.



Como o kernel só faz a redução nas matrizes contidas no mesmo bloco, se temos M blocos, então teremos M matrizes reduzidas, por isso, ao final do *kernel*, executamos na CPU uma redução sequencial dessas M matrizes restantes, para então conseguir o resultado final do programa.

OBS: As placas da NVIDIA possuem uma instrução de hardware para obter o mínimo entre dois números. O CUDA dá *override* nessa instrução, então quando utilizamos a função *min()* dentro do *Kernel* CUDA, essa instrução é chamada na GPU, ou seja, não há necessidade nenhuma de calcular o mínimo fazendo alguma conta com os números, visto que utilizando o *min()* da GPU não haverá *branch divergence*. Para saber mais, pesquise sobre a função $\_\_nv\_min()$.

Viabilidade

Utilizando o comando *nvprof* para cronometrar o tempo do nosso programa, podemos ver que o tempo de copiar o conteúdo para a memória da GPU chega a ser 15 vezes maior que o tempo de execução da função de redução, com isso podemos concluir que não é viável reduzir S com respeito à $\oplus$ na CPU utilizada, já que qualquer ganho a mais de performance em software não será capaz de reduzir o tempo que a informação leva para ser copiada.

References

[Mark Harris] "Optimizing Parallel Reduction in CUDA" https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf