GPU Computing: Übung #6

Abgabe am Donnerstag, 11. Dezember 2014

Günther Schindler, Alexander Schnapp, Klaus Naumann

Inhaltsverzeichnis

Reading:Roofline - An Insightful Visual Performance Model for Multicore Architectures

In this paper the author introduces a visual computational model for multicore Architecutures called 'roofline' model. It relates the operational intensity (mean operations per byte of DRAM traffic) with an upper bound for performance of a kernel (Attainable GFlops/s). For that 'roofline' is uses the minimum of Peak Floating-Point Performance (constant) and the Peak memory Bandwith multiplied with the operational intensity (line with postiv slope), so wether the problem is compute-bound or memory-bound.
In that way it tells the user what optimizations should he implement and in what order, by pointing out the limiting factor of perfomance. Using micro benchmarks he is testing 4 different cournels to apply this model. The Roofline model seams to be a very clear and easy to aply way for characterizing multicore architures for to find a first starting point for optimizations.

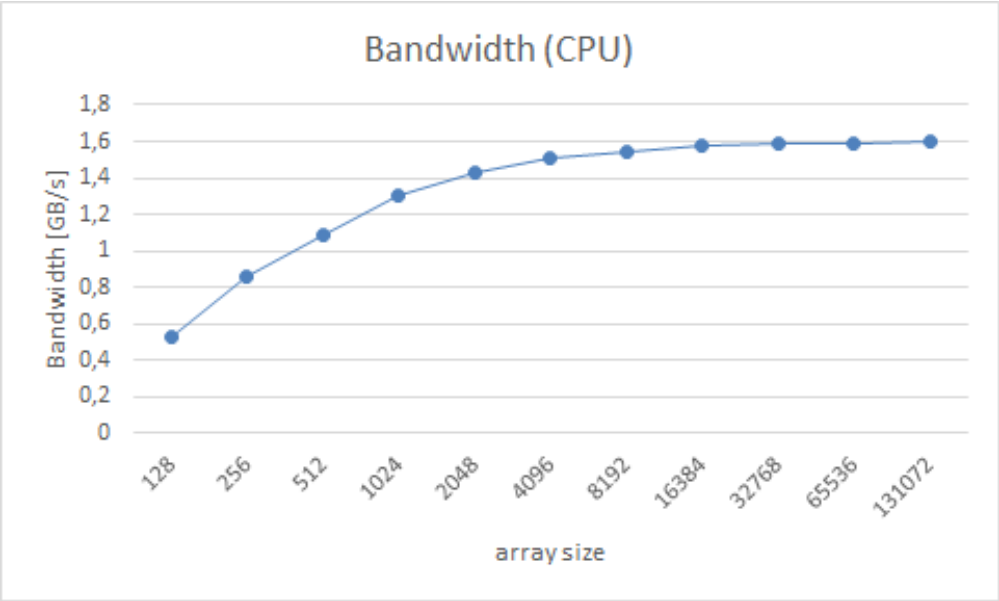

Abbildung 1: Bandbreite in Abhängigkeit von der Arraygröße für die unoptimierte sequentielle CPU-Version der Global Sum Reduction.

Reduction - CPU

In diesem Abschnitte sollte eine *global sum reduction* als sequentielle CPU-Version realisiert werden.
In Abbildung 1 sind die Messergebnisse der sequentiellen Implementierung dargestellt. Hier werden alle Elemente des Datentyps *float* eines *arrays* mit Hilfe einer *for-loop* addiert; als Ergebnis erhalten wir die Summe aller Elemente.

Was man in Abbildung 1 beobachten kann ist, dass bei einer Größe von 128 Elementen eine Bandbreite von ca. 0,5 *GB/s* gemessen wurde. Mit zunehmender Arraygröße steigt diese bis auf ca. 1,5 *GB/s* an und pendelt sich in diesem Bereich ein. Ab diesem Bereich ist der Cache nicht mehr ausreichend und der Prozess muss auf den Arbeitsspeicher ausweichen, was sich auf die Geschwindigkeit der Berechnung auswirkt.

Reduction - GPU initial version

In dieser Aufgabe war das Ziel eine unoptimierte parallele GPU-Version der *global sum reduction* zu implementieren und die Bandbreite in Abhängigkeit der Arraygröße zu messen und diese mit der Bandbreite der CPU-Version zu vergleichen. Hier der Code für den unoptimierten Kernel:

```
__global__ void
reduction_KernelNaive(int numElements, float* dataIn, float* dataOut)
{
    extern __shared__ float sPartArray[];

    const int tid = threadIdx.x;
     unsigned int elementId = blockIdx.x * blockDim.x + threadIdx.x;

    sPartArray[tid] = dataIn[elementId];
    __syncthreads();

    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        if(tid % (2 * s) == 0) {
            sPartArray[tid] += sPartArray[tid + s];
        }
        __syncthreads();
    }

     if (tid == 0) {
        dataOut[blockIdx.x] = sPartArray[0];
     }
}
```

Betrachtet man Abbildung 2 stellt man fest, dass die sequentielle CPU-Version bis zu einer Arraygröße von knapp über 8192 Elementen eine höhere Bandbreite aufweist, als die unoptimierte parallele GPU-Version. Der Grund hierfür sind der Overhead und die geringe Datenmenge der parallelen Implementierung, welche für kleine Arraygrößen eine negative Auswirkung auf die Bandbreite aufweisen. Mit steigenden Problemgrößen steigt auch die Bandbreite der parallelen Implementierung und kann somit zu weitaus schnelleren Ausführungszeiten führen. Dies wirkt sich dementsprechend auch auf den Speed-Up aus. Aus Gründen der Darstellung wurde hier die Darstellung von Abbildung 2 im oberen Bereich begrenzt. Als Peak wurde hier eine Bandbreite von ca. 38 *GB/s* für 262144 Elemente erreicht.

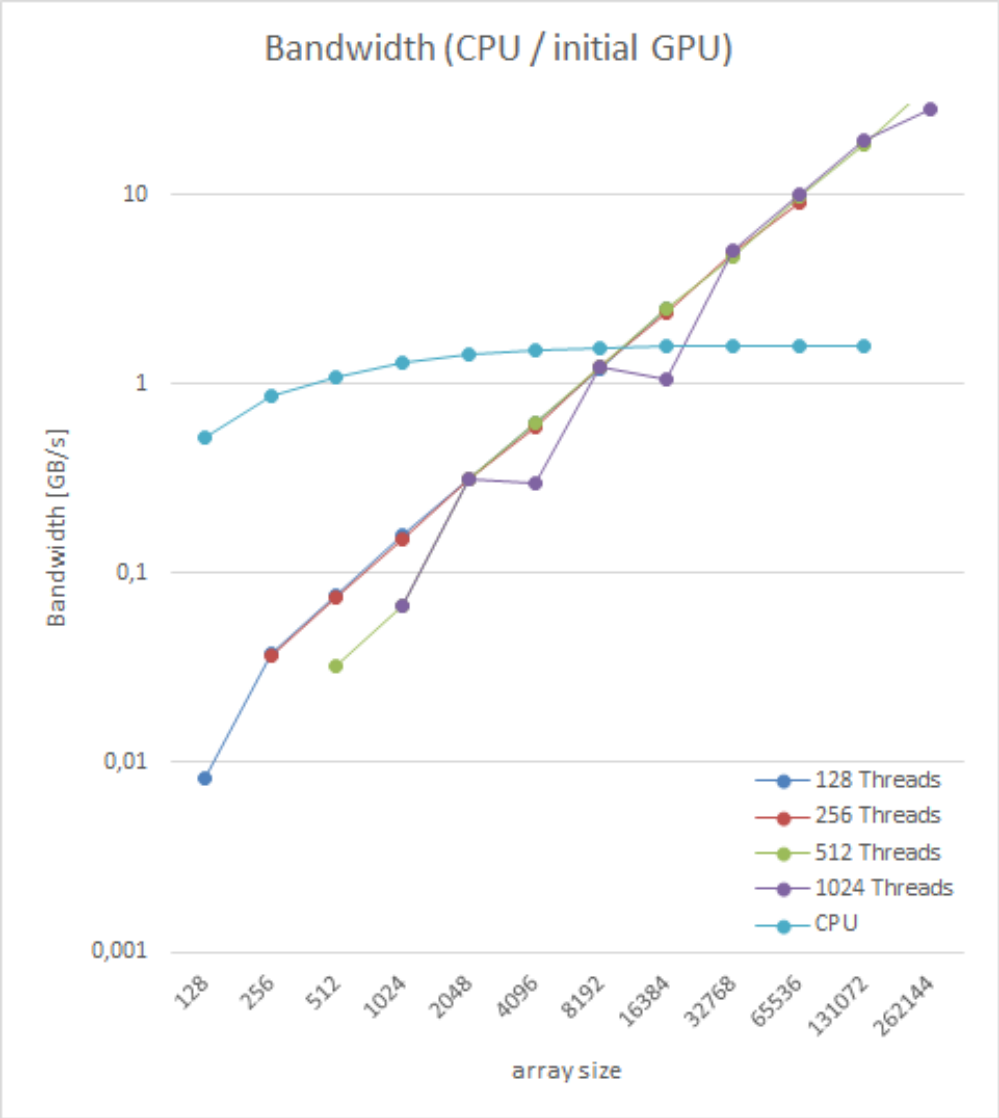

Abbildung 2: Bandbreite in Abhängigkeit von der Arraygröße für die unoptimierte parallele GPU-Version der Global Sum Reduction. Die Messungen wurden mit unterschiedlicher Anzahl von Threads durchgeführt.

Reduction - GPU optimized version

In der letzten Aufgabe dieser Übung sollte eine optimierte parallele GPU-Version der *global sum reduction* implementiert werden und die Bandbreiten aller drei Versionen verglichen werden. Für den optimierten Kernel wurde die *interleaved addressing non-divergent* Variante verwendet. Hier der Kernel-Code:

```
__global__ void
reduction_KernelOptimized(int numElements, float* dataIn, float* dataOut)
{
    extern __shared__ float sPartArray[];

    const int tid = threadIdx.x;
     unsigned int elementId = blockIdx.x * blockDim.x + threadIdx.x;

    sPartArray[tid] = dataIn[elementId];
    __syncthreads();

    for(unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
        if(tid < s) {
            sPartArray[tid] += sPartArray[tid + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        dataOut[blockIdx.x] = sPartArray[0];
    }
}
```

Auf den ersten Blick kann man feststellen, dass die optimierte Version der parallelen Implementierung den *break-even-point* früher - bei einer Arraygröße von ca. 3000 Elementen - erreicht als die unoptimierte Version. Dies liegt daran, dass die *Threads* eines *Warps* nun einem vorbestimmten Kontrollpfad folgen und somit keine Divergenz auftritt. Dies wird durch die *if-Abfrage* if(tid ¡ s) gewährleistet; wobei tid der *ThreadIdx.x* also der Thread-Index ist und s einen Stride darstellt. Als Maximum konnten wir hier eine Bandbreite von ca. 73 *GB/s* erreichen.

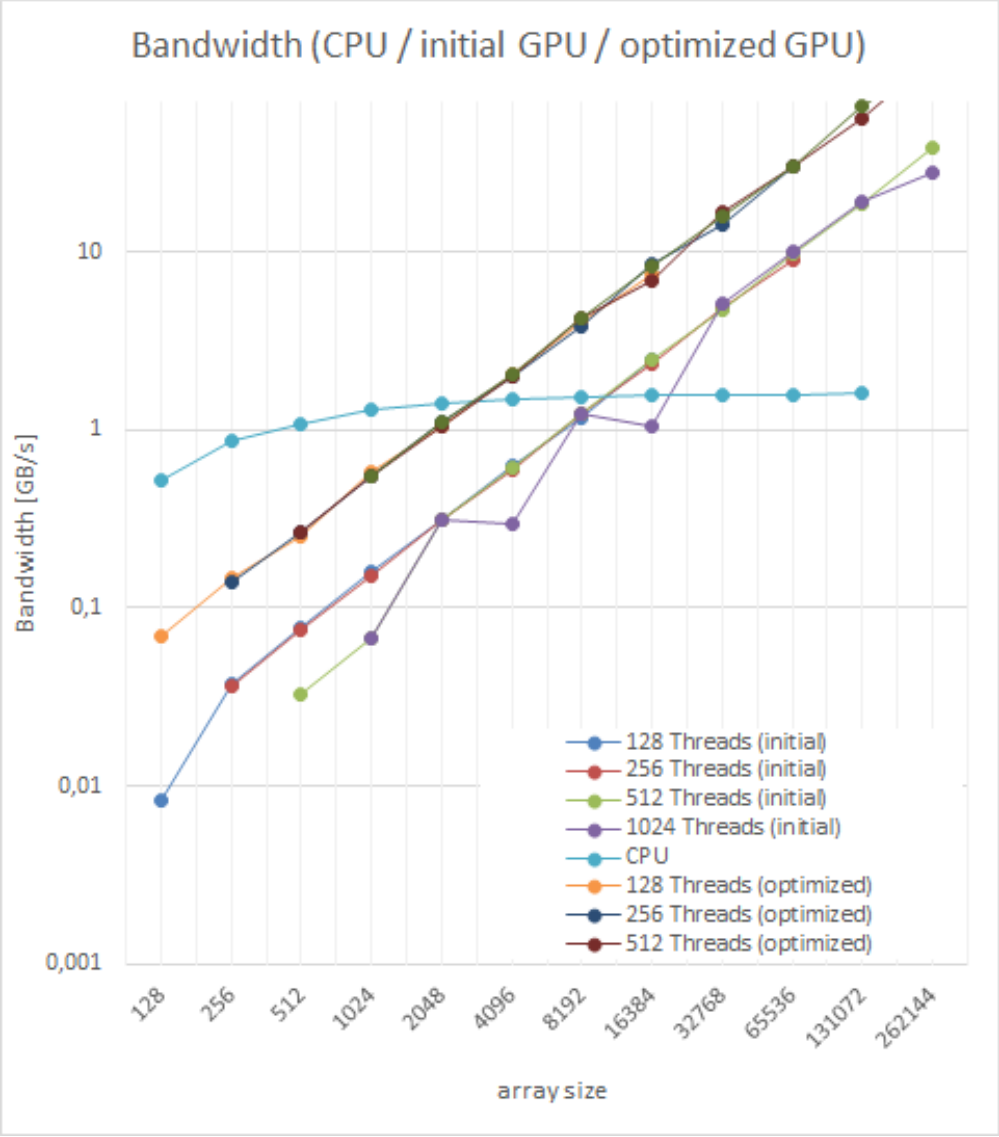

Abbildung 3: Bandbreite in Abhängigkeit von der Arraygröße für die unoptimierte als auch die optimierte parallele GPU-Version der Global Sum Reduction. Die Messungen wurden mit unterschiedlicher Anzahl von Threads durchgeführt.