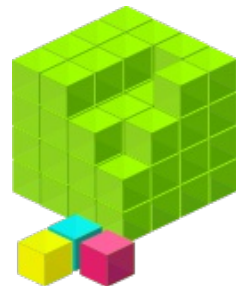
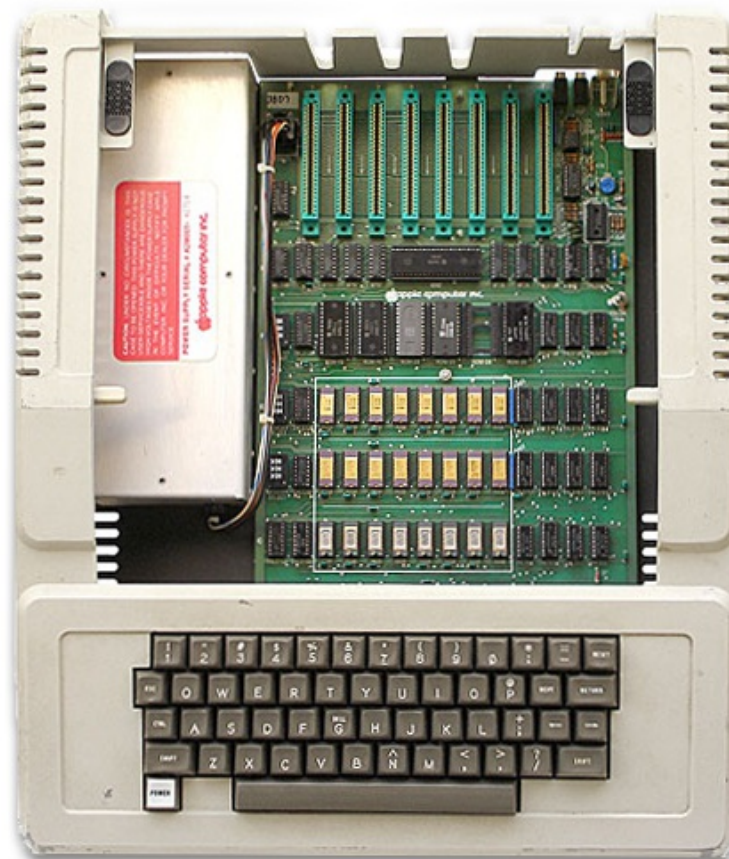


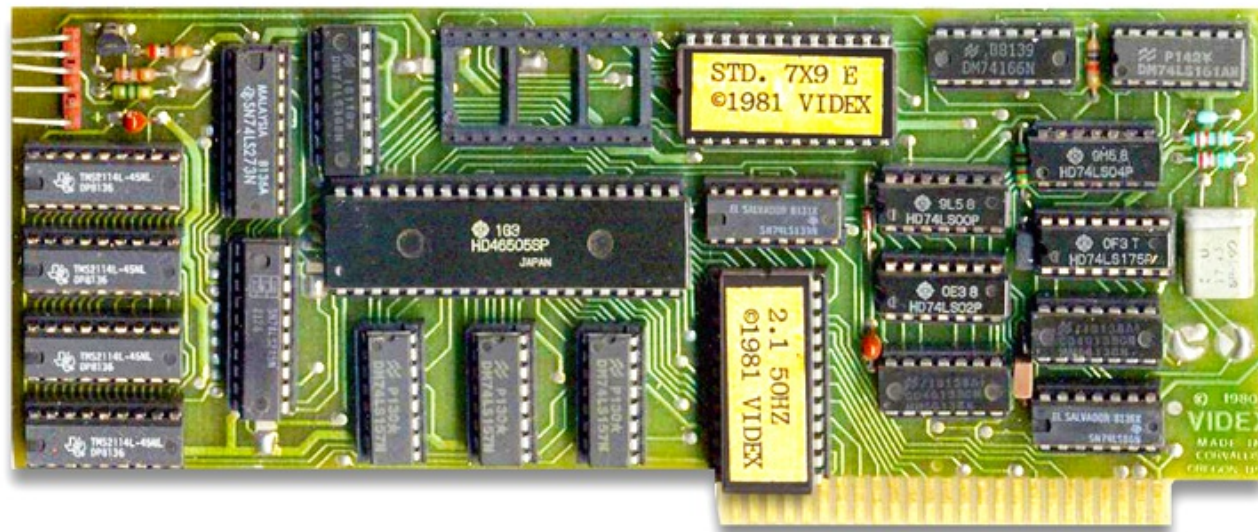
GPU Programmierung mit CUDA

Matthias Endler





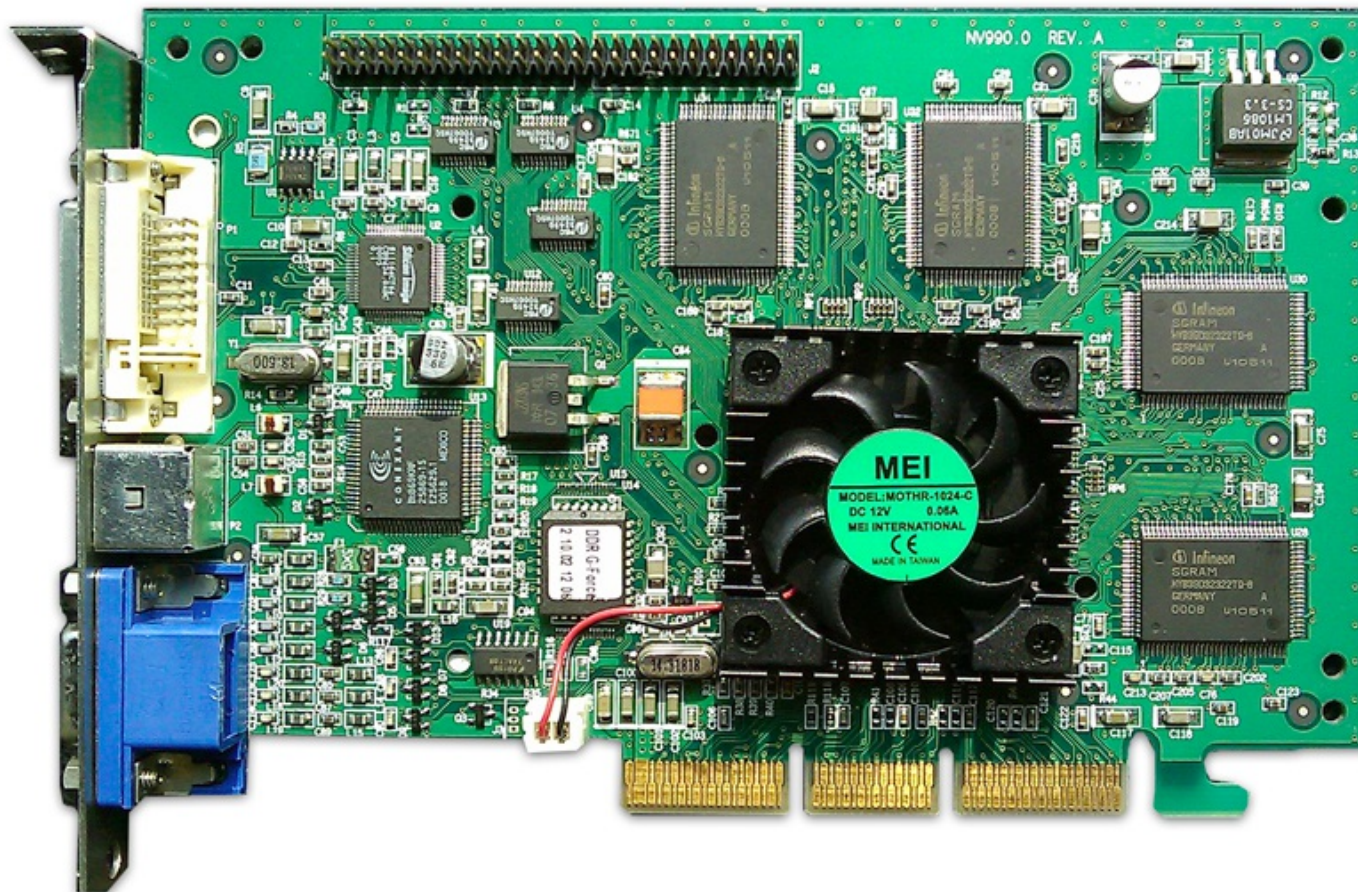
Apple II, 1977



Videx Videoterm, 1981

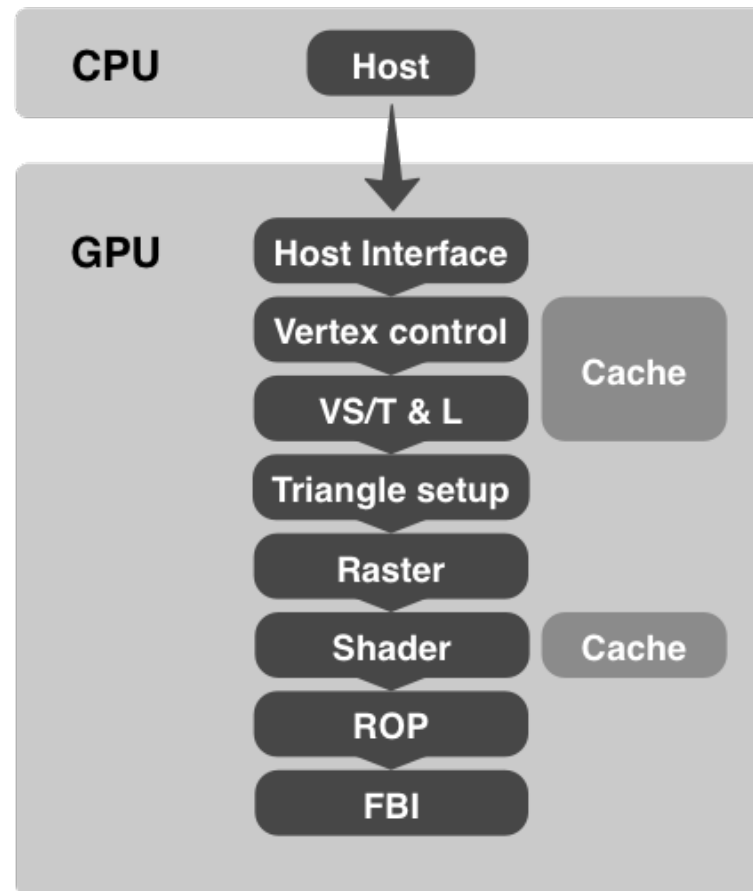


Videx Videoterm, 1981

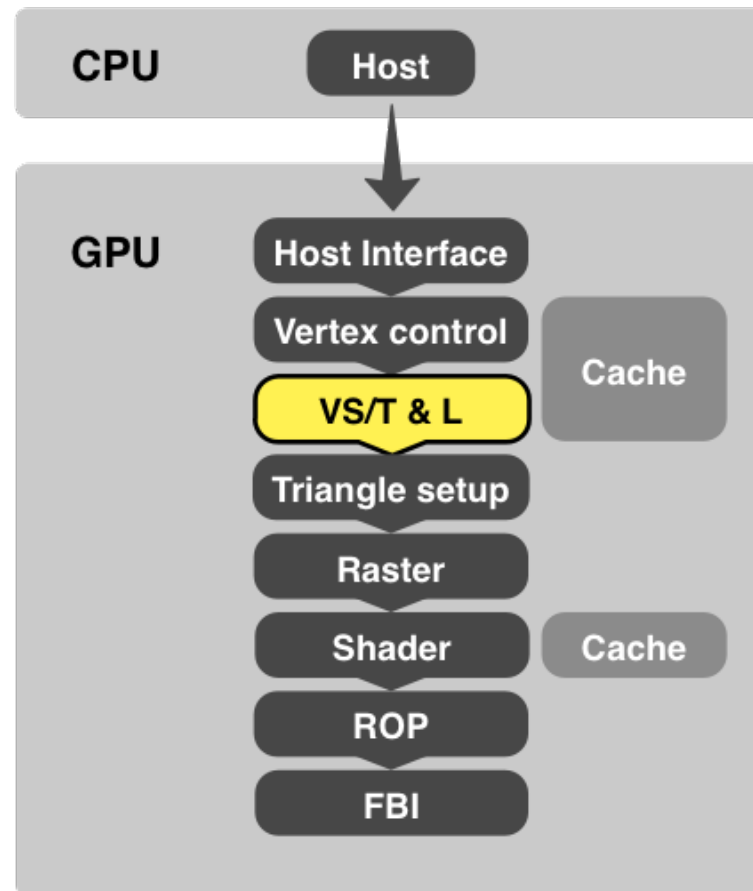


VisionTek Geforce 256, 1999

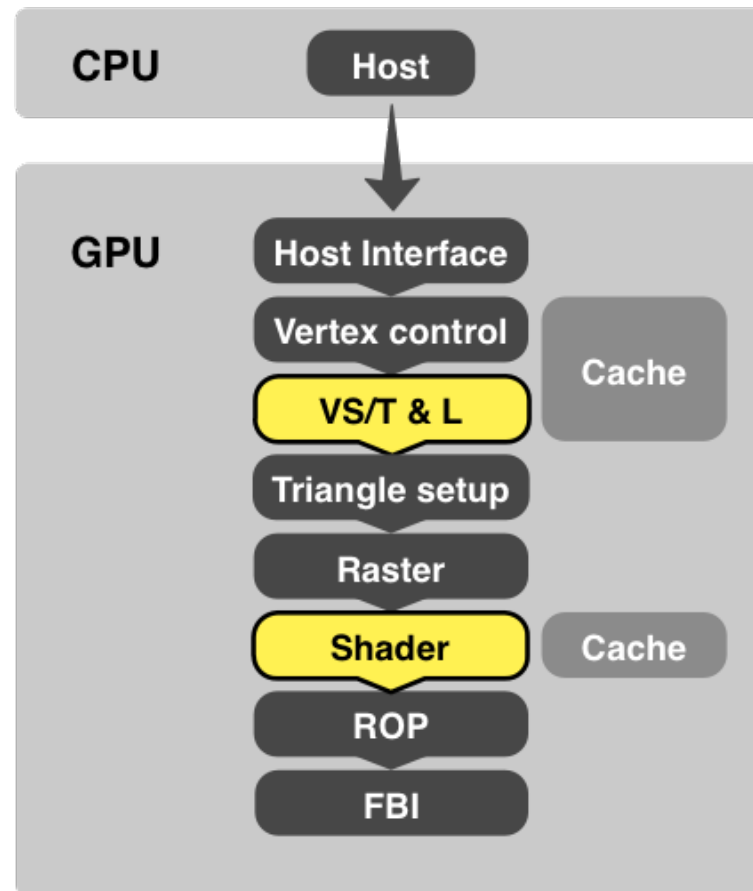
Fixed-function pipeline



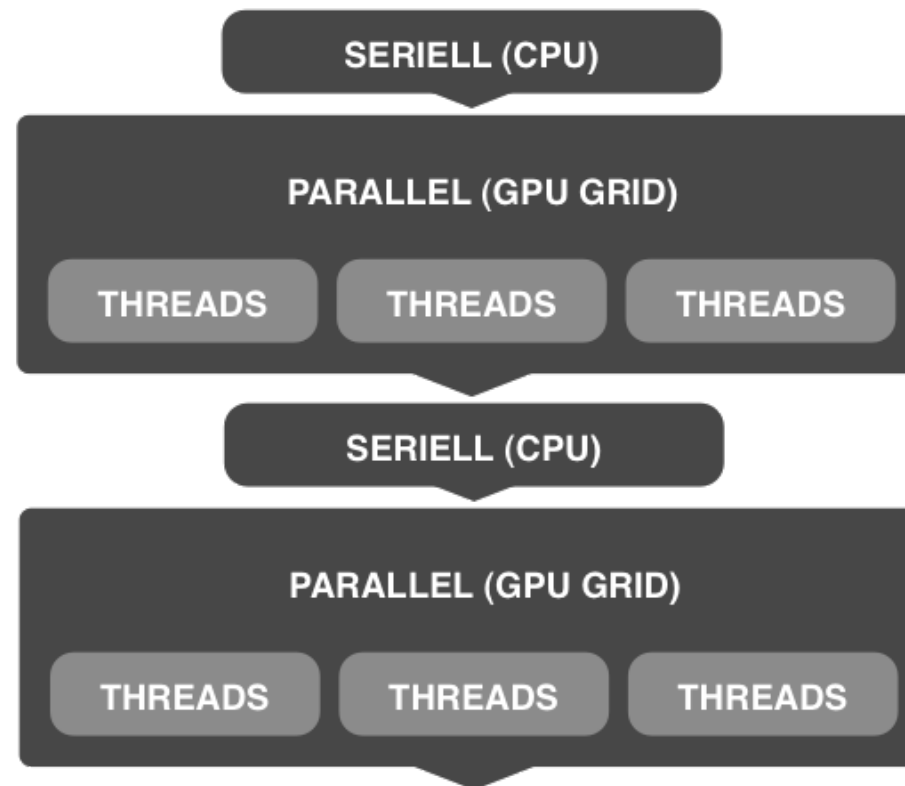
Fixed-function pipeline



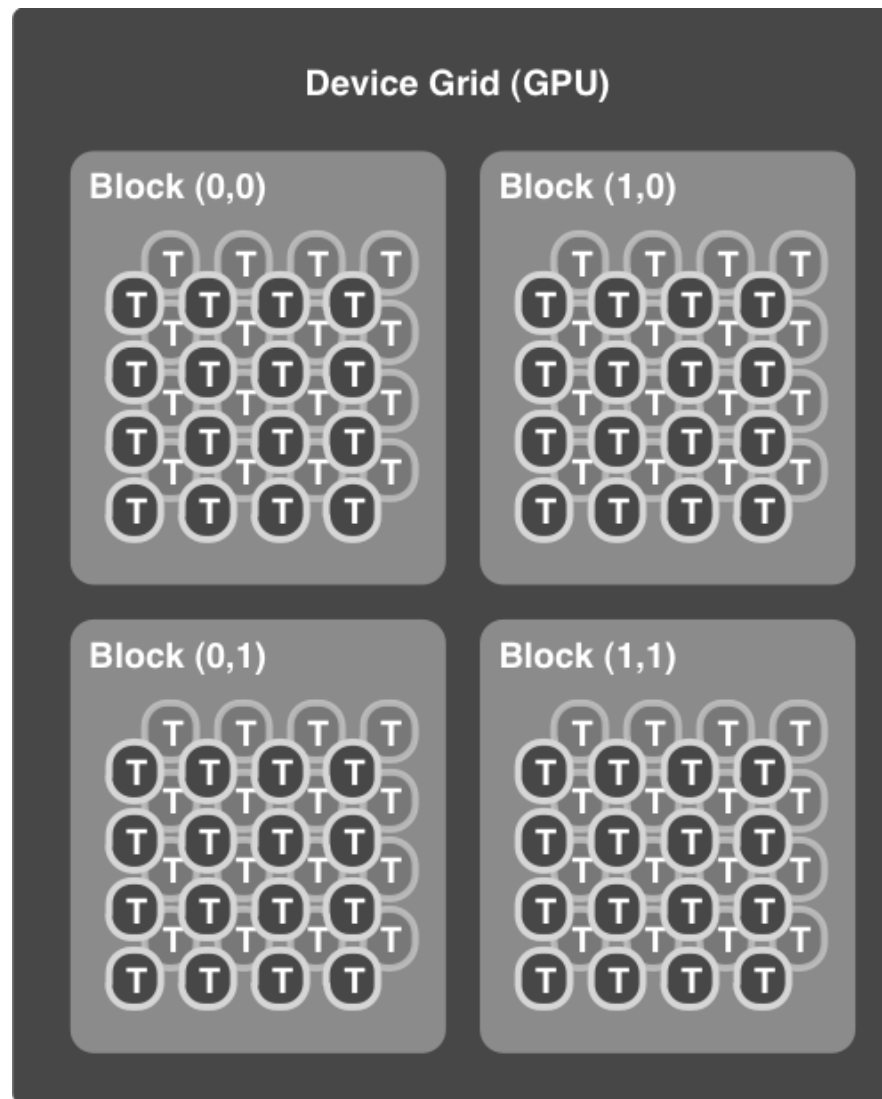
Fixed-function pipeline



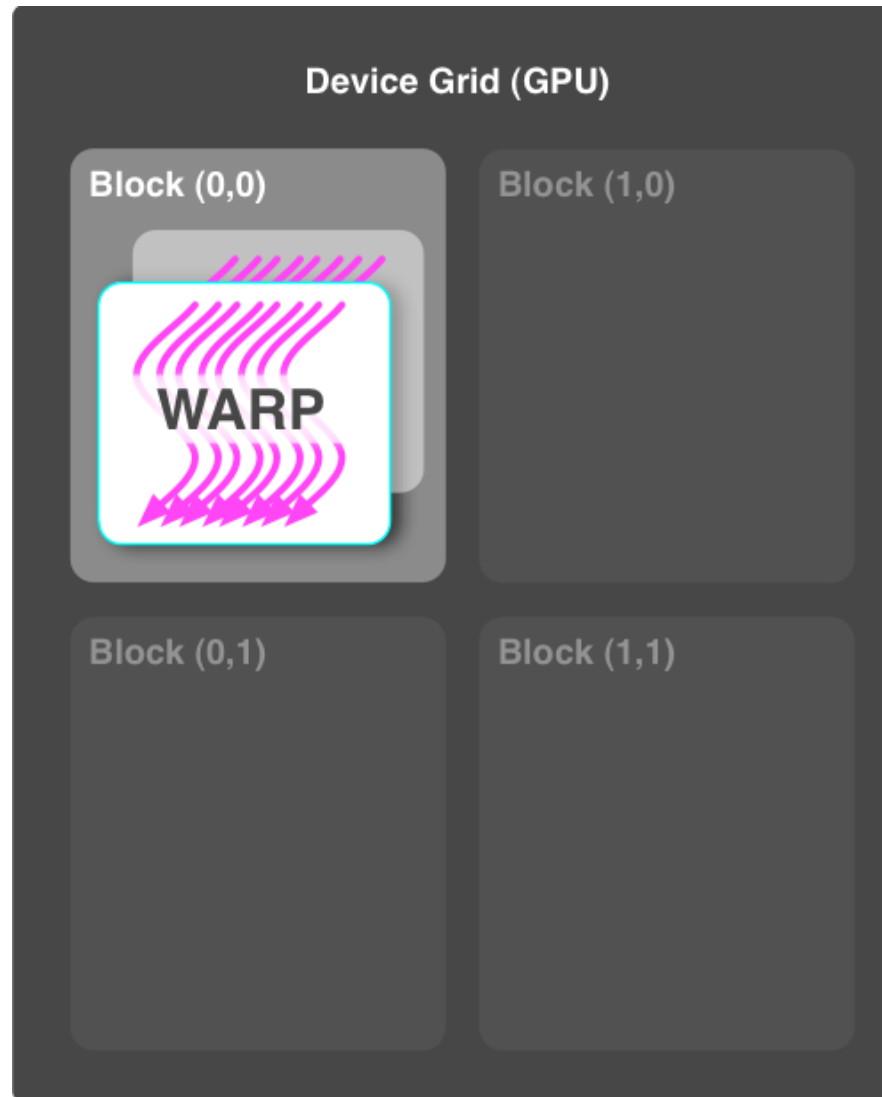
Ein CUDA Programm



CUDA Device Architektur



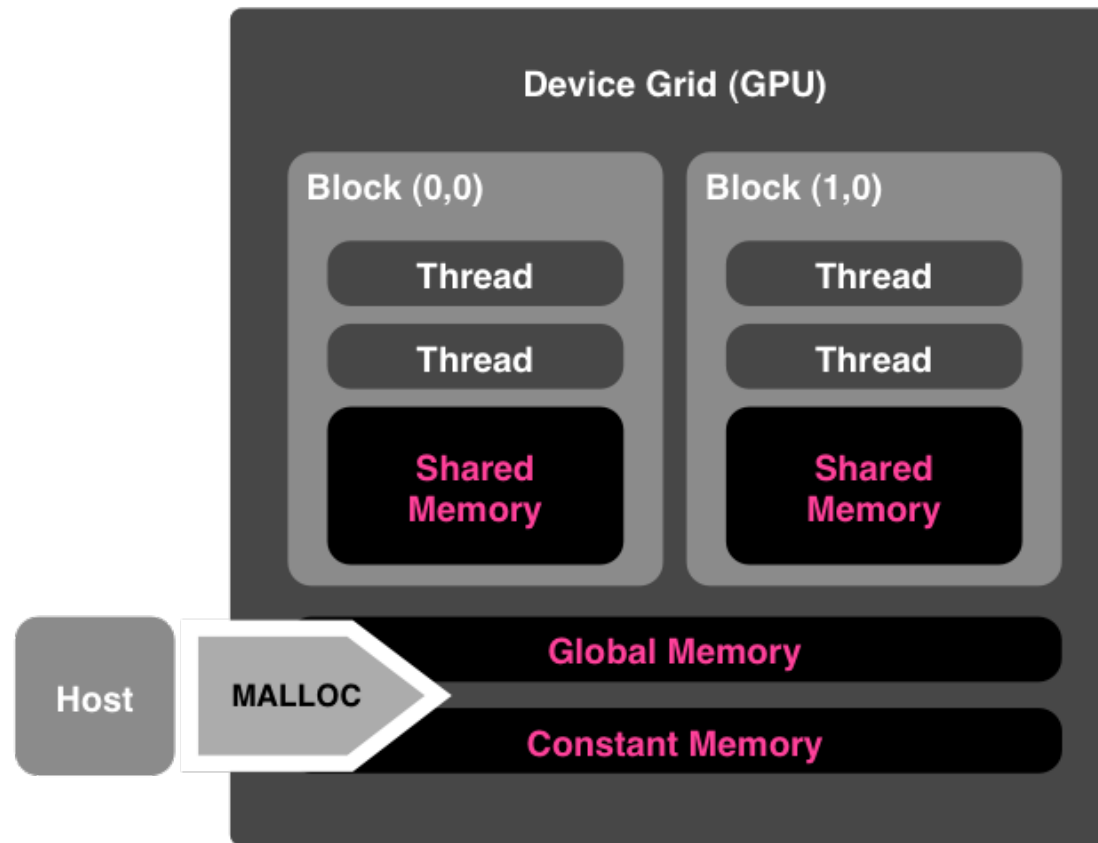
CUDA Device Architektur



CUDA Spracherweiterungen

Signatur	Aufruf durch	Ausführung auf	Anmerkungen
<code>__host__</code>	Host (CPU)	Host (CPU)	Standard
<code>__global__</code>	Host (CPU)	Device (GPU)	
<code>__device__</code>	Device (GPU)	Device (GPU)	

CUDA Memory Model



CUDA Memory Model (2)

Speicherallokierung

Signatur	Speicherort	Sichtbarkeit
<code>__device__</code>	Global Memory	Programmlaufzeit
<code>__constant__</code>	Constant Memory	Programmlaufzeit
<code>__shared__</code>	Shared Memory	Laufzeit des Blocks

Beispiel: Matrixprodukt

$$\underbrace{\begin{pmatrix} m_{11} & \cdots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} & \cdots & m_{nn} \end{pmatrix}}_M \cdot \underbrace{\begin{pmatrix} n_{11} & \cdots & n_{1n} \\ \vdots & \ddots & \vdots \\ n_{n1} & \cdots & n_{nn} \end{pmatrix}}_N = \underbrace{\begin{pmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nn} \end{pmatrix}}_P$$

Beispiel: Matrixprodukt

$$\underbrace{\begin{pmatrix} m_{11} & \cdots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} & \cdots & m_{nn} \end{pmatrix}}_M \cdot \underbrace{\begin{pmatrix} n_{11} & \cdots & n_{1n} \\ \vdots & \ddots & \vdots \\ n_{n1} & \cdots & n_{nn} \end{pmatrix}}_N = \underbrace{\begin{pmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nn} \end{pmatrix}}_P$$

Implementierung auf CPU

```
void mult_matrix(float *M, float *N, float *P)
{
    int i, j, k;
    for (i = 0; i < DIM; ++i) {
        for (j = 0; j < DIM; ++j) {
            float sum = 0;
            for (k = 0; k < DIM; ++k) {
                float a = M[i*DIM + k];
                float b = N[k*DIM + j];
                sum += a * b;
            }
            P[i*DIM + j] = sum;
        }
    }
}
```

Parallelisierung mit CUDA

```
void mult_matrix(float *M, float *N, float *P)
{

    /*
    * TODO:
    * - Allocate device memory
    * - Copy matrices to device
    * - Perform calculation
    * - Copy result and free matrices
    */

}
```


Parallelisierung mit CUDA

```
void mult_matrix(float *M, float *N, float *P)
{
    /* Allocate device memory */
    float *Md, *Nd, *Pd;
    size_t size = DIM*DIM * sizeof(float);
    cudaMalloc((void**) &Md, size);
    cudaMalloc((void**) &Nd, size);
    cudaMalloc((void**) &Pd, size);

    /* Copy matrices to device */
    /* Perform calculation */
    /* Copy result from device to host and free matrices */

}
```

Parallelisierung mit CUDA

```
void mult_matrix(float *M, float *N, float *P)
{
    /* Allocate device memory... */

    /* Copy matrices to device */
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    /* Perform calculation */
    /* Copy result from device to host and free matrices */

}
```

Parallelisierung mit CUDA

```
void mult_matrix(float *M, float *N, float *P)
{
    /* Allocate device memory... */
    /* Copy matrices to device... */

    /* Perform calculation */
    dim3 dimBlock(DIM, DIM);
    dim3 dimGrid(1,1);
    device_mult_matrix<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    /* Copy result from device to host and free matrices */
}
```

Parallelisierung mit CUDA

```
__global__ void device_mult_matrix(  
    float *Md, float *Nd, float *Pd)  
{  
    /* Thread ID */  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
  
    /* Compute one value per thread */  
    float v = 0;  
    int k;  
    for (k = 0; k < DIM; ++k) {  
        float Md_value = Md[ty * DIM + k];  
        float Nd_value = Nd[k * DIM + tx];  
        v += Md_value * Nd_value;  
    }  
  
    /* Write to result matrix on device */  
    Pd[ty * DIM + tx] = v;  
}
```

Parallelisierung mit CUDA

```
void mult_matrix(float *M, float *N, float *P)
{
    /* Allocate device memory... */
    /* Copy matrices to device... */
    /* Perform calculation... */

    /* Copy result from device to host */
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    /*free matrices */
    cudaFree(Md);
    cudaFree(Nd);
    cudaFree(Pd);
}
```


CUDA Spracherweiterungen (2)

```
__syncthreads();
```

Benchmark Bitonic Sort

