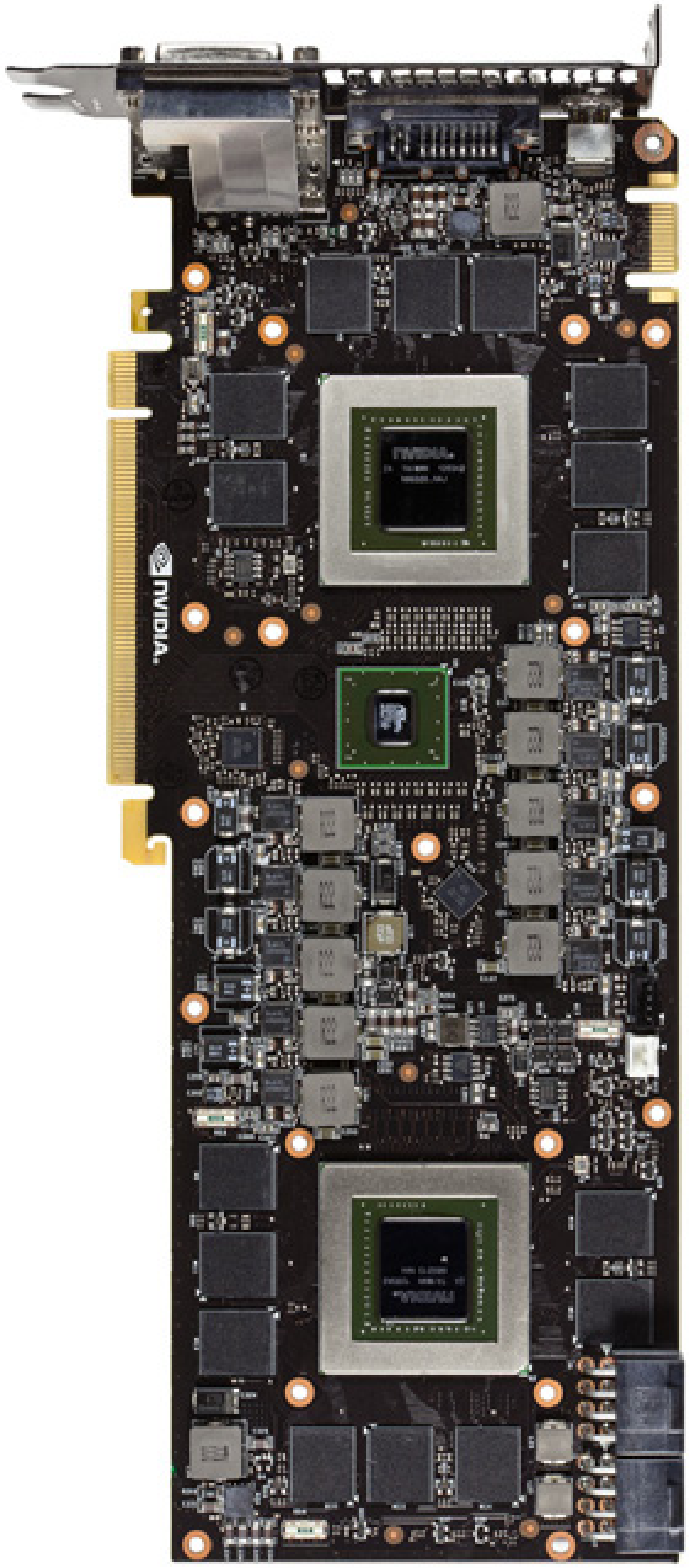
第10讲 CUDA程序设计进阶

1. 存储层次和访问模式
2. CUDA线程交互

CUDA存储层次

CUDA Memory Hierarchy

Memory



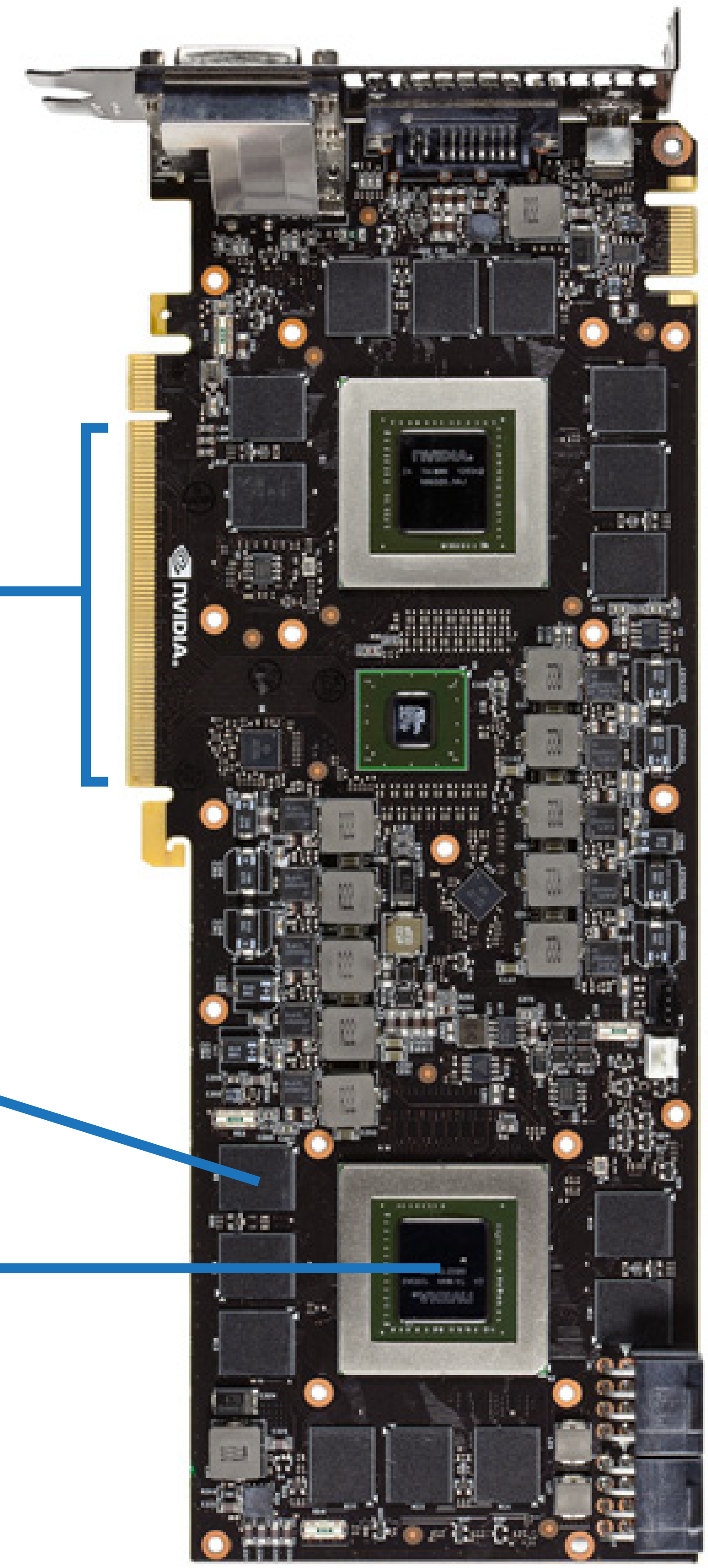
GTX 690

Memory

host memory

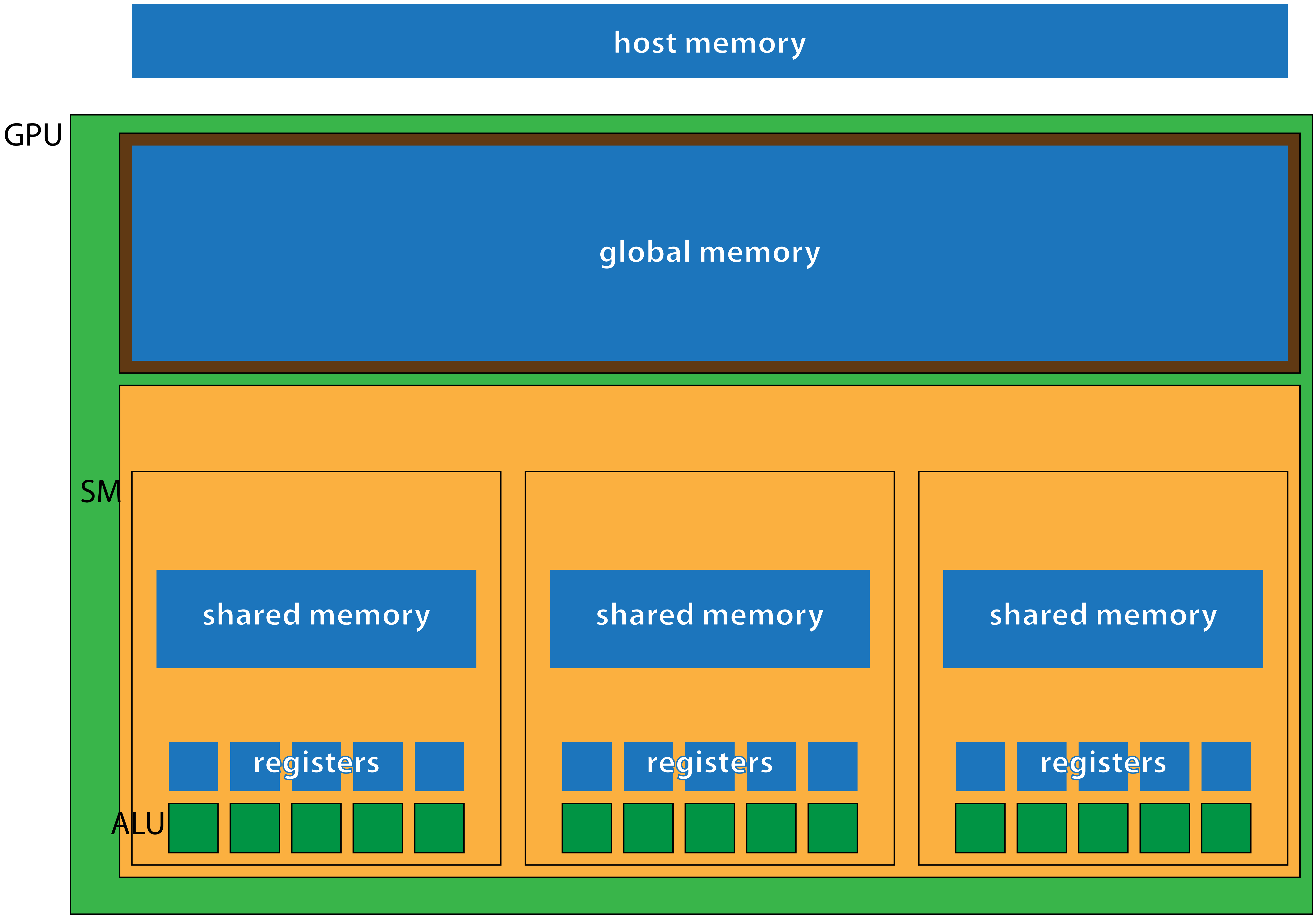
main GPU memory
(global memory)

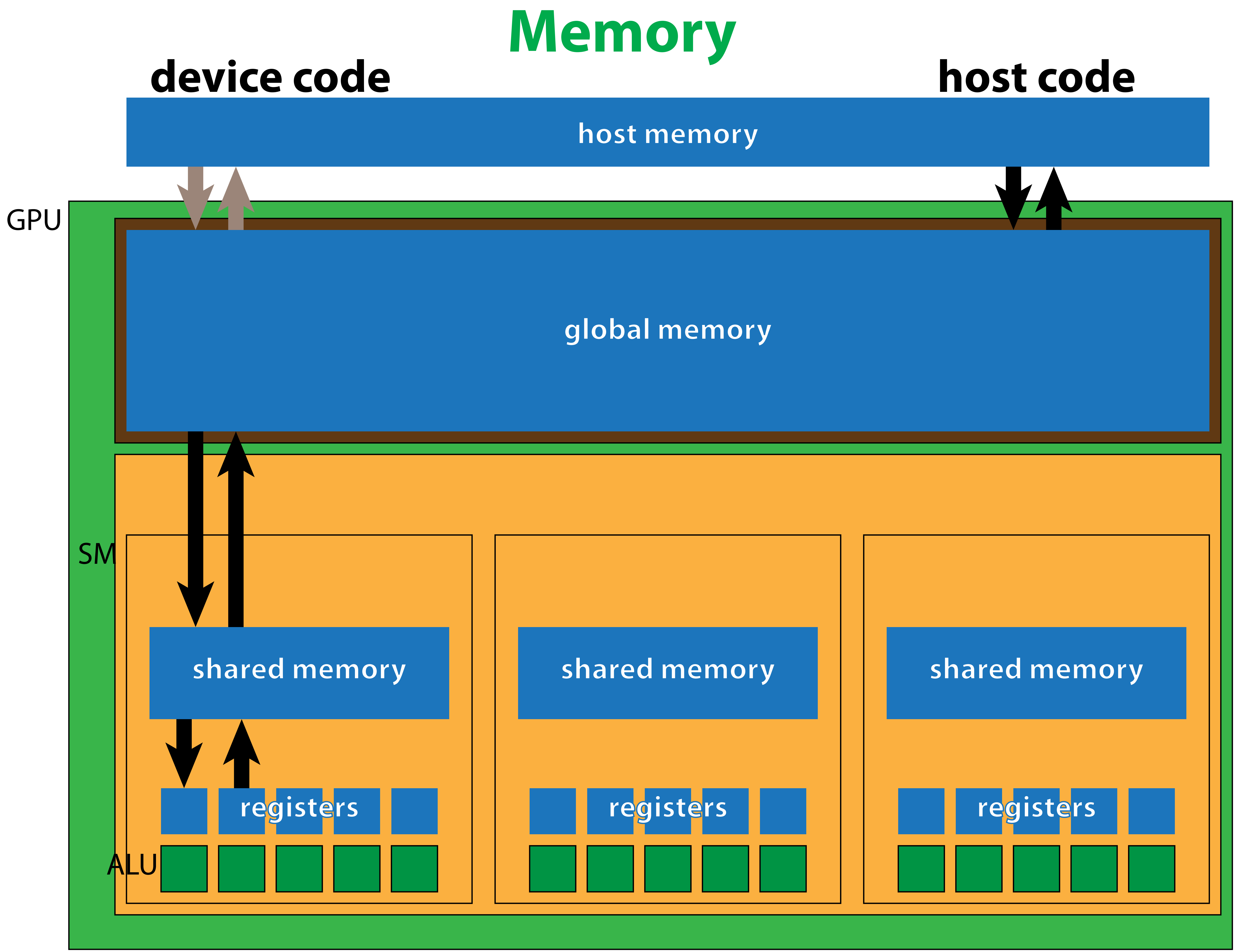
shared memory
caches
registers



GTX 690

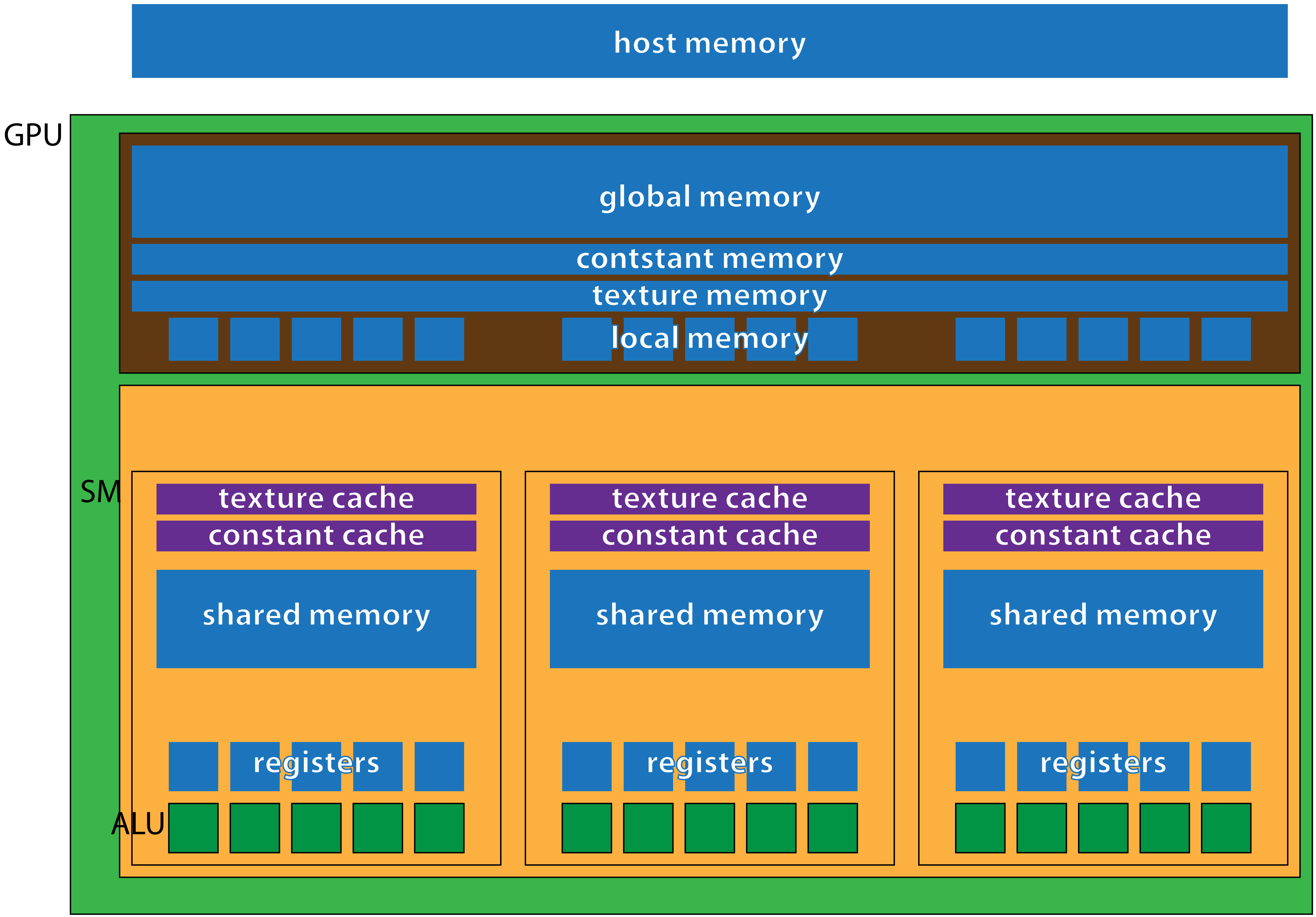
Memory


host memory
GPU
global memory
SM
shared memory
shared memory
shared memory
registers
registers
registers
ALU

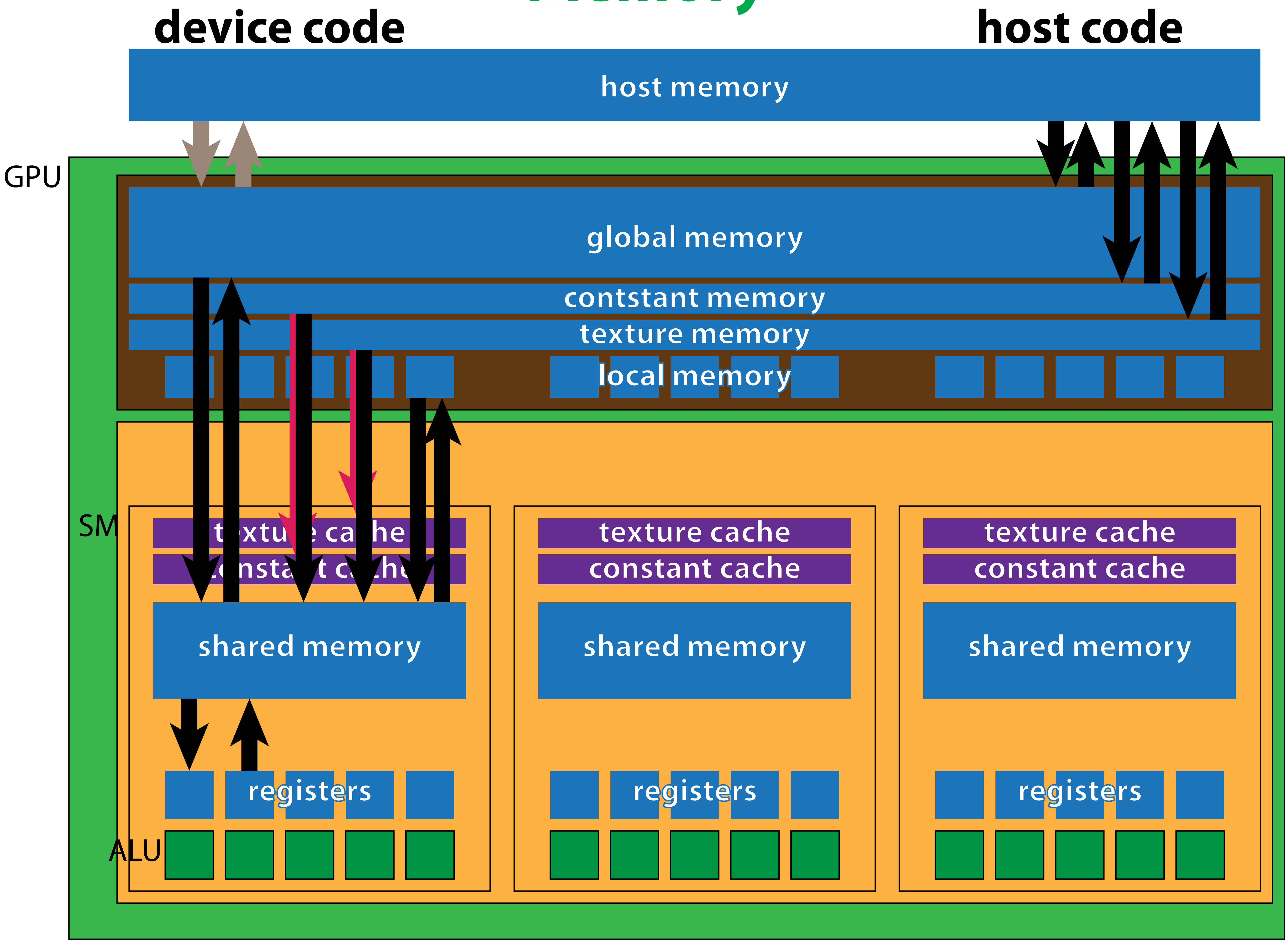


Memory
device code
host code
host memory
GPU
global memory
SM
shared memory
shared memory
shared memory
registers
registers
registers
ALU

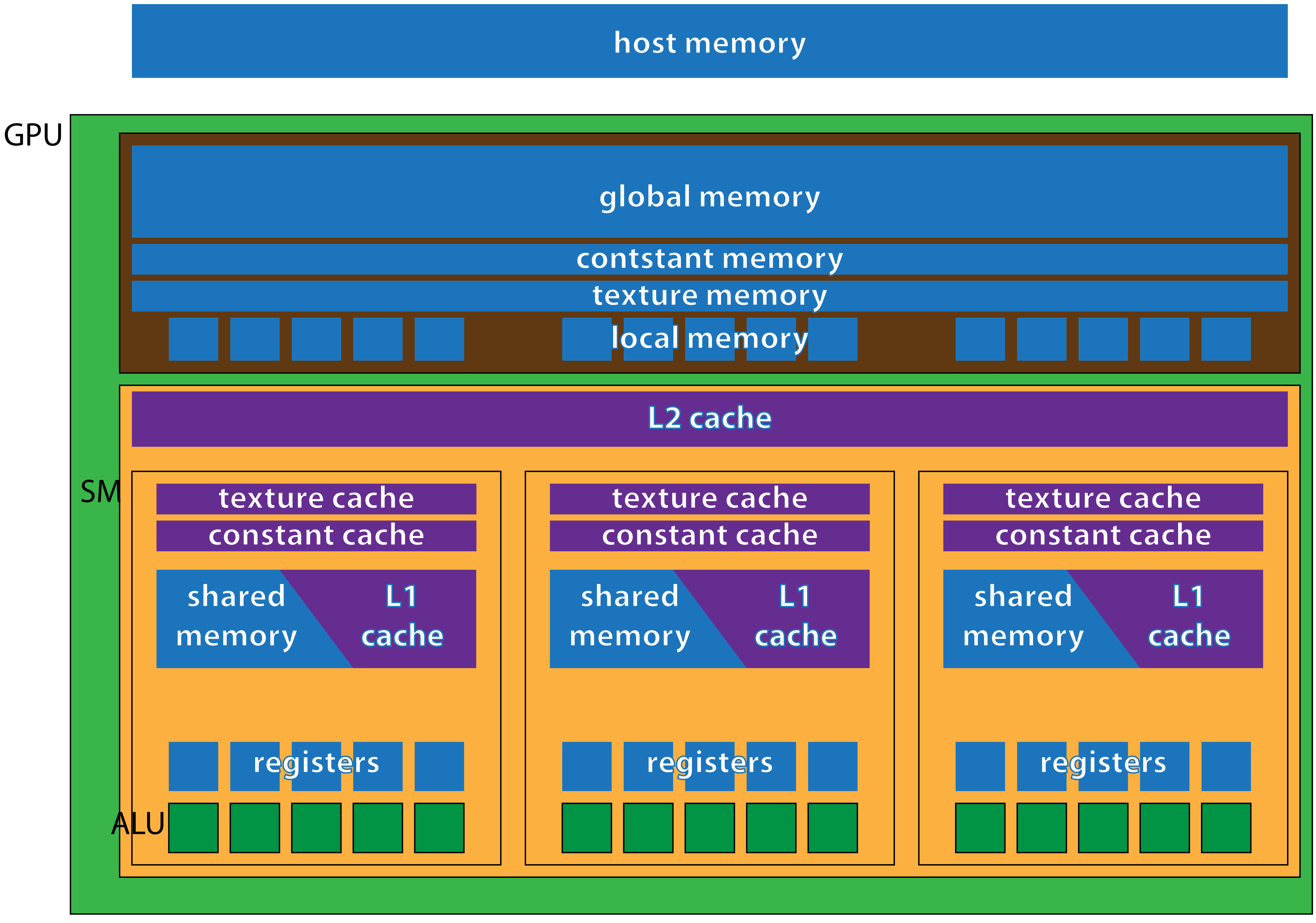
Memory


host memory
GPU
global memory
contstant memory
texture memory
local memory
SM
texture cache
constant cache
shared memory
registers
ALU
texture cache
constant cache
shared memory
registers
texture cache
constant cache
shared memory
registers

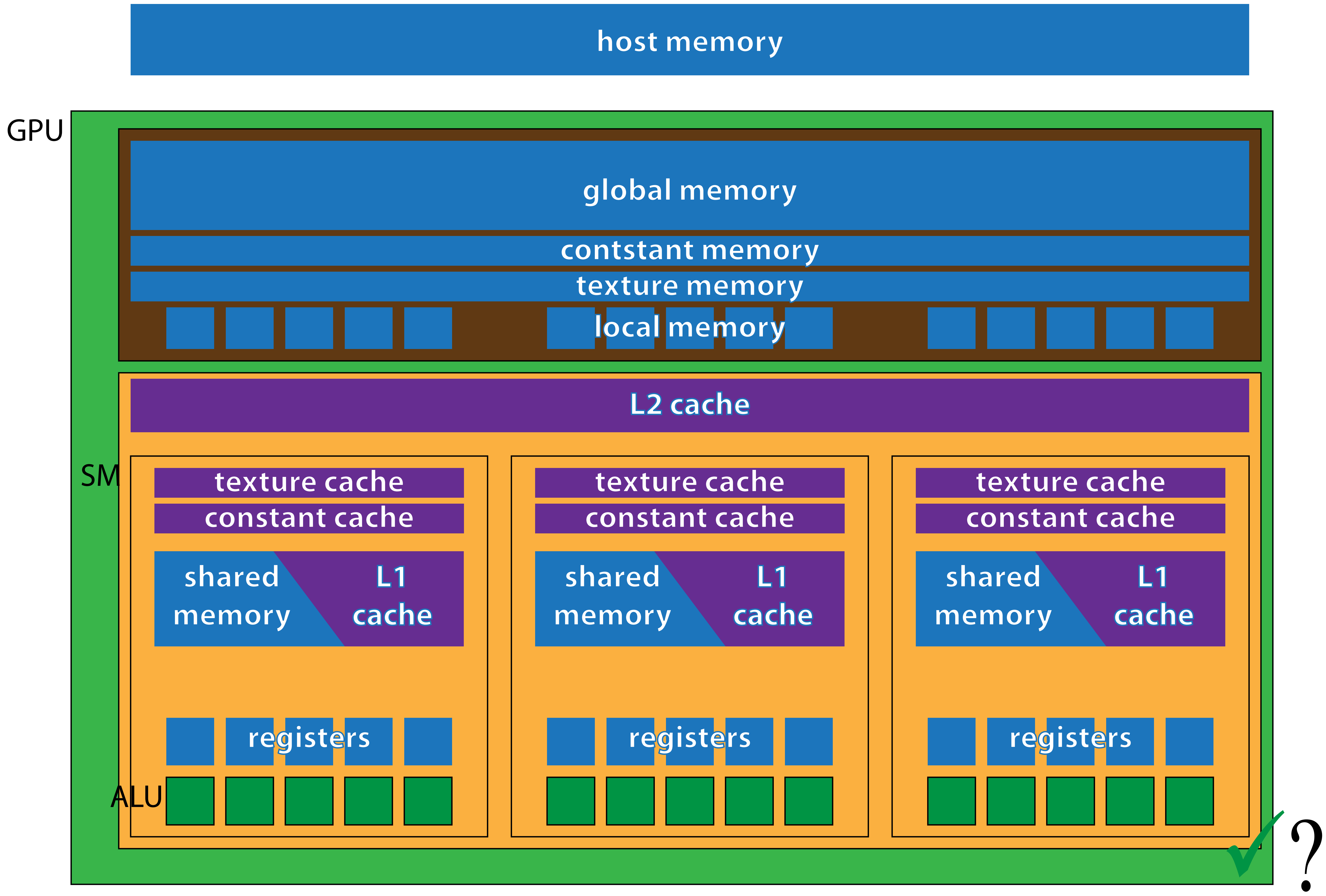
Memory


device code
host code
host memory
GPU
global memory
contstant memory
texture memory
local memory
SM
texture cache
constant cache
shared memory
texture cache
constant cache
shared memory
registers
registers
registers
ALU

Memory


host memory
GPU
global memory
contstant memory
texture memory
local memory
L2 cache
SM
texture cache
constant cache
shared memory
L1 cache
registers
ALU
texture cache
constant cache
shared memory
L1 cache
registers
texture cache
constant cache
shared memory
L1 cache
registers

Memory


host memory
GPU
global memory
contstant memory
texture memory
local memory
L2 cache
SM
texture cache
constant cache
shared memory
L1 cache
registers
ALU
texture cache
constant cache
shared memory
L1 cache
registers
texture cache
constant cache
shared memory
L1 cache
registers
?

Memory Usage

global memory

Dynamic

```
int* devPtr;
cudaMalloc(&devPtr, sizeof(int));

cudaMemcpy(
  devPtr, hostPtr, sizeof(int),
  cudaMemcpyHostToDevice
)
cudaMemcpy(
  hostPtr, devPtr, sizeof(int),
  cudaMemcpyDeviceToHost
)

cudaFree(devPtr)
```

Static

```
__device__ int devVar;

cudaMemcpyToSymbol(
  devVar, &hostVar, sizeof(int), 0,
  cudaMemcpyHostToDevice
)
cudaMemcpyFromSymbol(
  &hostVar, devVar, sizeof(int), 0,
  cudaMemcpyDeviceToHost
)
```

automatically at the end of application

```
__host__ cudaError_t cudaMemcpyToSymbol ( const T&symbol, const void*src, size_t count, size_t offset =0, cudaMemcpyKind kind = cudaMemcpyHostToDevice )

__host__ cudaError_t cudaMemcpyFromSymbol ( void* dst, const T& symbol, size_t count, size_t offset = 0, cudaMemcpyKind kind = cudaMemcpyDeviceToHost )
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
  add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
  addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
  add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
  addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
  add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
  addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
  add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
  addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
! add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
! addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
! add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
! addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
! add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
! addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
  add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
  addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
! add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
! addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
  add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
  addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
! add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
! addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory Usage

global memory

Dynamic

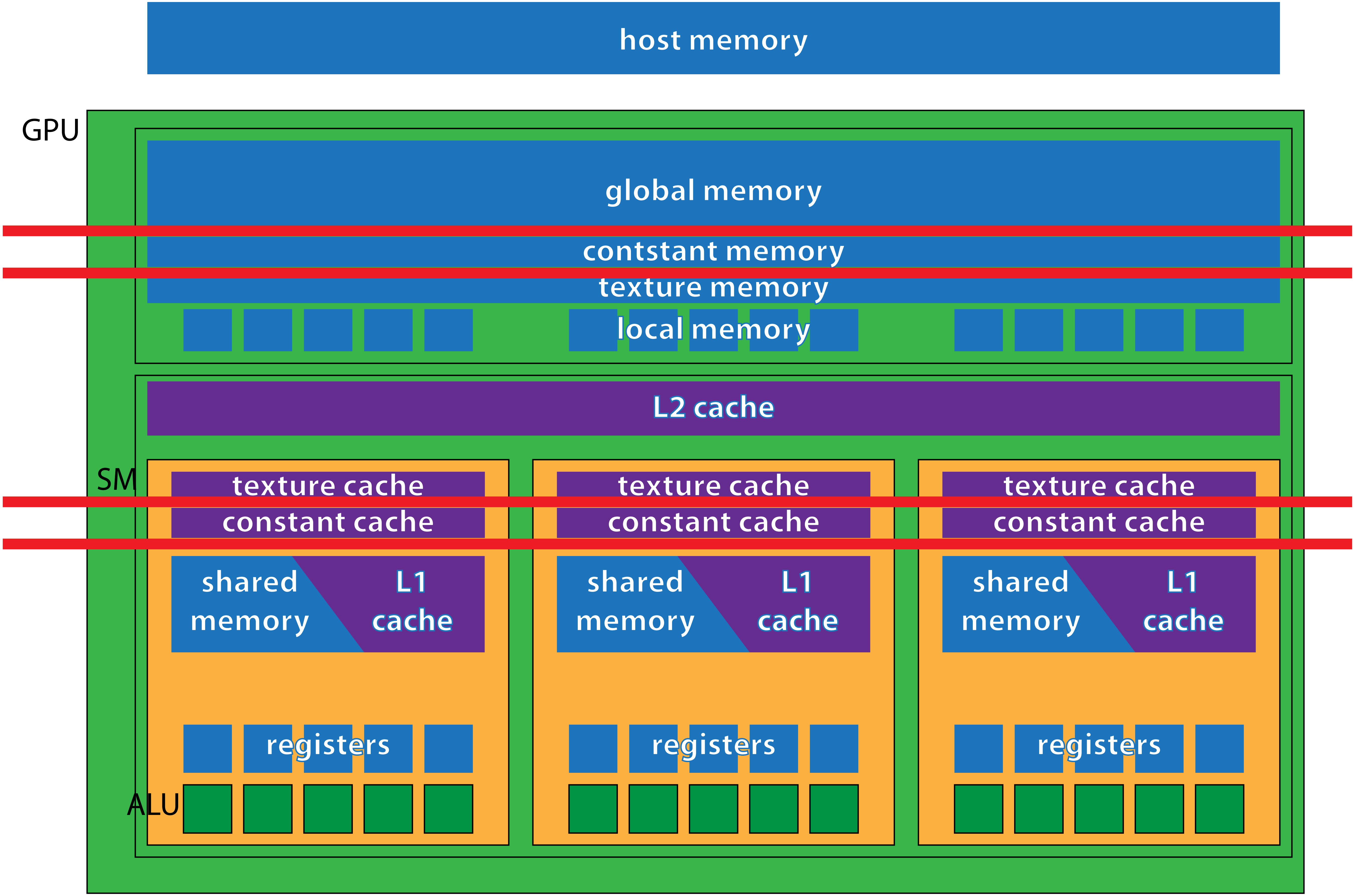
```
__global__ add4f(float* u, float* v) {
  int i=threadIdx.x;
  u[i]+=v[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  float* devU, devV;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMalloc(&devV, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(devV, hostV, size,
    cudaMemcpyHostToDevice);
! add4f<<<1,4>>>(devU, devV);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devV);
  cudaFree(devU);
  return 0;
}
```

Static

```
__device__ float devU[4];
__device__ float devV[4];

__global__ addUV() {
  int i=threadIdx.x;
  devU[i]+=devV[i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float hostV[4] = {1, 2, 3, 4};
  size_t size = sizeof(float)*4;
  cudaMemcpyToSymbol(devU, hostU, size,
    0, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(devV, hostV, size,
    0, cudaMemcpyHostToDevice);
! addUV<<<1,4>>>();
  cudaMemcpyFromSymbol(hostU, devU, size,
    0, cudaMemcpyDeviceToHost);
  return 0;
}
```

✓?

Memory Usage



Memory Usage


```
__constant__ int devVar;
```

```
__constant__ float devData[256];
float hostdata[256];
```

```
cudaMemcpyToSymbol(
  devVar, &hostVar, sizeof(int), 0,
  cudaMemcpyHostToDevice
)
cudaMemcpyFromSymbol(
  &hostVar, devVar, sizeof(int), 0,
  cudaMemcpyDeviceToHost
)
```

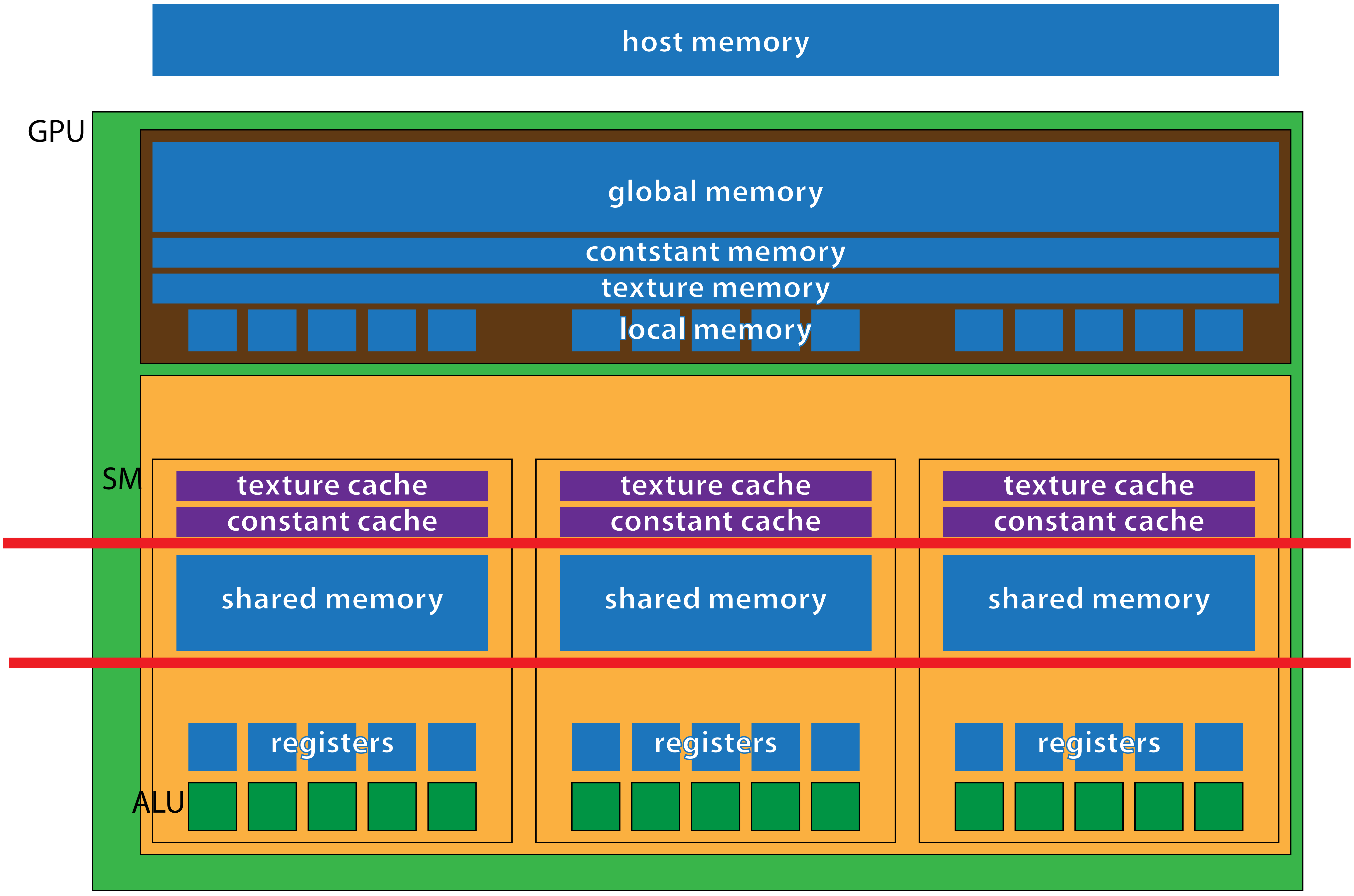
```
cudaMemcpyToSymbol(
    devData, hostdata, 256*sizeof(int), 0,
  cudaMemcpyHostToDevice
)
cudaMemcpyFromSymbol(
    hostdata, devData, 256*sizeof(int), 0,
  cudaMemcpyDeviceToHost
)
```

automatically at the end of application

Memory Usage

- parameters - up to 4KB [Fermi] [Kepler]
- Load Uniform [Fermi] [Kepler?]
  - variable resides in global memory
  - read-only in the kernel
  - not dependent on threadIdx

Memory Usage



Memory Usage

Static

- file scope
- device function

```
__shared__ int shArr[4];
```

size is fixed at compile time

Dynamic

- file scope
- device function

```
extern __shared__ int shArr[];

kernel<<<grid,block,sizeof(int)*4>>>();
```

size is a run-time parameter

all extern shared variables occupy same memory

not accesible from host

automatically at the end of kernel

Memory Usage

Static

$u_i = u_i^2 + u_{n-i-1}$

```
__global__ example(float* u) {
  int i=threadIdx.x;
  __shared__ float  tmp[4];
  tmp[i]=u[i];
  u[i]=tmp[i]*tmp[i]+tmp[3-i];
}


int main() {
  float hostU[4] = {1, 2, 3, 4};
  float* devU;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  example<<<1,4>>>(devU);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devU);
  return 0;
}
```

Dynamic

```
extern __shared__ float  tmp[];

__global__ example(float* u) {
  int i=threadIdx.x;
  tmp[i]=u[i];
  u[i]=tmp[i]*tmp[i]+tmp[3-i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float* devU;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  example<<<1,4,size>>>(devU);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devU);
  return 0;
}
```

Memory Usage

$u_i = u_i^2 + u_{n-i-1}$

Static

```
__global__ example(float* u) {
  int i=threadIdx.x;
  __shared__ float  tmp[4];
  tmp[i]=u[i];
  u[i]=tmp[i]*tmp[i]+tmp[3-i];
}


int main() {
  float hostU[4] = {1, 2, 3, 4};
  float* devU;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  example<<<1,4>>>(devU);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devU);
  return 0;
}
```

Dynamic

```
extern __shared__ float  tmp[];

__global__ example(float* u) {
  int i=threadIdx.x;
  tmp[i]=u[i];
  u[i]=tmp[i]*tmp[i]+tmp[3-i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float* devU;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  example<<<1,4,size>>>(devU);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devU);
  return 0;
}
```

Memory Usage

Static

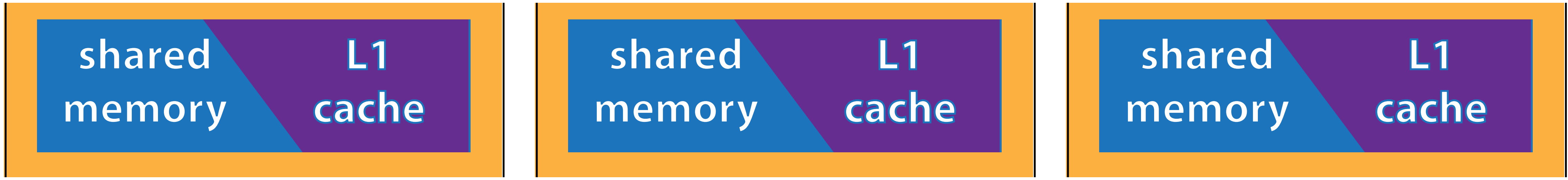
$u_i = u_i^2 + u_{n-i-1}$

```
__global__ example(float* u) {
  int i=threadIdx.x;
  __shared__ float  tmp[4];
  tmp[i]=u[i];
  u[i]=tmp[i]*tmp[i]+tmp[3-i];
}


int main() {
  float hostU[4] = {1, 2, 3, 4};
  float* devU;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  example<<<1,4>>>(devU);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devU);
  return 0;
}
```

Dynamic

```
extern __shared__ float  tmp[];

__global__ example(float* u) {
  int i=threadIdx.x;
  tmp[i]=u[i];
  u[i]=tmp[i]*tmp[i]+tmp[3-i];
}

int main() {
  float hostU[4] = {1, 2, 3, 4};
  float* devU;
  size_t size = sizeof(float)*4;
  cudaMalloc(&devU, size);
  cudaMemcpy(devU, hostU, size,
    cudaMemcpyHostToDevice);
  example<<<1,4,size>>>(devU);
  cudaMemcpy(hostU, devU, size,
    cudaMemcpyDeviceToHost);
  cudaFree(devU);
  return 0;
}
```

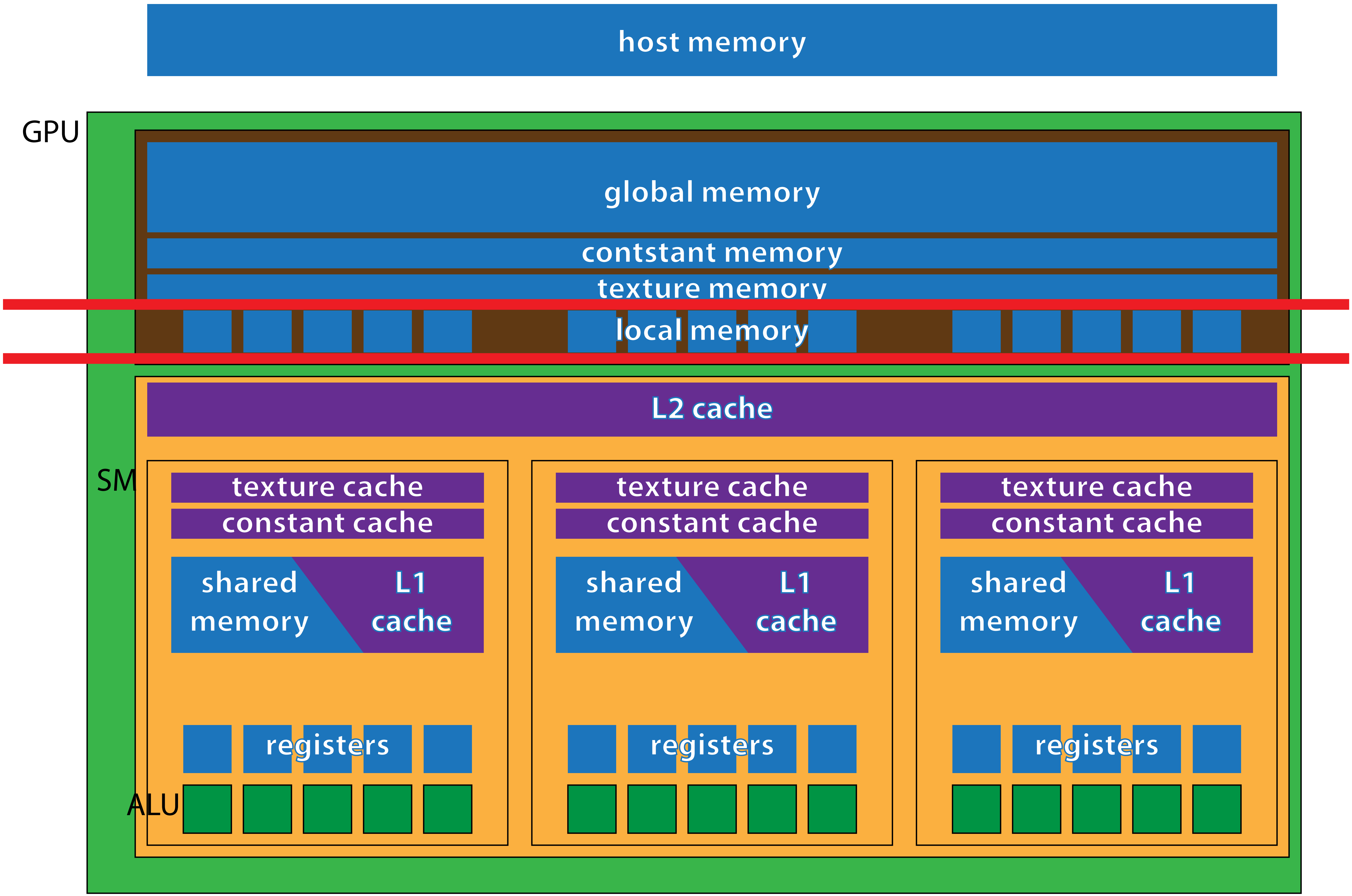
Memory Usage



- few bytes for control
- parameters - up to 256B [pre-Fermi]
- L1 cache - 16KB, 32KB or 48KB [Fermi] [Kepler]

[Kepler] only

Memory Usage



Memory Usage

local memory

- per-thread, but slow (unless L1 or L2)
- no explicit keyword
- used for register spills
- used when address is used
- used with recursive functions

Memory Usage

local memory

- per-thread, but slow (unless L1 or L2)
- no explicit keyword
- used for register spills
- used when address is used
- used with recursive functions

$$\max \frac{\text{regs}}{\text{thread}}$$

[pre-Fermi]	128
[Fermi]	63
[Kepler]	63/255

Memory Usage

local memory

- per-thread, but slow (unless L1 or L2)
- no explicit keyword
- used for register spills
- used when address is used
- used with recursive functions

	$\max \frac{\text{regs}}{\text{thread}}$	$\max \frac{\text{regs}}{\text{SM}}$
[pre-Fermi]	128	8K/16K
[Fermi]	63	32K
[Kepler]	63/255	64K

Memory Usage

local memory

- per-thread, but slow (unless L1 or L2)
- no explicit keyword
- used for register spills
- used when address is used
- used with recursive functions

	max $\frac{\text{regs}}{\text{thread}}$	max $\frac{\text{regs}}{\text{SM}}$	max $\frac{\text{threads}}{\text{SM}}$	如果仅使用SM最大线程数的一半线程 max $\frac{\text{regs}}{\text{thread}}$ if half
[pre-Fermi]	128	8K/16K	768/1024	21/32
[Fermi]	63	32K	1536	42
[Kepler]	63/255	64K	2048	64

Memory Usage

local memory

- per-thread, but slow (unless L1 or L2)
- no explicit keyword
- used for register spills
- used when address is used
- used with recursive functions

当使用地址时，变量将在local memory中

- `float x = ...;`
  `float *px = &x;`   使用x的地址

- `float x[4];`
  `x[i] = ...;`   使用x的地址

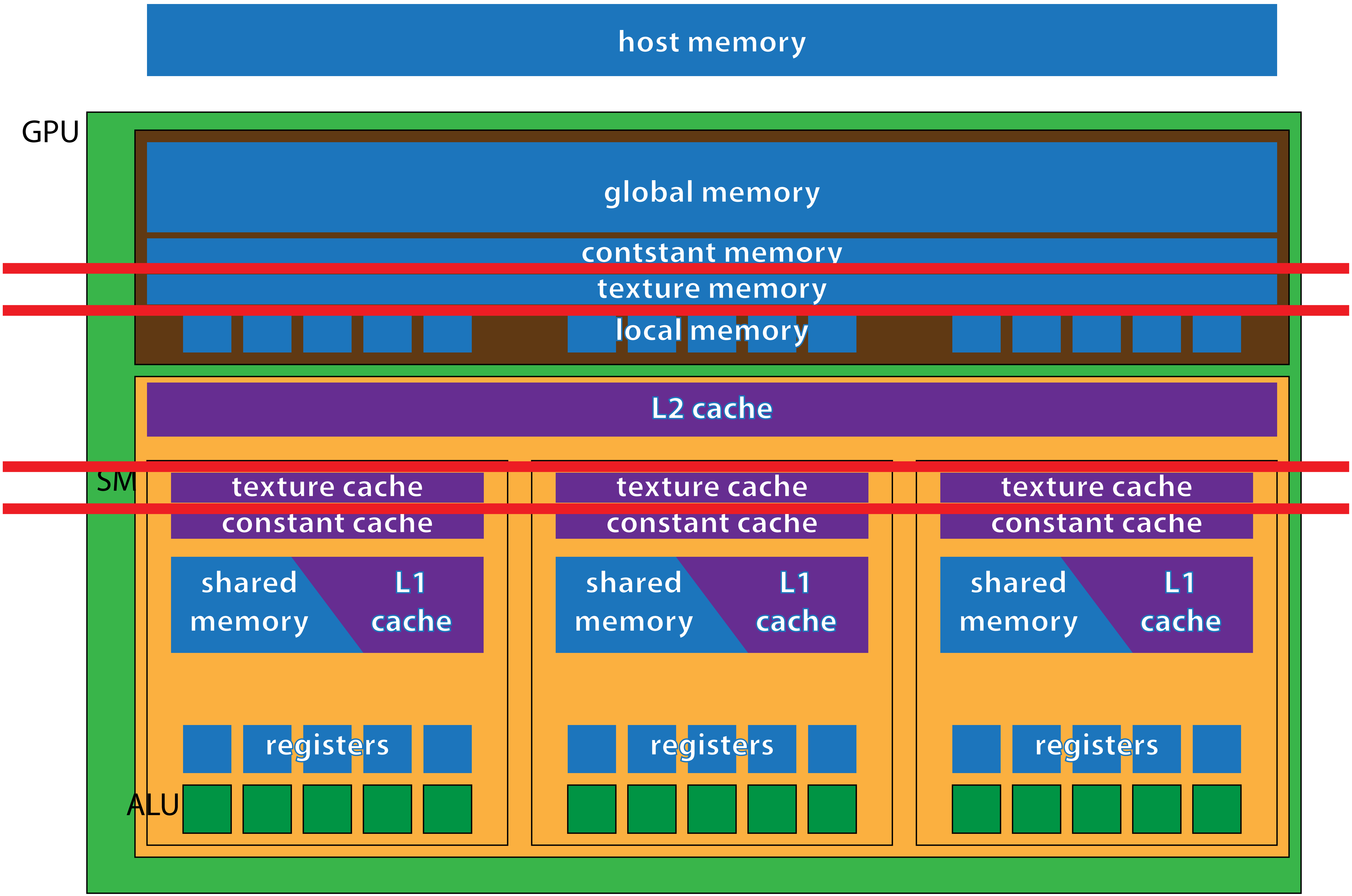
- `float x[1000];`   使用x的地址

Memory Usage

local memory

- per-thread, but slow (unless L1 or L2)
- no explicit keyword
- used for register spills
- used when address is used
- used with recursive functions

Memory Usage



Memory Usage

- separate cache in TPC [pre-Fermi] or SM [Fermi] [Kepler]
- 1D, 2D and 3D memory addressing
- normalized addressing
- value normalization
- layered textures and cubemaps [Fermi] [Kepler]
- handling border (clamp, warp, mirror)
- interpolation (nearest, linear)

Memory Usage


```
texture<float, Type, ReadMode> texRef;
```

```
texRef.normalized = ...
texRef.addressMode[0] = ...
texRef.filterMode = ...
...
```

```
float* devPtr;
cudaMalloc(&devPtr, ...);
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
size_t offset;
cudaBindTexture(&offset, texRef, devPtr, channelDesc, size);
```

Host: use `devPtr`

```
cudaUnbindTexture(texRef);
cudaFree(devPtr);
```

Memory Usage

Data type: one of `char, short, int, long, long long, float, double` or vector structures of size 2 or 4 (e.g. `short2, float4`)

```
texture<float, Type, ReadMode> texRef;
```

Value normalization:
`cudaReadModeElementType` ← default
`cudaReadModeNormalizedFloat`

Dimensionality:
`cudaTextureType1D` ← default
`cudaTextureType2D`
`cudaTextureType3D`
`cudaTextureType1DLayered`
`cudaTextureType2DLayered`
`cudaTextureTypeCubemap`
`cudaTextureTypeCubemapLayered`

```
texRef.normalized = ...
texRef.addressMode[0] = ...
texRef.filterMode = ...
...
```

```
float* devPtr;
cudaMalloc(&devPtr, ...);
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
size_t offset;
cudaBindTexture(&offset, texRef, devPtr, channelDesc, size);
```

Host: use `devPtr`

```
cudaUnbindTexture(texRef);
cudaFree(devPtr);
```

Memory Usage


```
texture<float, Type, ReadMode> texRef;
```

Normalized addressing
- `false` - `[0..N-1]` ← default
- `true` - `[0..1-1/N]`

```
texRef.normalized = ...
texRef.addressMode[0] = ...
texRef.filterMode = ...
...
```

Border handling (per each dimension)

`cudaAddressModeBorder`
`cudaAddressModeClamp` ← default
normalized addressing: `cudaAddressModeWarp`, `cudaAddressModeMirror`

more

Interpolation

`cudaFilterModePoint`
`cudaFilterModeLinear`

```
float* devPtr;
cudaMalloc(&devPtr, ...);
cudaChannelFormatDesc channe                    nelDesc<float>();
size_t offset;
cudaBindTexture(&offset, texRef, devPtr, channelDesc, size);
```

Host: use `devPtr`

```
cudaUnbindTexture(texRef);
cudaFree(devPtr);
```

Memory Usage

[pre-Fermi] [Fermi] [Kepler]

```
texture<float, Type, ReadMode> texRef;
```

```
texRef.normalized = ...
texRef.addressMode[0] = ...
texRef.filterMode = ...
...
```

```
float* devPtr;
cudaMalloc(&devPtr, ...);
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
size_t offset;
cudaBindTexture(&offset, texRef, devPtr, channelDesc, size);
```

Host: use `devPtr`

```
cudaUnbindTexture(texRef);
cudaFree(devPtr);
```

[Kepler]

```
cudaTextureDesc texDesc;
cudaResourceDesc resDesc;
```

```
cudaCreateTextureObject(...);
```

Memory Usage


```
texture<float, Type, ReadMode> texRef;
cudaBindTexture(&offset, texRef, devPtr, channelDesc, size);
```

Dimensionality:

```
cudaTextureType1D
cudaTextureType2D
cudaTextureType3D
cudaTextureType1DLayered
cudaTextureType2DLayered
cudaTextureTypeCubemap
cudaTextureTypeCubemapLayered
```

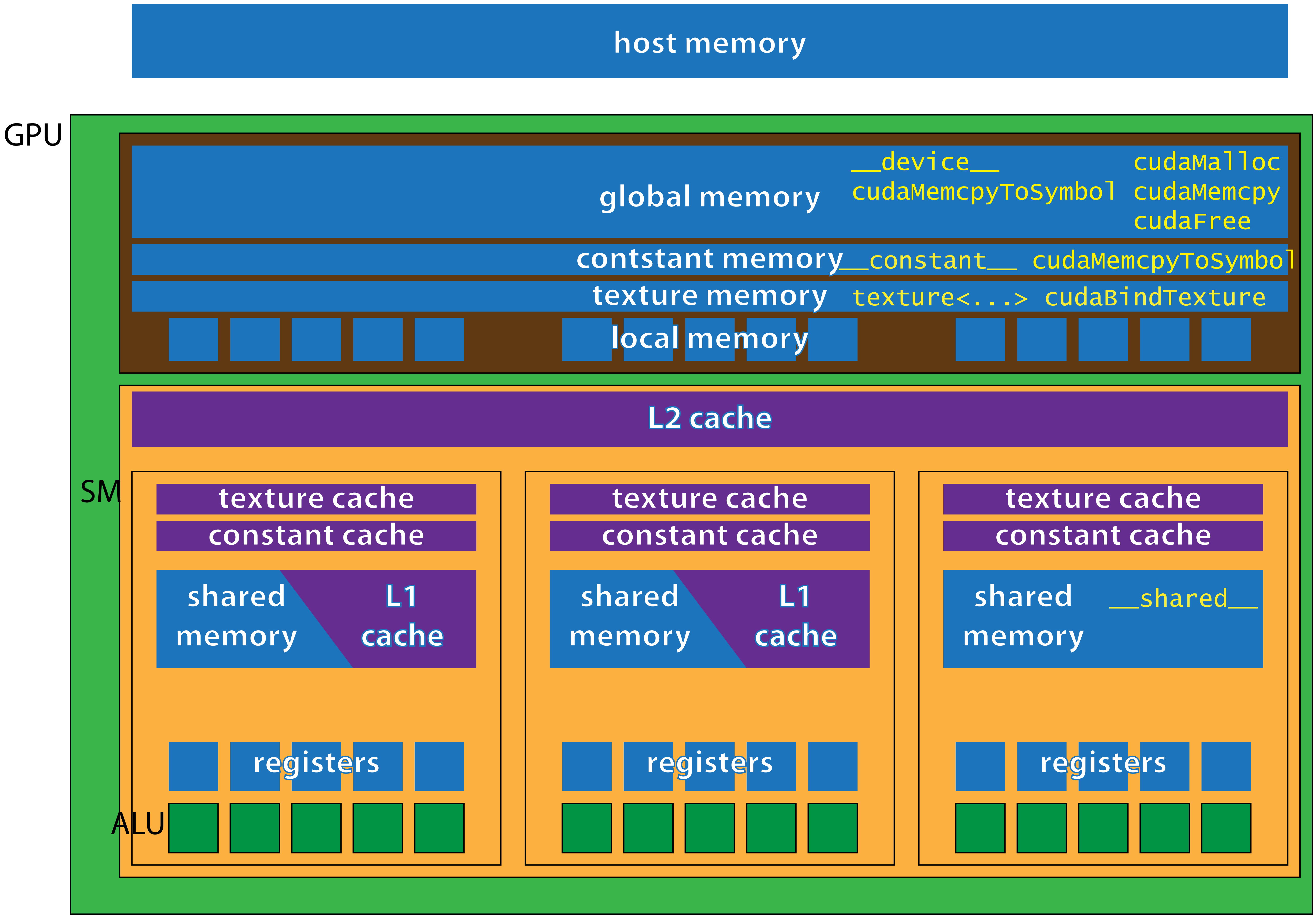
← default

Device code

```
float v = tex1Dfetch(texRef, x);
float v = tex1D(texRef, x);
float v = tex2D(texRef, x, y);
float v = tex3D(texRef, x, y, z);
...
```

← (address not normalized, no interpolation)

Recap


host memory
GPU
global memory
__device__
cudaMemcpyToSymbol
cudaMalloc
cudaMemcpy
cudaFree
contstant memory
__constant__
cudaMemcpyToSymbol
texture memory
texture<...>
cudaBindTexture
local memory
L2 cache
SM
texture cache
constant cache
shared memory
L1 cache
registers
ALU
texture cache
constant cache
shared memory
L1 cache
registers
texture cache
constant cache
shared memory
__shared__
registers

Not covered

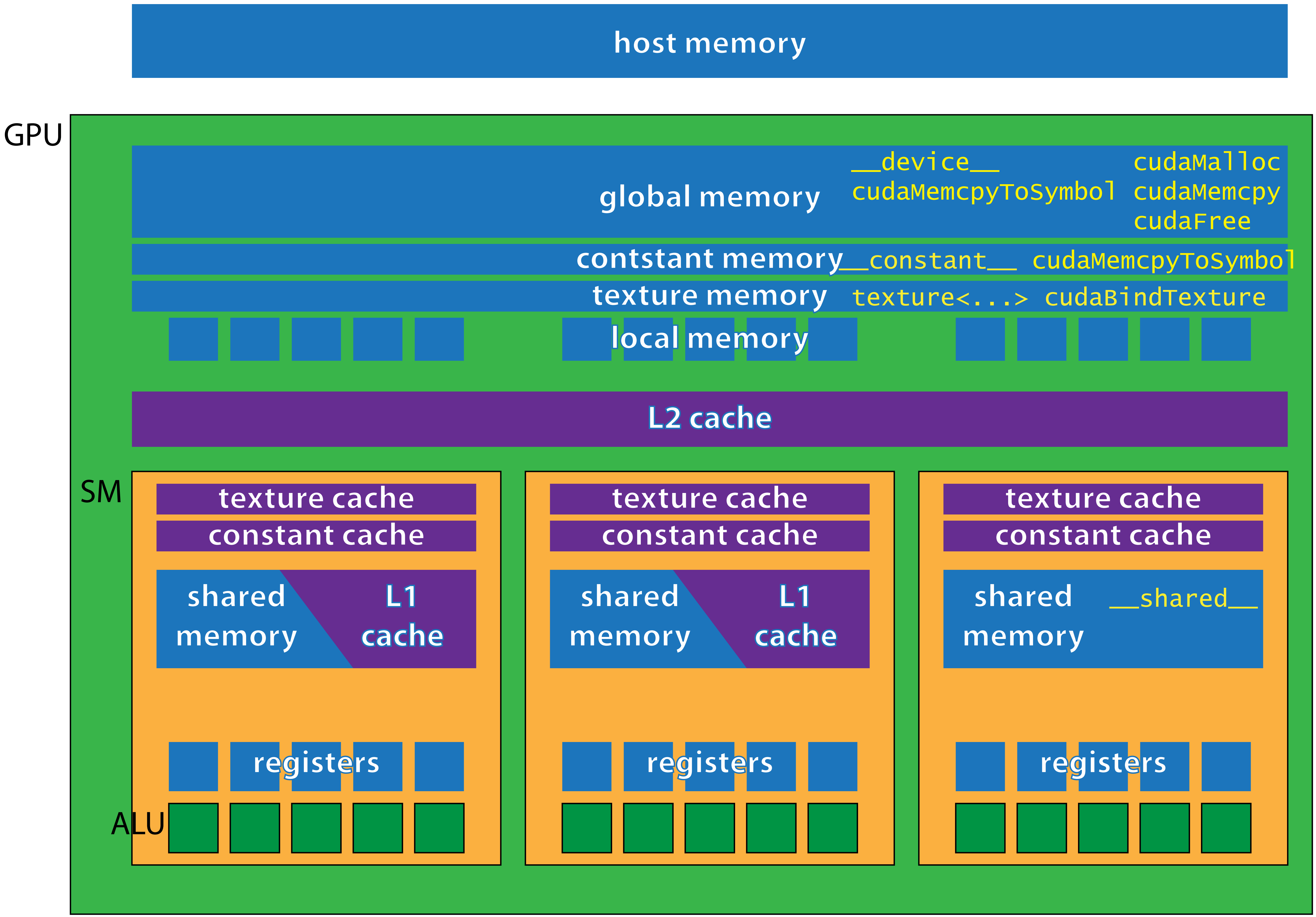
- accessing host memory from kernel
- asynchronous memory transfers
- L1 cache configuration
- efficient access patterns
- texture objects [Kepler]
- cudaArray
- cudaSurface

存储访问模式

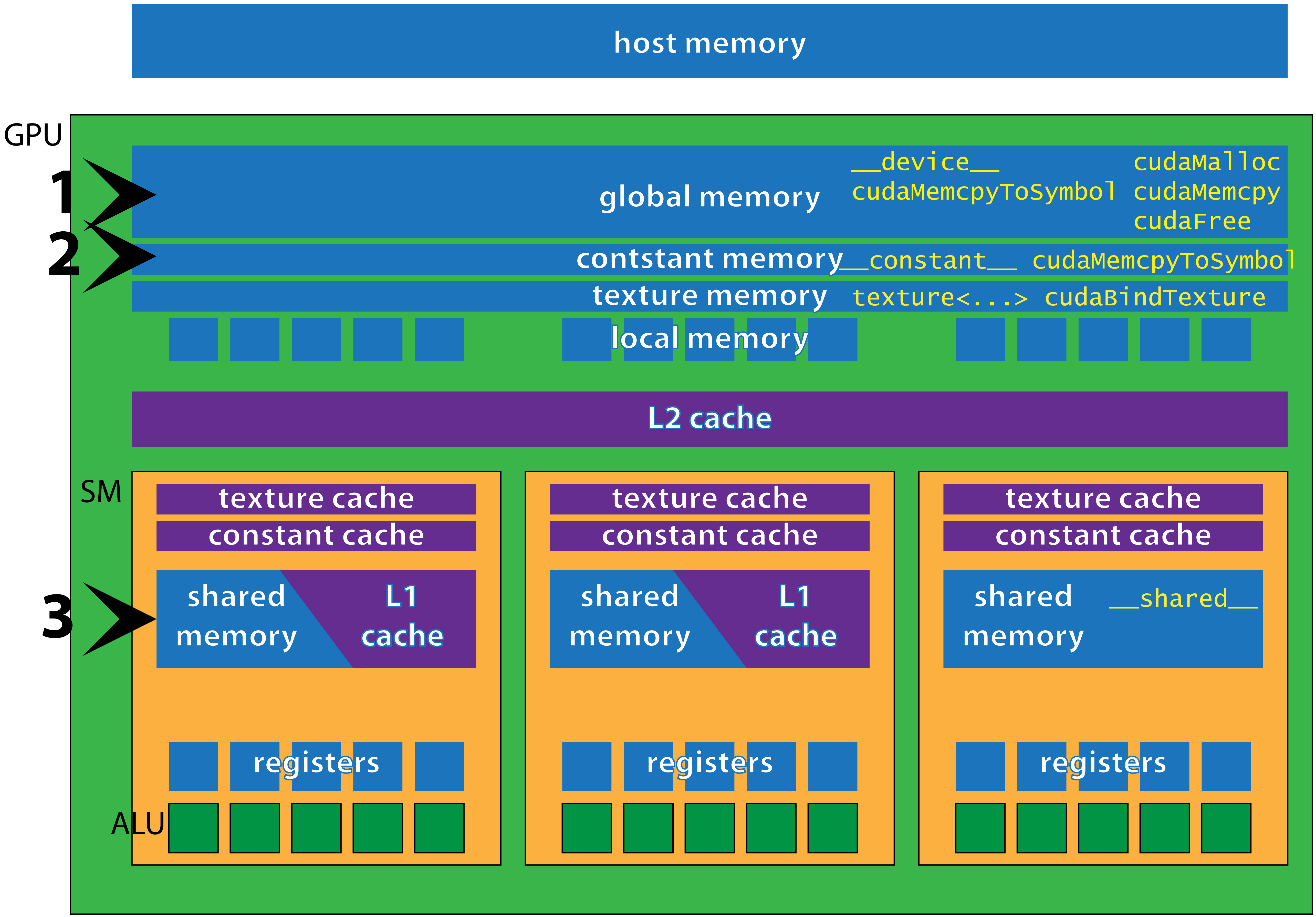
Memory access patterns

November 2018

复习、回忆


host memory
GPU
global memory
__device__
cudaMemcpyToSymbol
cudaMalloc
cudaMemcpy
cudaFree
contstant memory
__constant__
cudaMemcpyToSymbol
texture memory
texture<...>
cudaBindTexture
local memory
L2 cache
SM
texture cache
constant cache
shared memory
L1 cache
texture cache
constant cache
shared memory
L1 cache
texture cache
constant cache
shared memory
__shared__
registers
registers
registers
ALU

复习、回忆


host memory
GPU
1
2
3
global memory
__device__
cudaMemcpyToSymbol
cudaMalloc
cudaMemcpy
cudaFree
contstant memory
__constant__
cudaMemcpyToSymbol
texture memory
texture<...>
cudaBindTexture
local memory
L2 cache
SM
texture cache
constant cache
shared memory
L1 cache
texture cache
constant cache
shared memory
L1 cache
texture cache
constant cache
shared memory
__shared__
registers
registers
registers
ALU

Global memory

- off-chip
- linearly addressable

全局存储器（global memory），使用的是普通的显存。整个网格中的任意线程都能读写全局存储器的任意位置。为了能够高效的访问显存，读取和存储必须对齐，宽度为4Byte。如果没有正确的对齐，读写将被编译器拆分为多次操作，极大的影响效率。此外，多个half-warp的读写操作如果能够满足合并访问（coalesced access），那么多次访存操作会被合并成一次完成，从而提高访问效率。

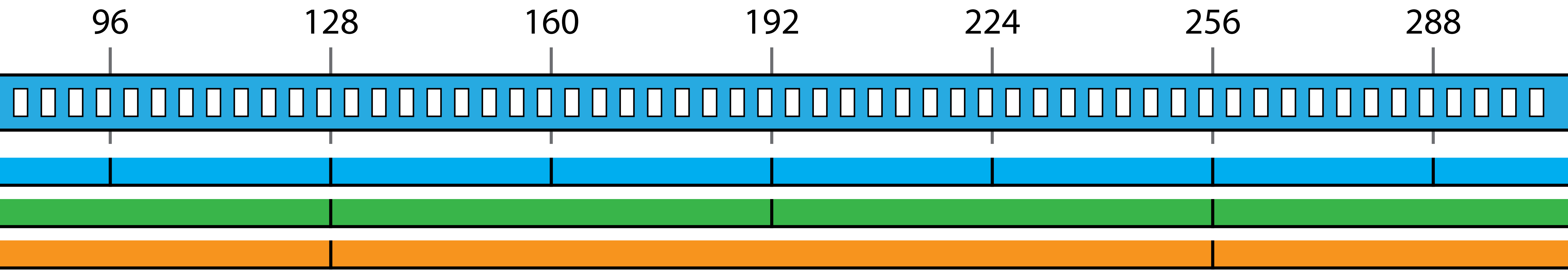
G80的合并访存条件十分严格。首先，访存的开始地址必须对齐：16x32bit的合并必须对齐到64Byte（即访存起始地址必须是64Byte的整数倍）；16x64bit的合并访存起始必须对齐到128Byte；16x128bit合并访存的起始地址必须对齐到128Byte，但是必须横跨连续的两个128Byte区域。其次，只有当第K个线程访问的就是第K个数据字时，才能实现合并访问，否则half warp中的16个访存指令就会被发射成16次单独的访存。
GT200不仅放宽了合并访问条件，而且还能支持对8bit和16bit数据字的合并访问（分别使用32Byte和64Byte传输）。在一次合并传输的数据中，并不要求线程编号和访问的数据字编号相同。其次，当访问128Byte数据时如果地址没有对齐到128Byte，在G80中会产生16次访存指令发射，而在GT200中只会产生两次合并访存。而且，这两次合并访存并不是两次128Byte的。例如，一次128Byte访存中有32Byte在一个区域中，另外一个区域中有96Byte，那么只会产生一次32Byte合并访存（对有32Byte数据的区域）和一次128Byte（对有96Byte数据的区域）。

Coalesced Global Memory Accesses

- 一个half-warp (16 threads on G80)内线程对global memory的并行访问可以被coalesced成对整块内存区域的访问, 如果以下条件满足:
  - 各个线程访问的数据长度为4, 8, 或16字节
  - 被访问地址构成一片连续内存空间
  - 第 N个线程访问第N个global memory地址
  - 起始global memory地址对齐到访问的数据长度的16倍
- 允许其间某些线程不参与内存访问

Global memory

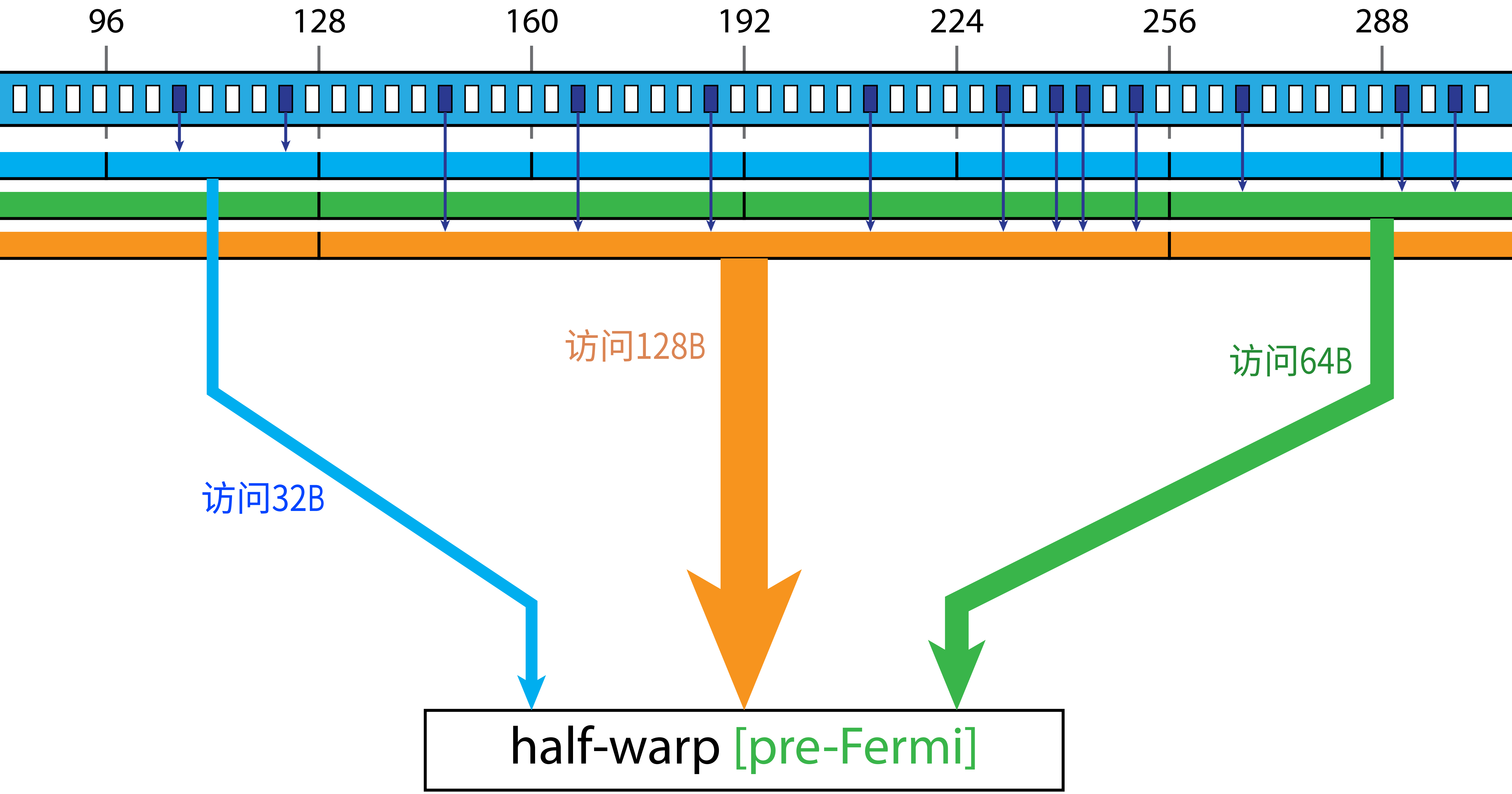
- segmented into 32B, 64B and 128B chunks



Global memory

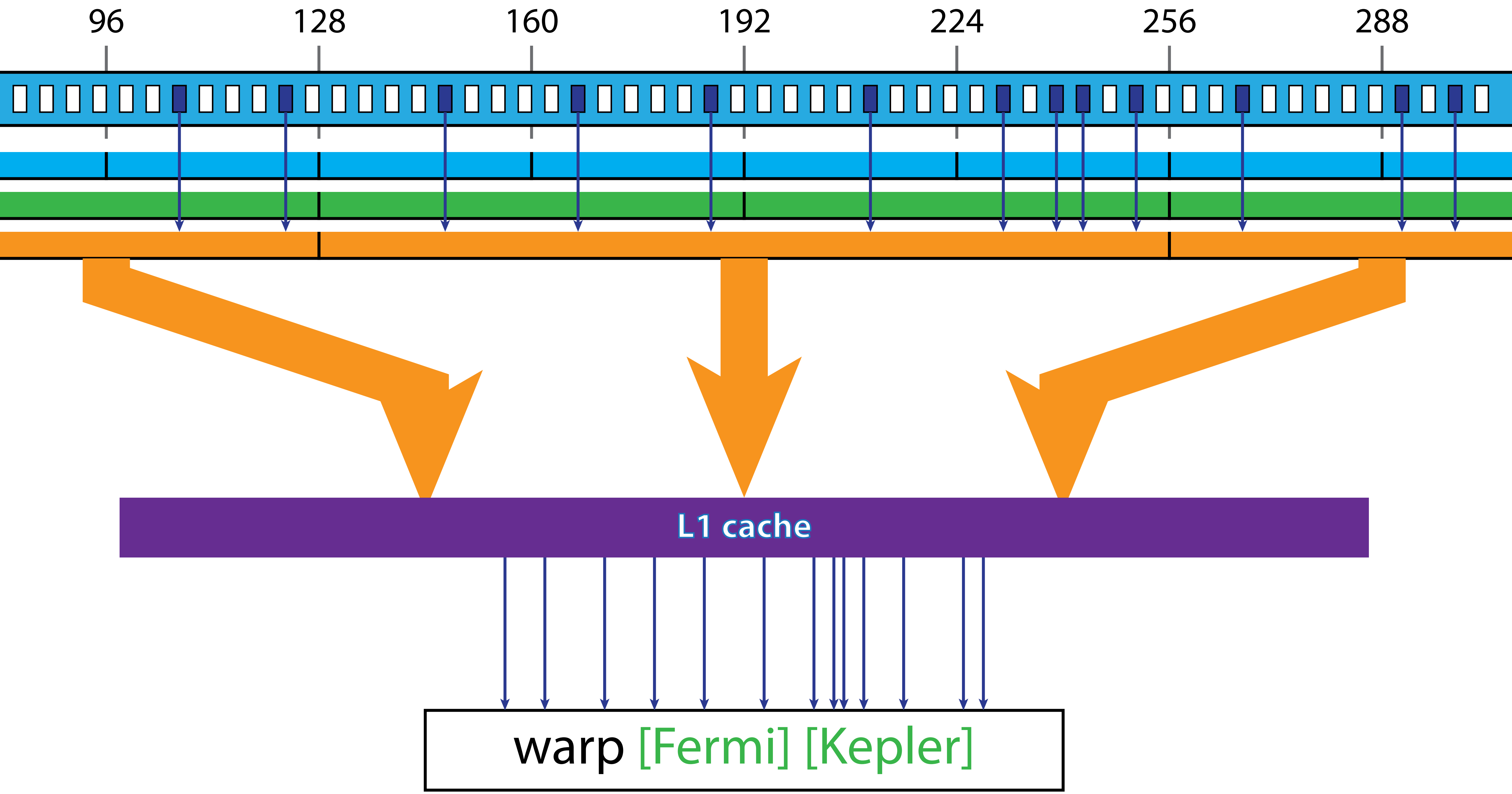
- segmented into 32B, 64B and 128B chunks

  对全局存储总是按32B，64B和128B大小对齐访问，即使只需一个字节



Global memory

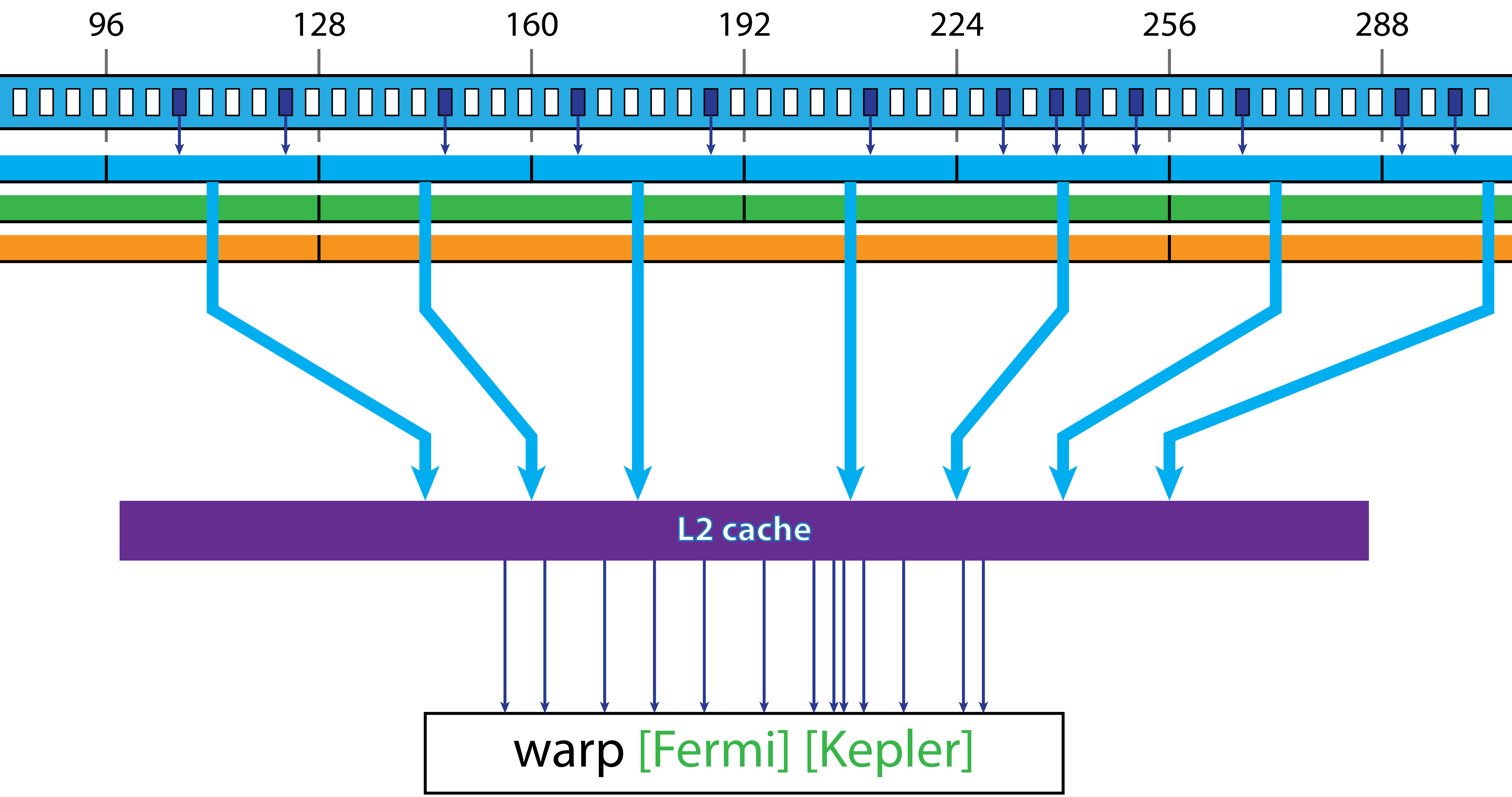
- segmented into 32B, 64B and 128B chunks



对于Fermi和Kepler架构，由于有L1 cache，会先将所涉及的所有128B的段送入cache

Global memory

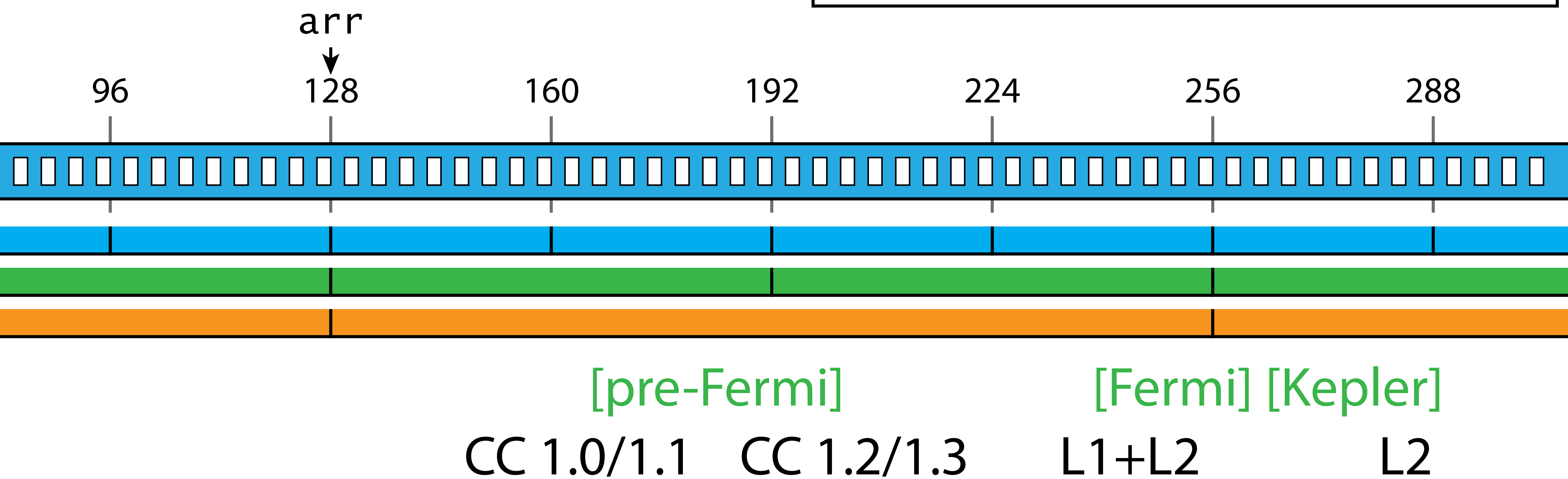
- segmented into 32B, 64B and 128B chunks



对于Fermi和Kepler架构，由于有L2 cache，会先将所涉及的所有32B的段送入cache

Global memory

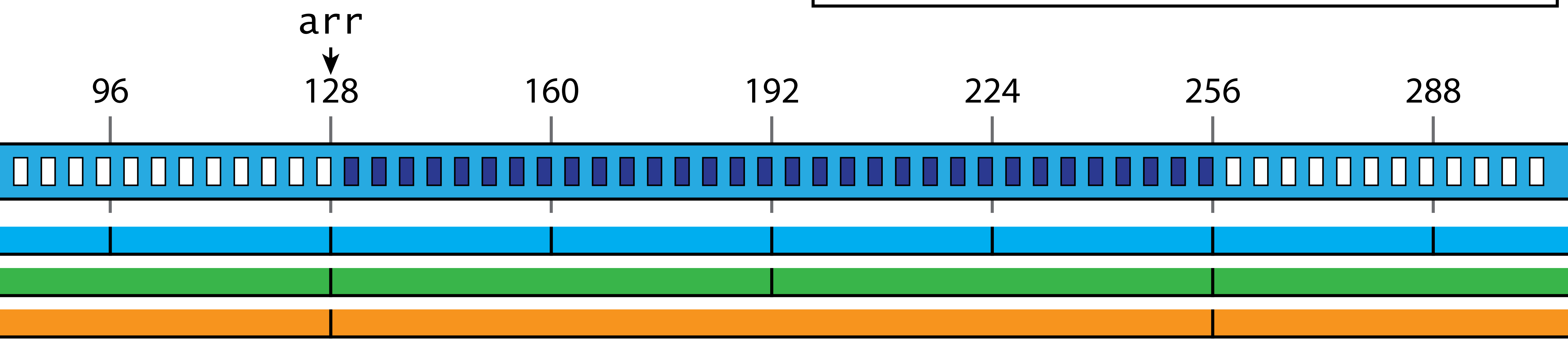
```
__device__ int arr[128];
...
int v = arr[ ?? ]
```



Global memory

对齐且顺序访问时

```
__device__ int arr[128];
...
int v = arr[ threadIdx.x ]
```



[pre-Fermi] [Fermi] [Kepler]

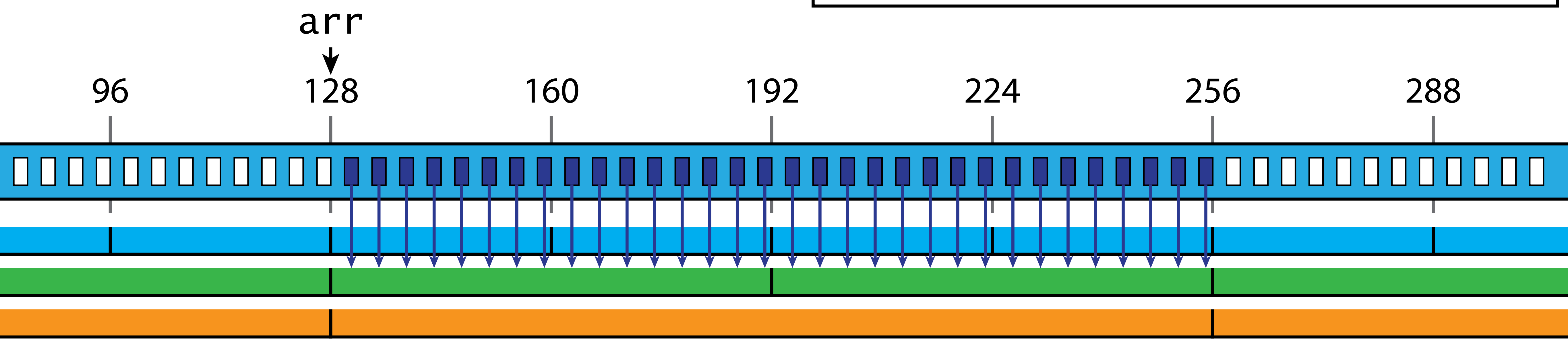
CC 1.0/1.1 CC 1.2/1.3 L1+L2 L2

`arr[threadIdx.x]`

Global memory

对齐且顺序访问时

```
__device__ int arr[128];
...
int v = arr[ threadIdx.x ]
```



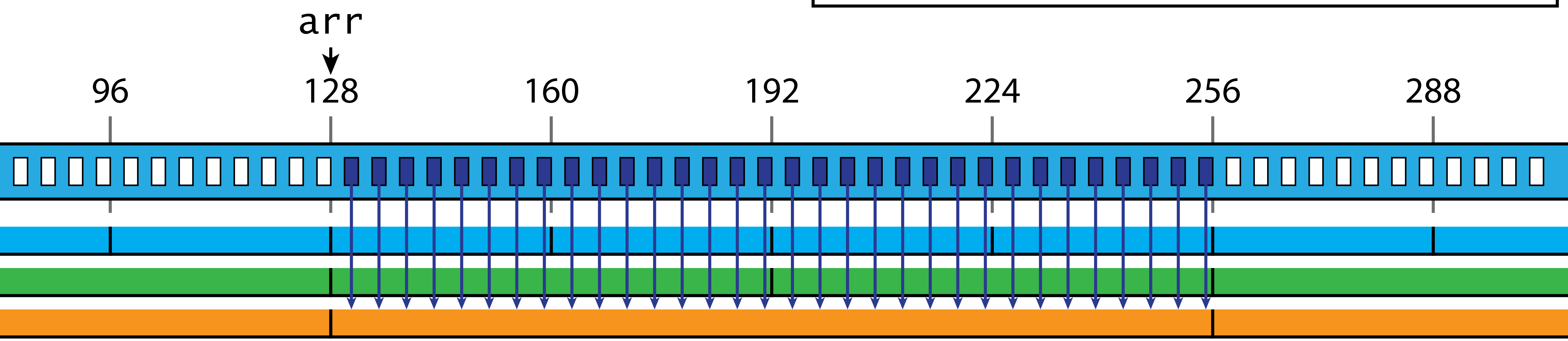
[pre-Fermi] [Fermi] [Kepler]

	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%		

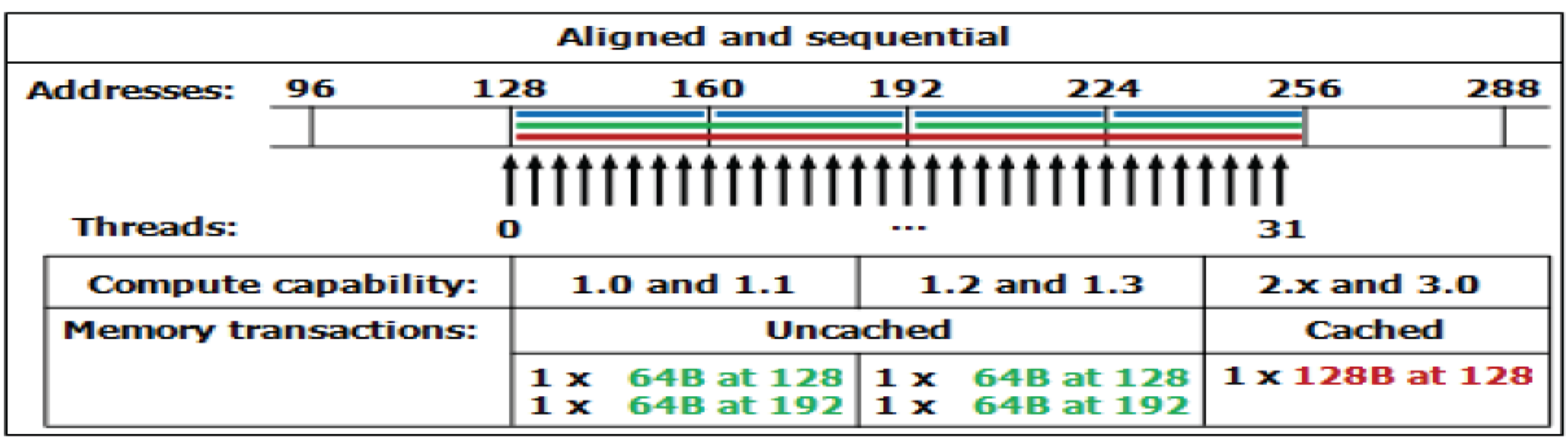
Global memory

对齐且顺序访问时

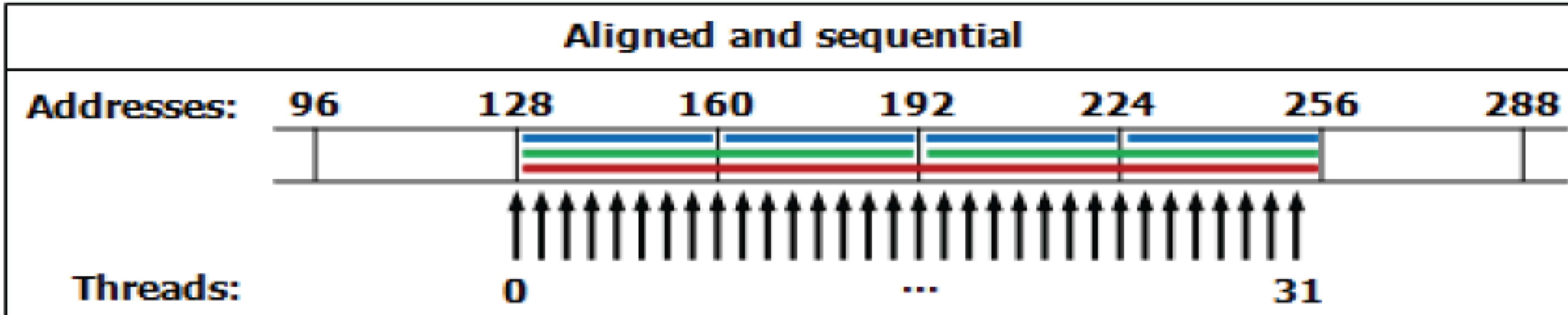
```
__device__ int arr[128];
...
int v = arr[ threadIdx.x ]
```



	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%



一个线程束访问全局存储器的例子，每个线程4字节和相关的基于计算能力的存储器事务



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

访问显存时要遵守严格的合并访问规则

将half warp访问global的起始位置严格的对齐到16的整数倍

在G8x，G9x硬件上thread访问显存的位置必须逐一递增

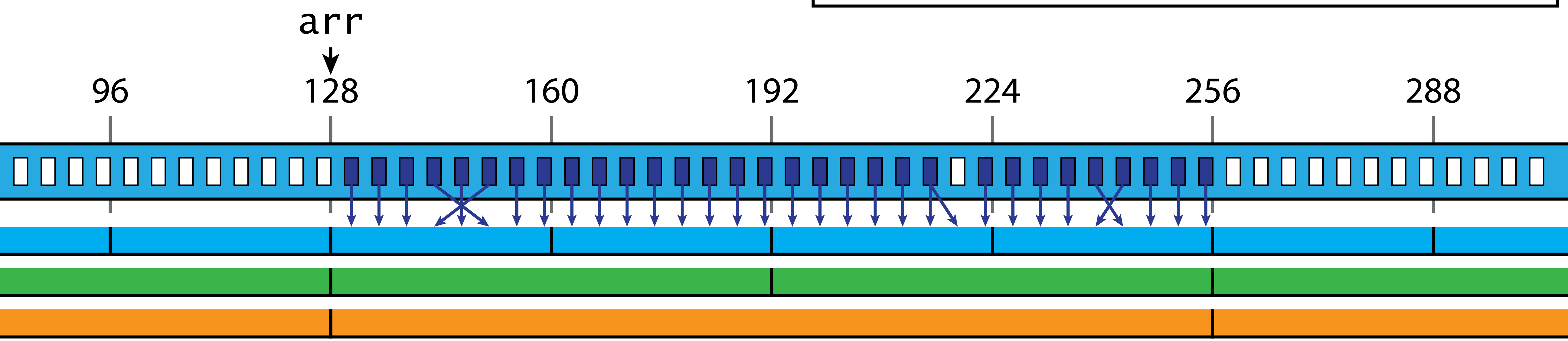
GT200有了很大的改进，对齐和次序比较灵活

好的合并访问可以将存储器访问次数减少十几倍

Global memory

对齐且交叉次序访问时

```
__device__ int arr[128];
...
int v = arr[permute(threadIdx.x)]
```

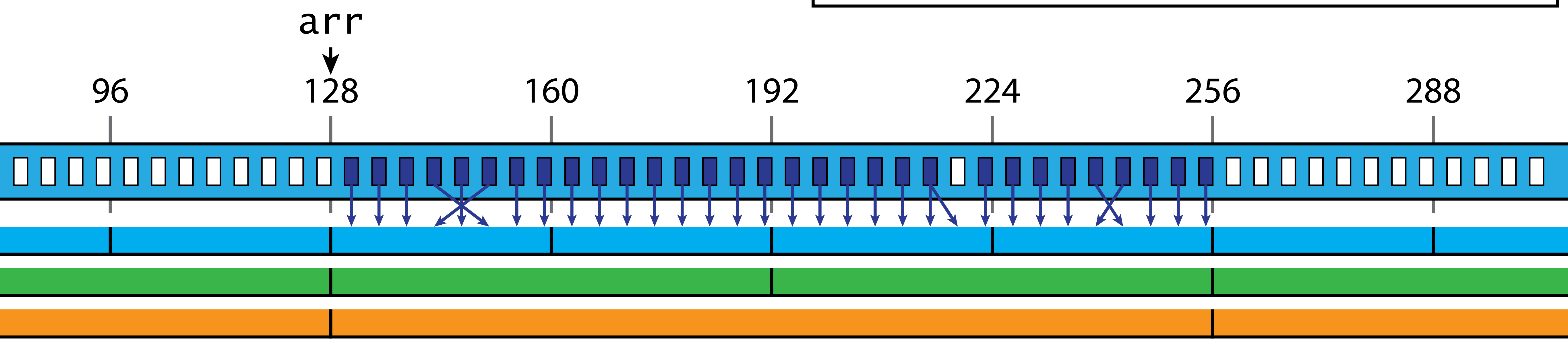


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
交叉 arr[permute(tid)]				

Global memory

对齐且交叉次序访问时

```
__device__ int arr[128];
...
int v = arr[permute(threadIdx.x)]
```

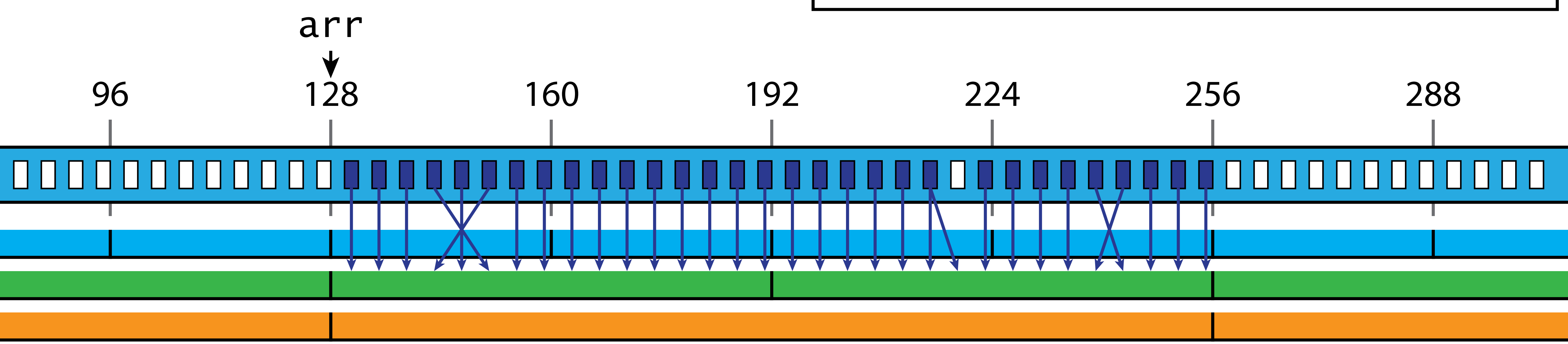


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
交叉 arr[permute(tid)]	32x32B 12%			

Global memory

对齐且交叉次序访问时

```
__device__ int arr[128];
...
int v = arr[permute(threadIdx.x)]
```

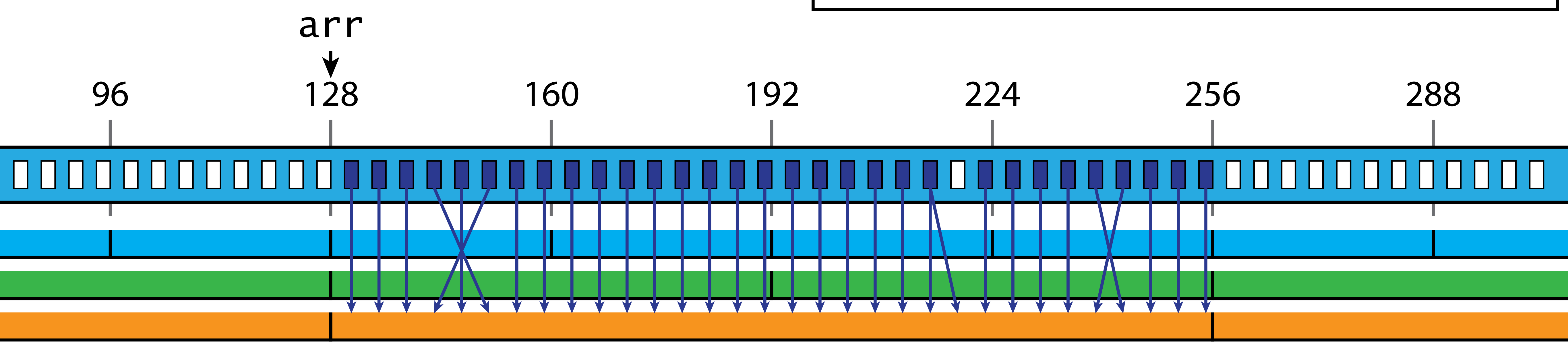


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
交叉 arr[permute(tid)]	32x32B 12%	2x64B 100%		

Global memory

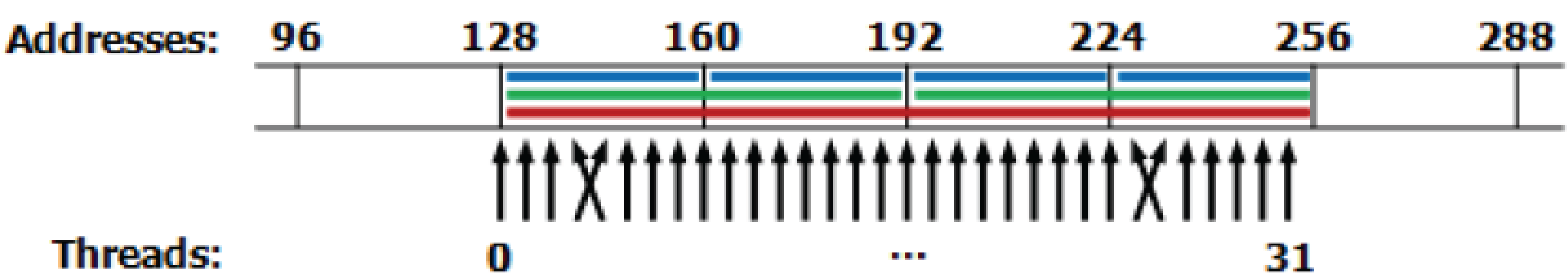
对齐且交叉次序访问时

```
__device__ int arr[128];
...
int v = arr[permute(threadIdx.x)]
```



	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
`arr[threadIdx.x]`	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
`arr[permute(tid)]`	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%

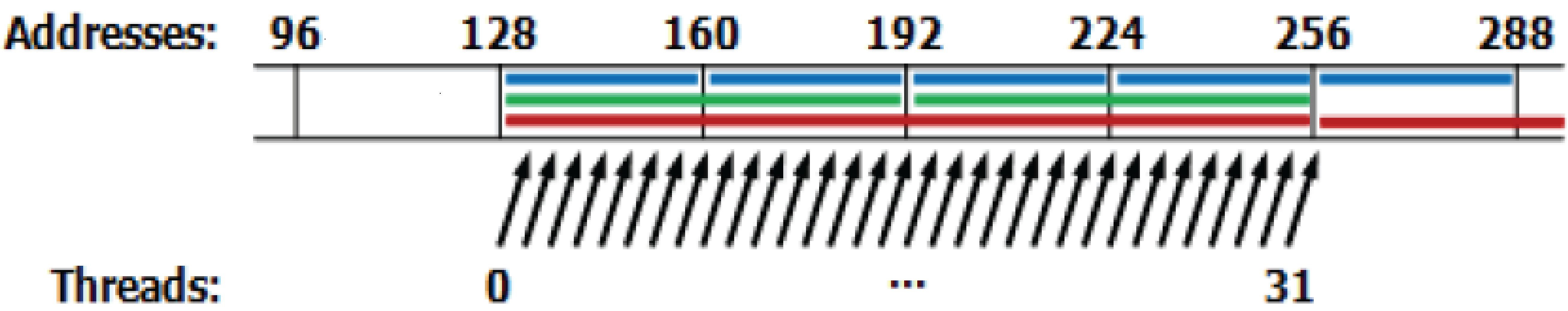
Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

一个线程束访问全局存储器的例子，每个线程4字节和相关的基于计算能力的存储器事务

Misaligned and sequential

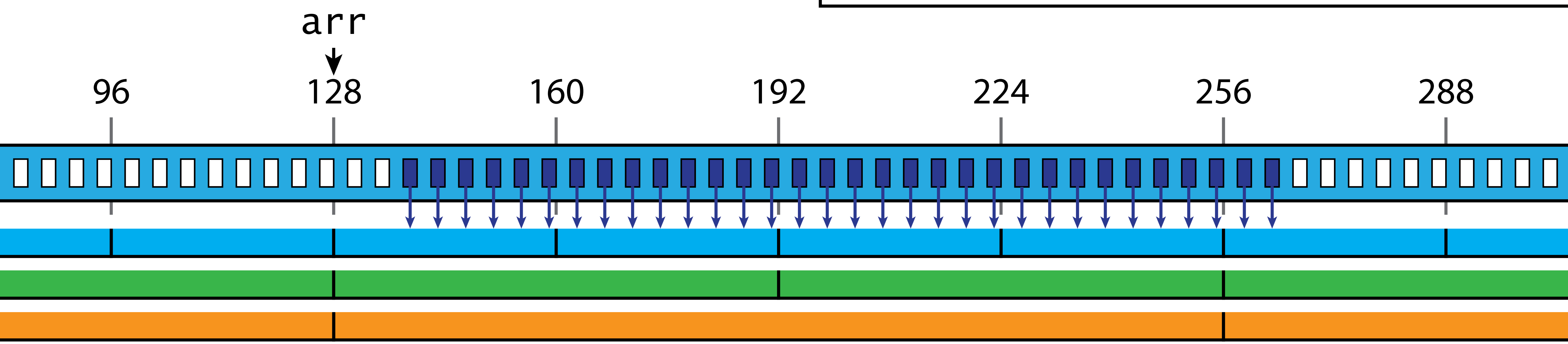


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Global memory

未对齐但顺序访问时

```
__device__ int arr[128];
...
int v = arr[threadIdx.x + shift]
```

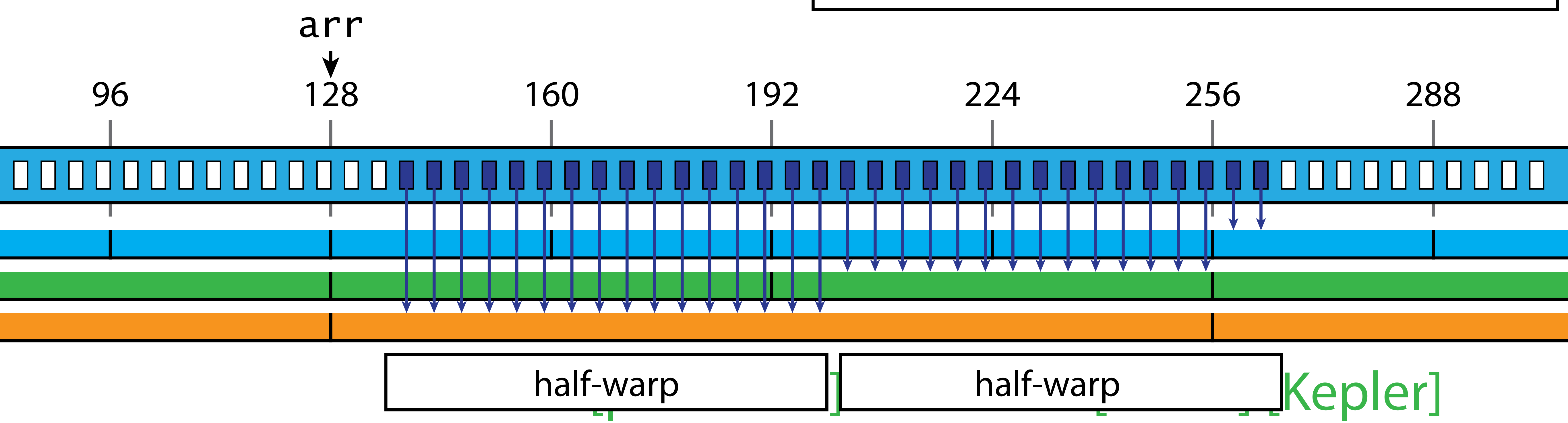


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
`arr[threadIdx.x]`	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
`arr[permute(tid)]`	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
`arr[tid+shift]`	32x32B 12%			

Global memory

未对齐但顺序访问时

```
__device__ int arr[128];
...
int v = arr[threadIdx.x + shift]
```

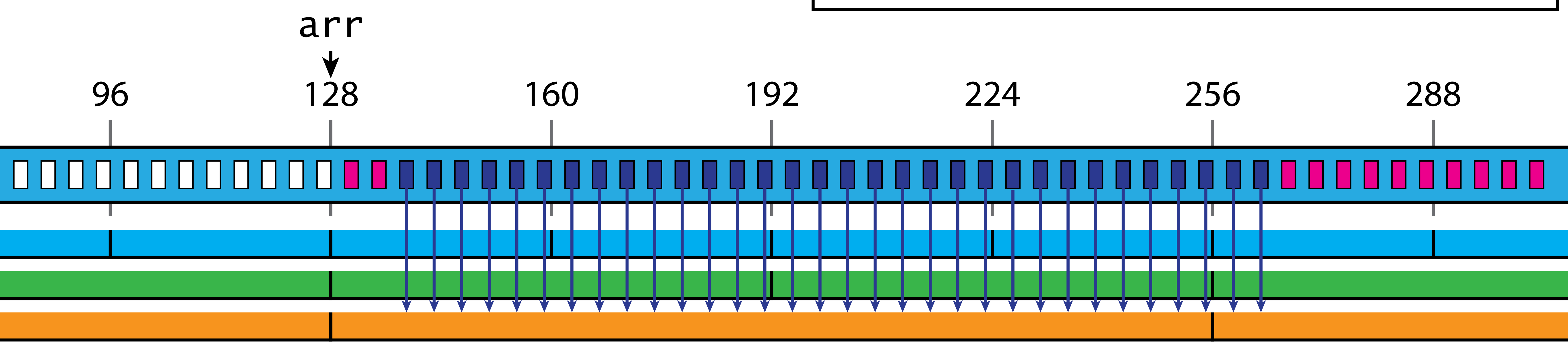


	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
`arr[threadIdx.x]`	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
`arr[permute(tid)]`	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
`arr[tid+shift]`	32x32B 12%	128B+64B+32B 57%		

Global memory

未对齐但顺序访问时

```
__device__ int arr[128];
...
int v = arr[threadIdx.x + shift]
```

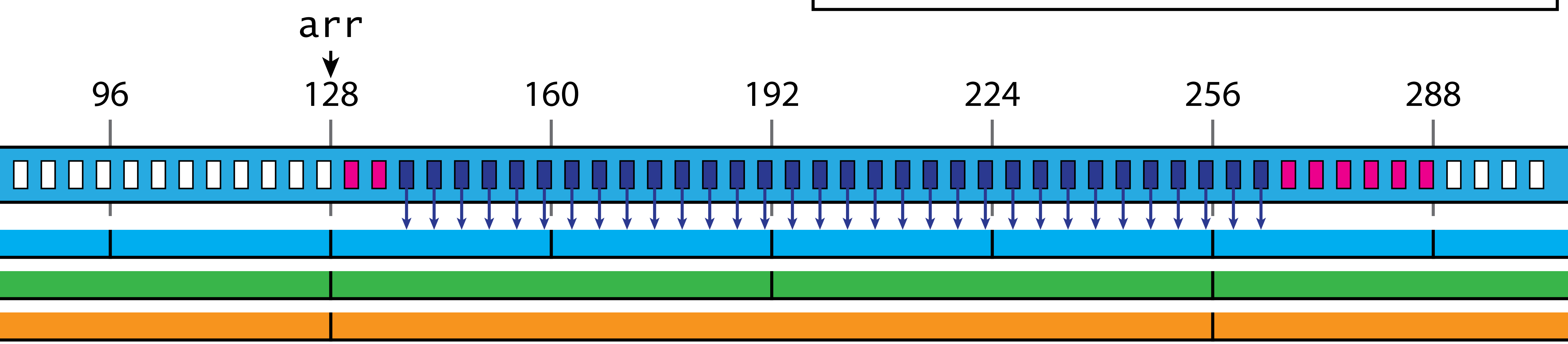


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
arr[permute(tid)]	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
arr[tid+shift]	32x32B 12%	128B+64B+32B 57%	2x128B 50%	

Global memory

未对齐但顺序访问时

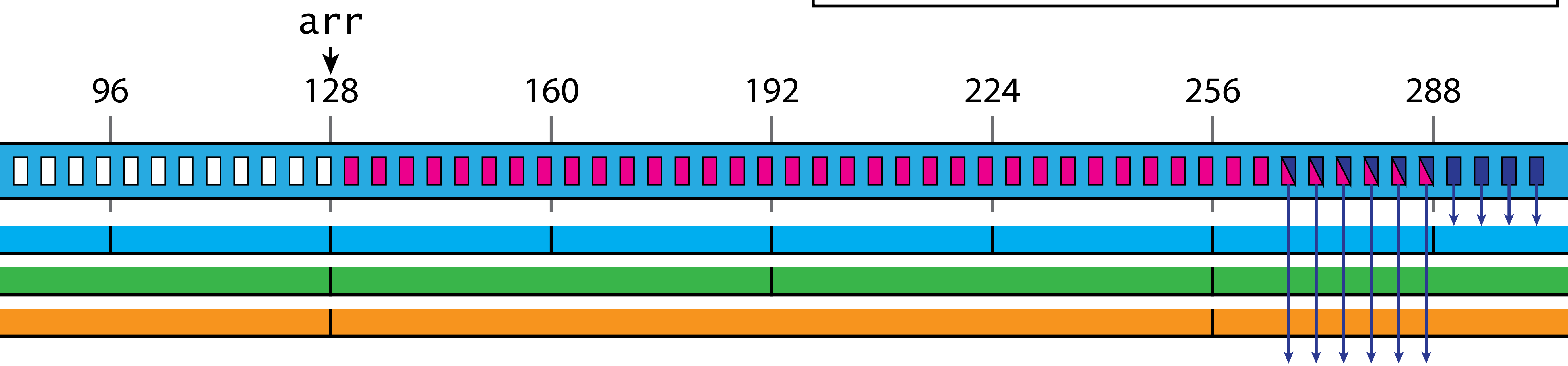
```
__device__ int arr[128];
...
int v = arr[threadIdx.x + shift]
```



	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
arr[permute(tid)]	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
arr[tid+shift]	32x32B 12%	128B+64B+32B 57%	2x128B 50%	5x32B 80%

Global memory

```
__device__ int arr[128];
...
int v = arr[threadIdx.x + shift]
```



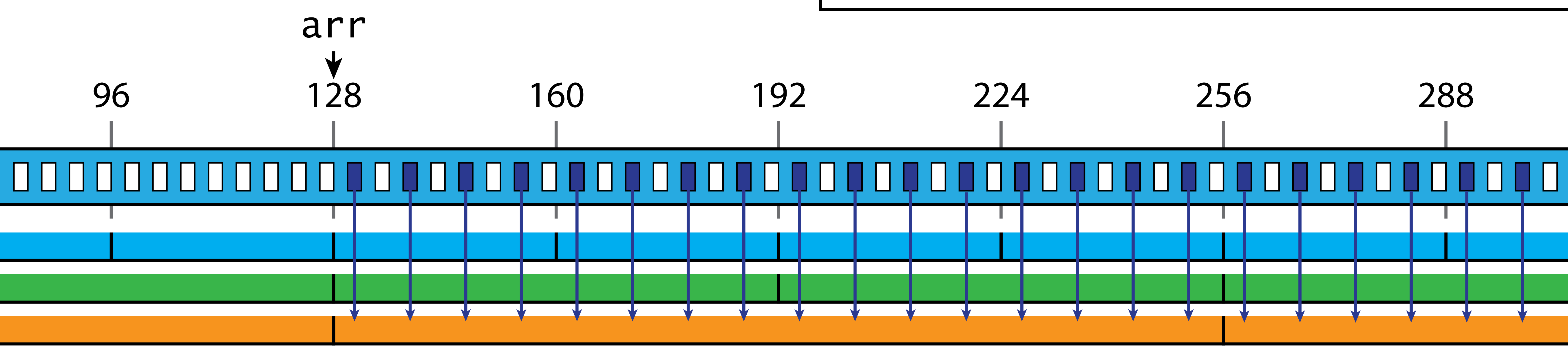
	[pre-Fermi]		[Fermi]	[Kepler]
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
`arr[threadIdx.x]`	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
`arr[permute(tid)]`	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
`arr[tid+shift]`	32x32B 12%	128B+64B+32B 57%	2x128B 50% ⇝ 89%	5x32B 80% ⇝ 97%

8 warps, cache used

Global memory

跳跃间隔访问时（隔一个）

```
__device__ int arr[128];
...
int v = arr[stride*threadIdx.x]
```

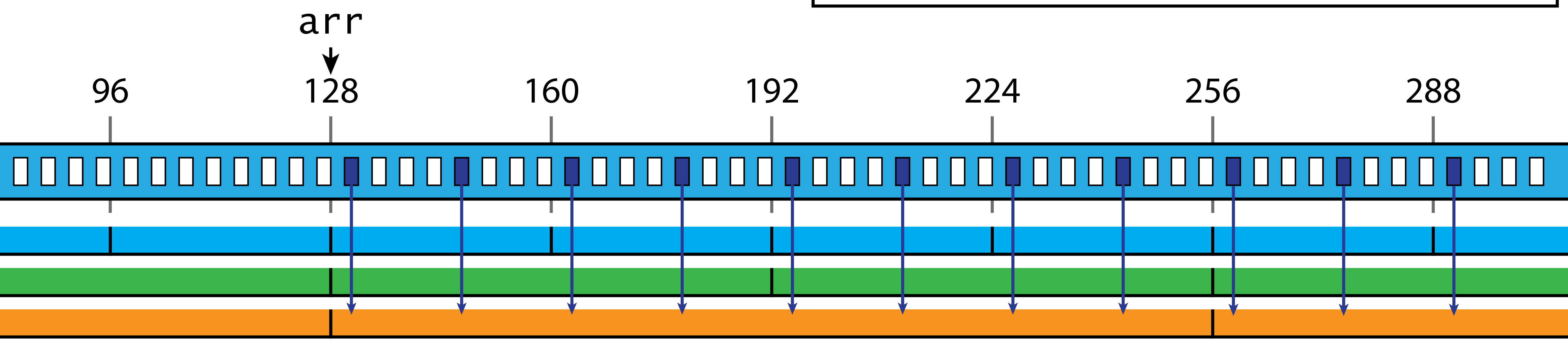


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
arr[permute(tid)]	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
arr[tid+shift]	32x32B 12%	128B+64B+32B 57%	2x128B 50% ⇝ 89%	5x32B 80% ⇝ 97%
arr[2*tid]	2x128B 50%	2x128B 50%	2x128B 50% ⇝ 50%	8x32B 50% ⇝ 50%

Global memory

跳跃间隔访问时（隔3个）

```
__device__ int arr[128];
...
int v = arr[stride*threadIdx.x]
```

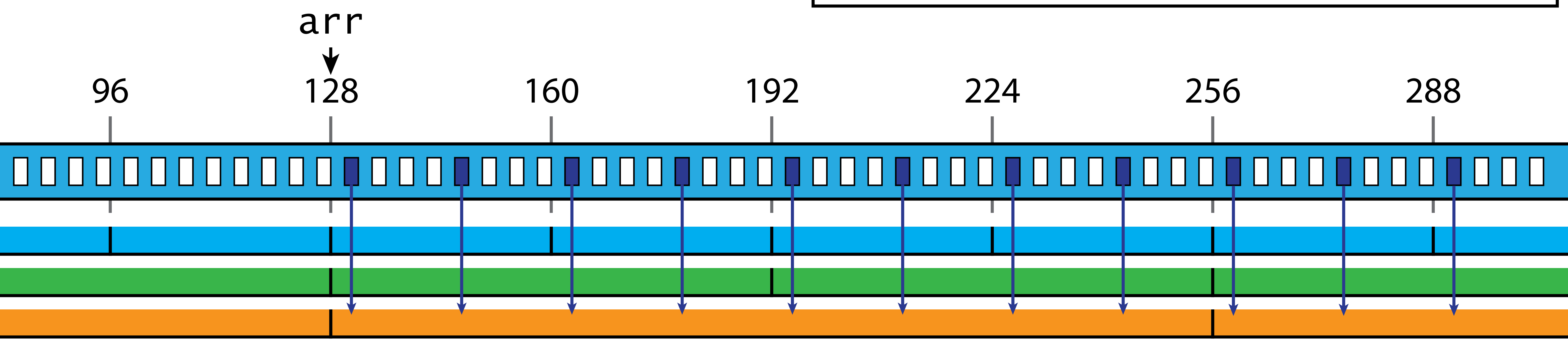


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
arr[permute(tid)]	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
arr[tid+shift]	32x32B 12%	128B+64B+32B 57%	2x128B 50% → 89%	5x32B 80% → 97%
arr[4*tid]	4x128B 25%	4x128B 25%	4x128B 25% → 25%	16x32B 25% → 25%

Global memory

由于int4是4个元素，故与乘4一样

```
__device__ int4 arr[128];
...
int v = arr[threadIdx.x].x;
```

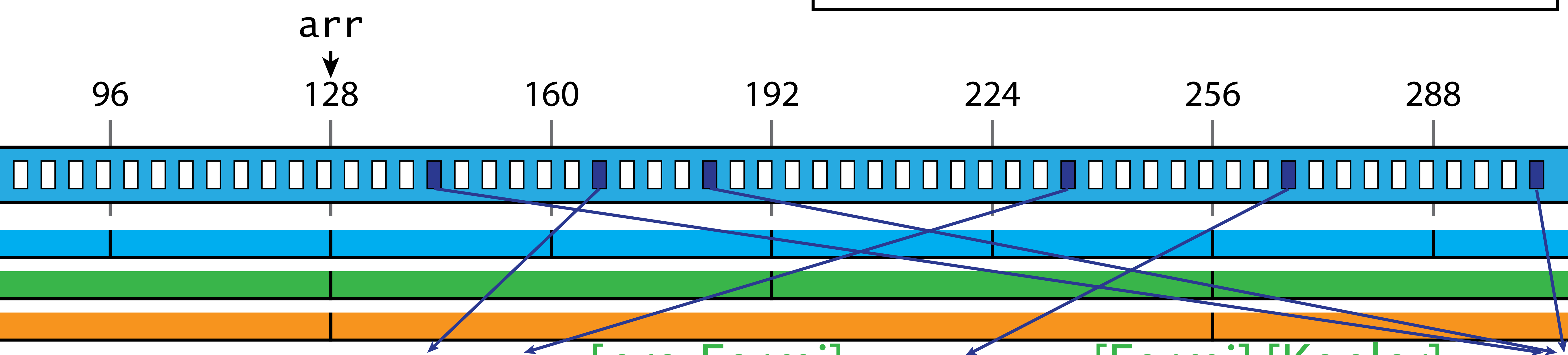


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
arr[permute(tid)]	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
arr[tid+shift]	32x32B 12%	128B+64B+32B 57%	2x128B 50% → 89%	5x32B 80% → 97%
arr[4*tid]	4x128B 25%	4x128B 25%	4x128B 25% → 25%	16x32B 25% → 25%

Global memory

随机访问时

```
__device__ int arr[128];
...
int v = arr[ random ]
```



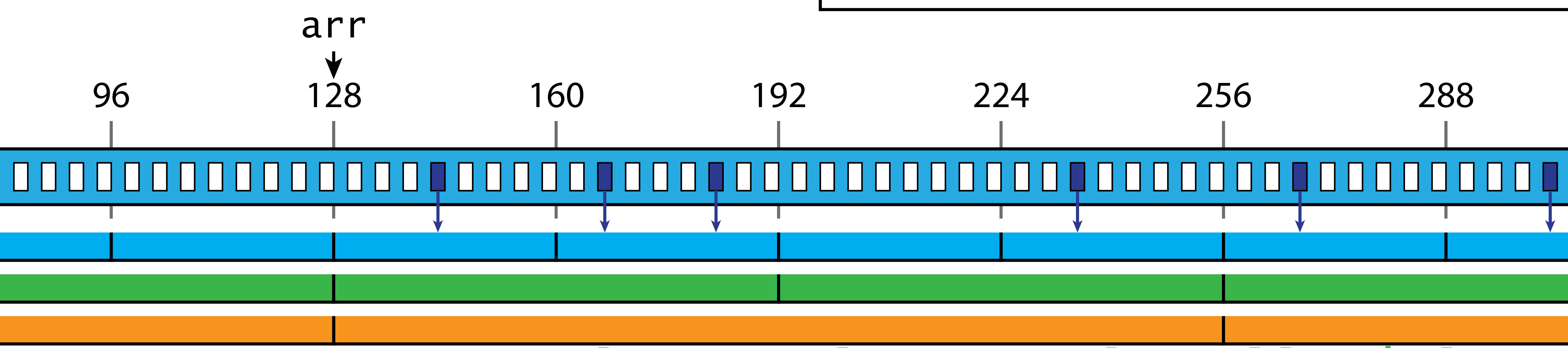
[pre-Fermi] [Fermi] [Kepler]

	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
arr[permute(tid)]	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
arr[tid+shift]	32x32B 12%	128B+64B+32B 57%	2x128B 50% ⇝ 89%	5x32B 80% ⇝ 97%
arr[4*tid]	4x128B 25%	4x128B 25%	4x128B 25% ⇝ 25%	16x32B 25% ⇝ 25%
arr[random]				

Global memory

随机访问时

```
__device__ int arr[128];
...
int v = arr[ random ]
```

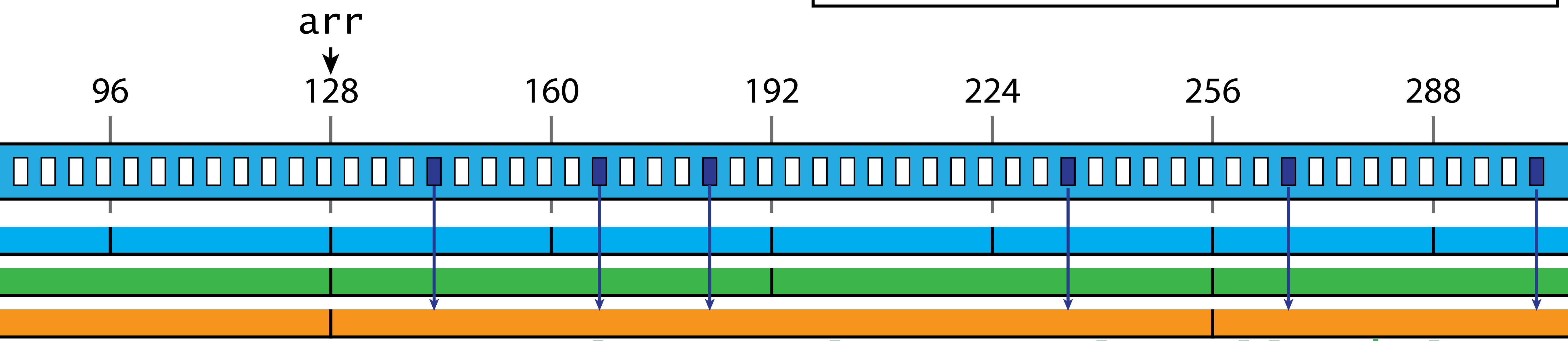


	[pre-Fermi]		[Fermi] [Kepler]	
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
arr[permute(tid)]	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
arr[tid+shift]	32x32B 12%	128B+64B+32B 57%	2x128B 50% ⇝ 89%	5x32B 80% ⇝ 97%
arr[4*tid]	4x128B 25%	4x128B 25%	4x128B 25% ⇝ 25%	16x32B 25% ⇝ 25%
arr[random]	32x32B 12%	32x32B 12%		32x32B 12% ⇝ 12%

Global memory

随机访问时

```
__device__ int arr[128];
...
int v = arr[ random ]
```



	[pre-Fermi]		[Fermi]	[Kepler]
	CC 1.0/1.1	CC 1.2/1.3	L1+L2	L2
arr[threadIdx.x]	2x64B 100%	2x64B 100%	1x128B 100%	4x32B 100%
arr[permute(tid)]	32x32B 12%	2x64B 100%	1x128B 100%	4x32B 100%
arr[tid+shift]	32x32B 12%	128B+64B+32B 57%	2x128B 50% → 89%	5x32B 80% → 97%
arr[4*tid]	4x128B 25%	4x128B 25%	4x128B 25% → 25%	16x32B 25% → 25%
arr[random]	32x32B 12%	32x32B 12%	32x128B 3% → 3%	32x32B 12% → 12%

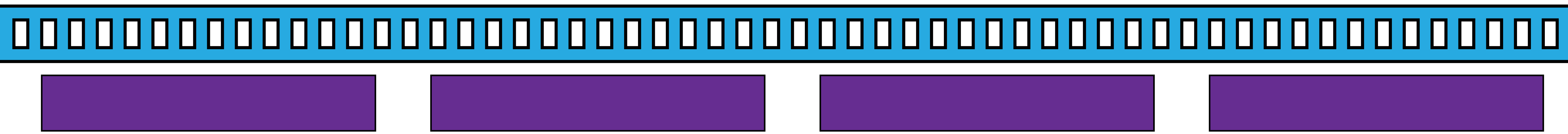
Constant memory

- off-chip (global memory)
- linearly addressable
- read-only for SM
- cached in SM

如global memory一样属于片外存储器

其速度比shared 要慢，但是它具有缓存，

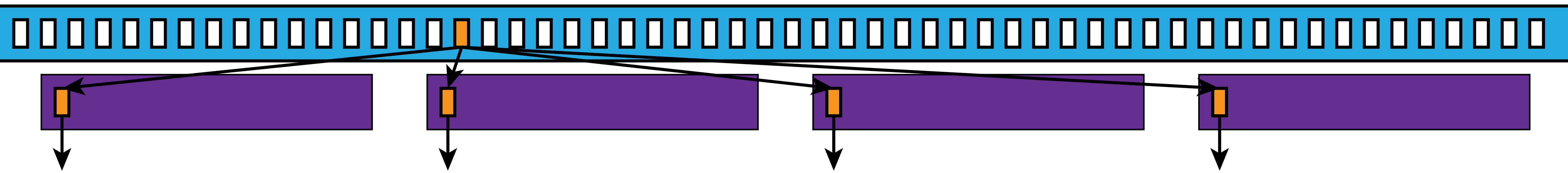
并且无须考虑冲突问题，主要用来加速对常数的访问。



CUDA 允许分配最多64ＫＢ的常量存储器，常量存储器顾名思义内容是不变的，所以也有人称其为不变存储器。每个ＳＭ有６－８ＫＢ的常量缓存，一般而言一到两个周期可读取常量存储器。如果half-warp内的线程访问的不是同一个地址，那么各个线程的访问将会串行化。

Constant memory

- off-chip (global memory)
- linearly addressable
- read-only for SM
- cached in SM



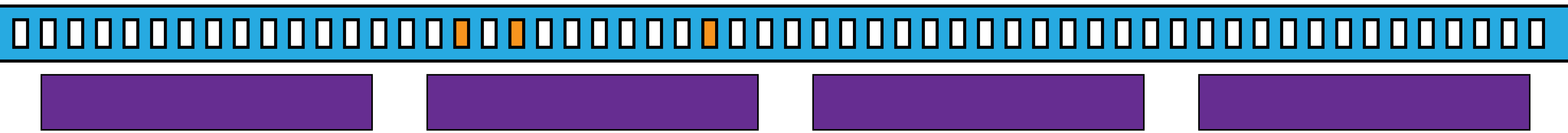
每个block中的warp内的线程访问的是同一个地址，OK。

**如果half-warp内的线程访问的不是同一个地址，那么各个线程的访问将会串行化。

Constant memory

- off-chip (global memory)
- linearly addressable
- read-only for SM
- cached in SM

如果half-warp内的线程访问的不是同一个地址，
那么各个线程的访问将会串行化。



n addresses = n memory transactions

```
__constant__ int arr[128];
__constant__ int c;
...
int v = c;
v=arr[42];
v=arr[blockIdx.x];
v=arr[threadIdx.x/32];
v=arr[threadIdx.x];
```

所有的线程访问的是同一个地址

所有的线程访问的是同一个地址

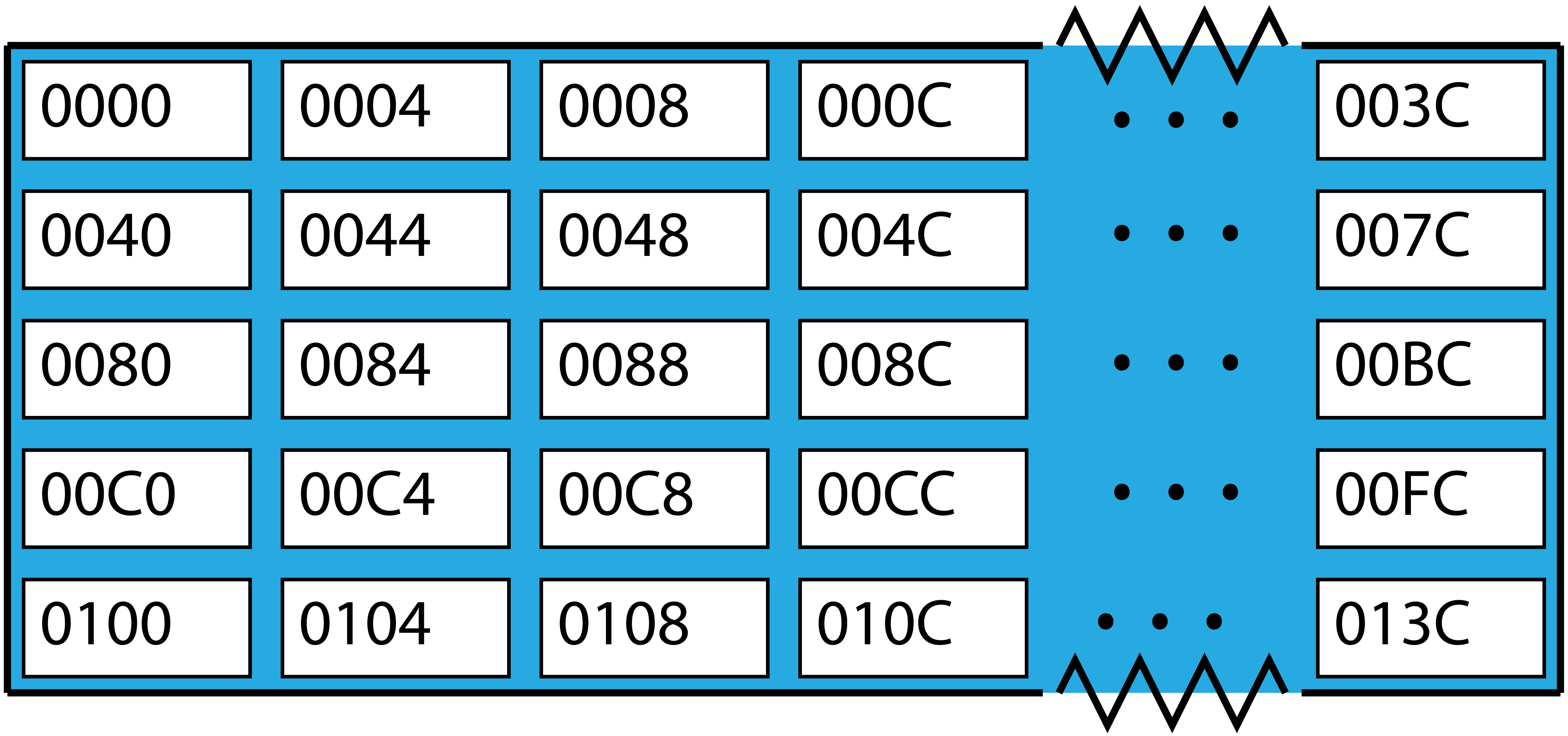
每个block仅一个线程访问一个地址

每个warp仅一个线程访问一个地址

每个warp中不同线程访问不同地址，串行

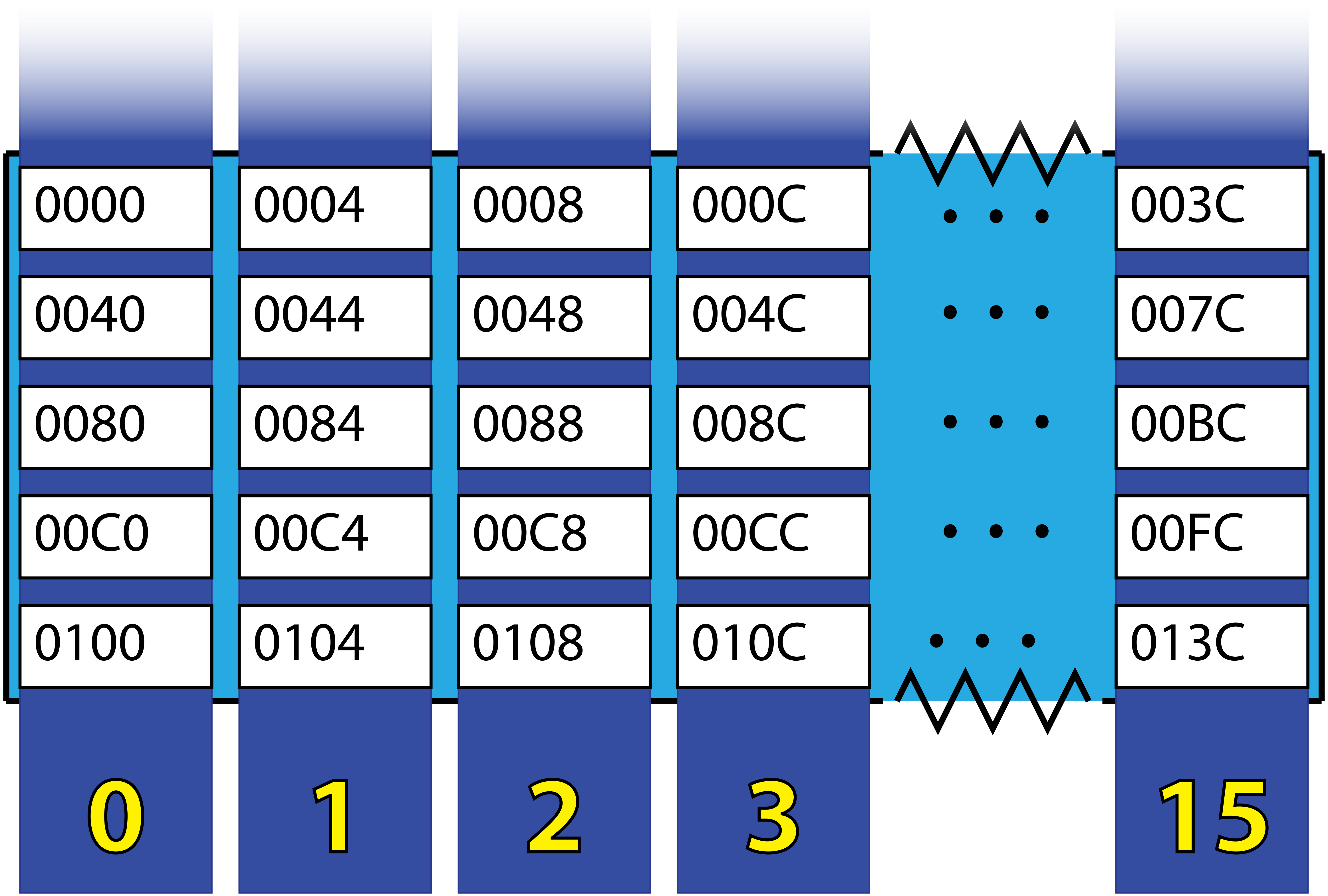
Shared memory

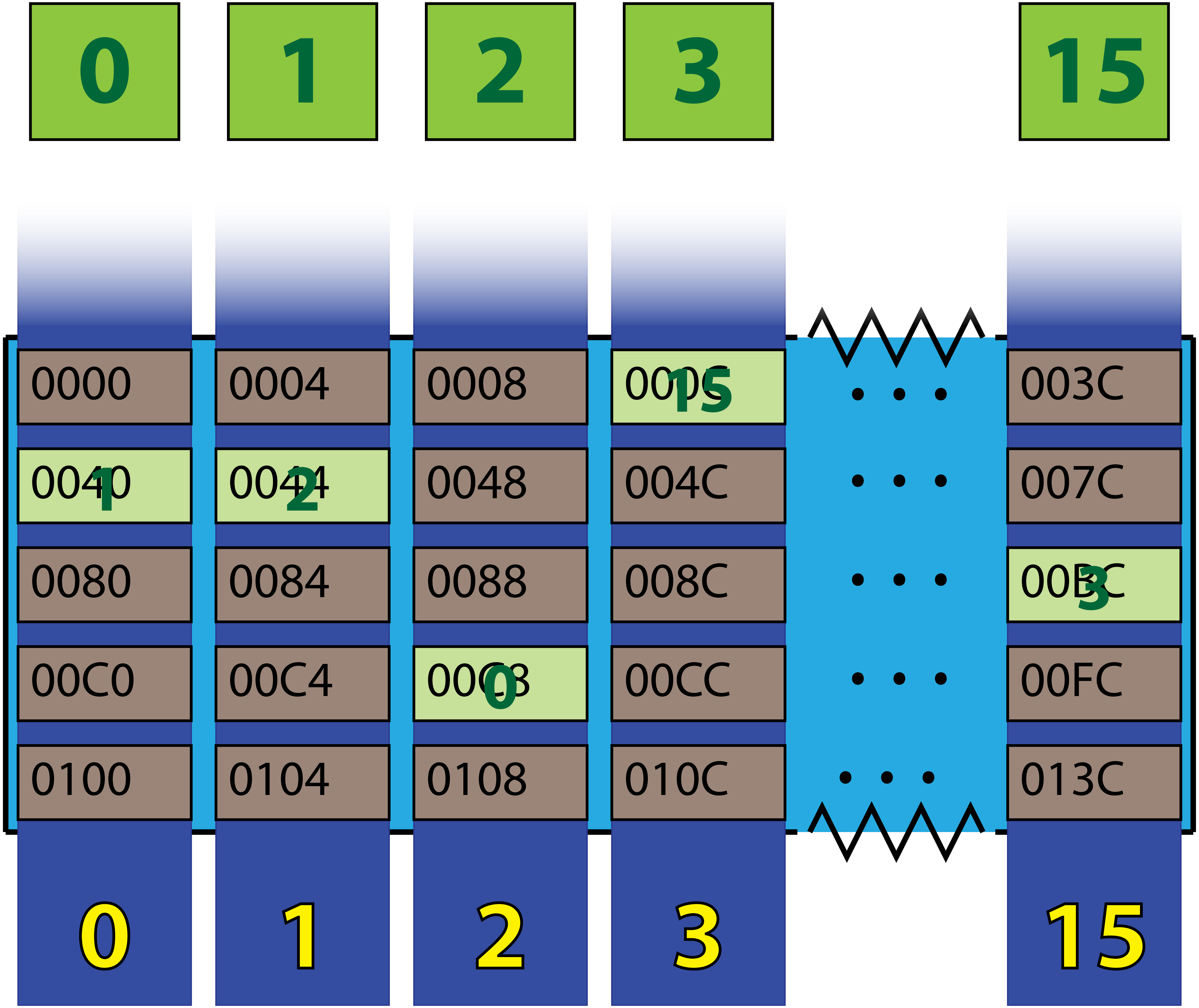
- on-chip
- linearly addressable

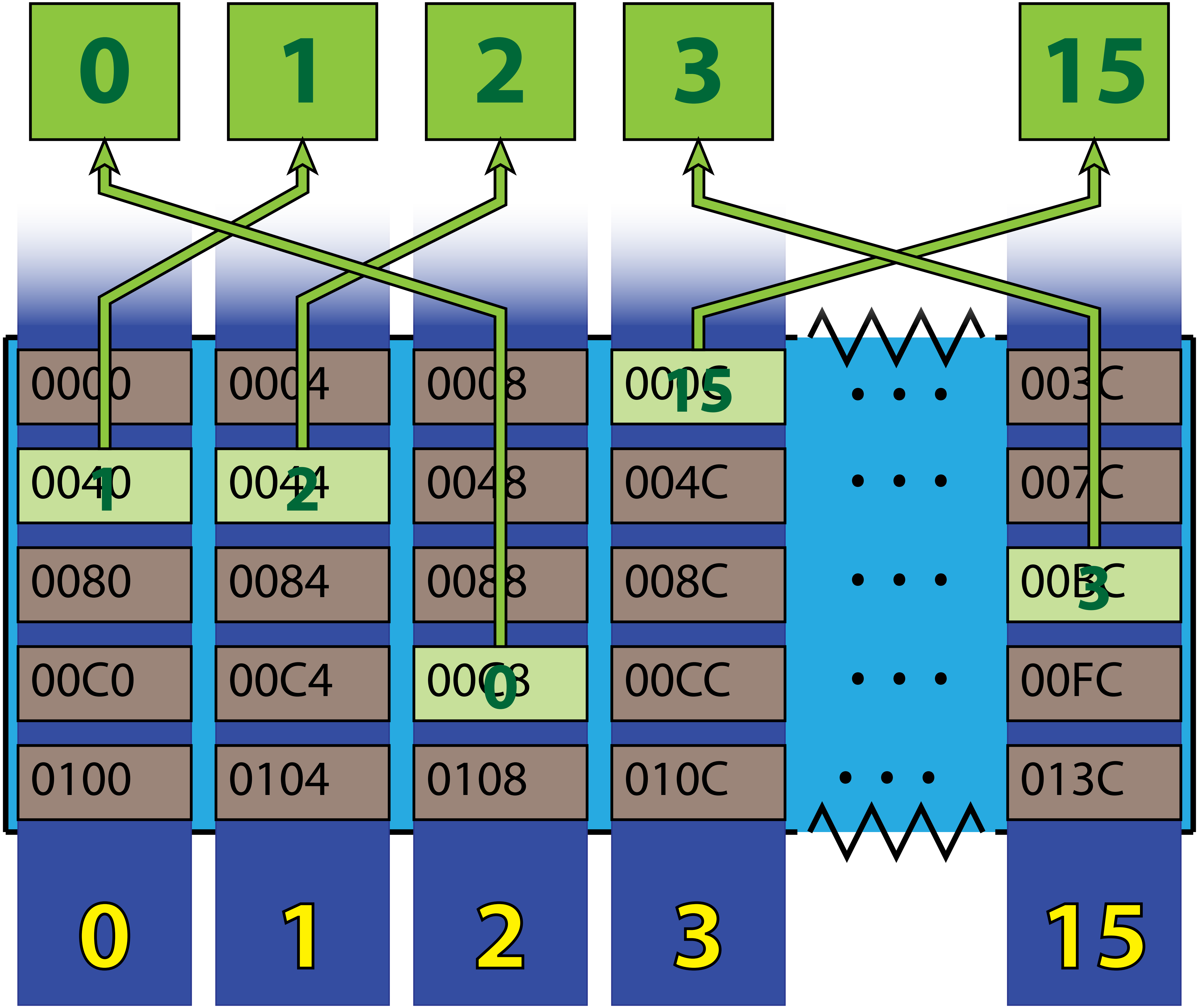


Shared memory

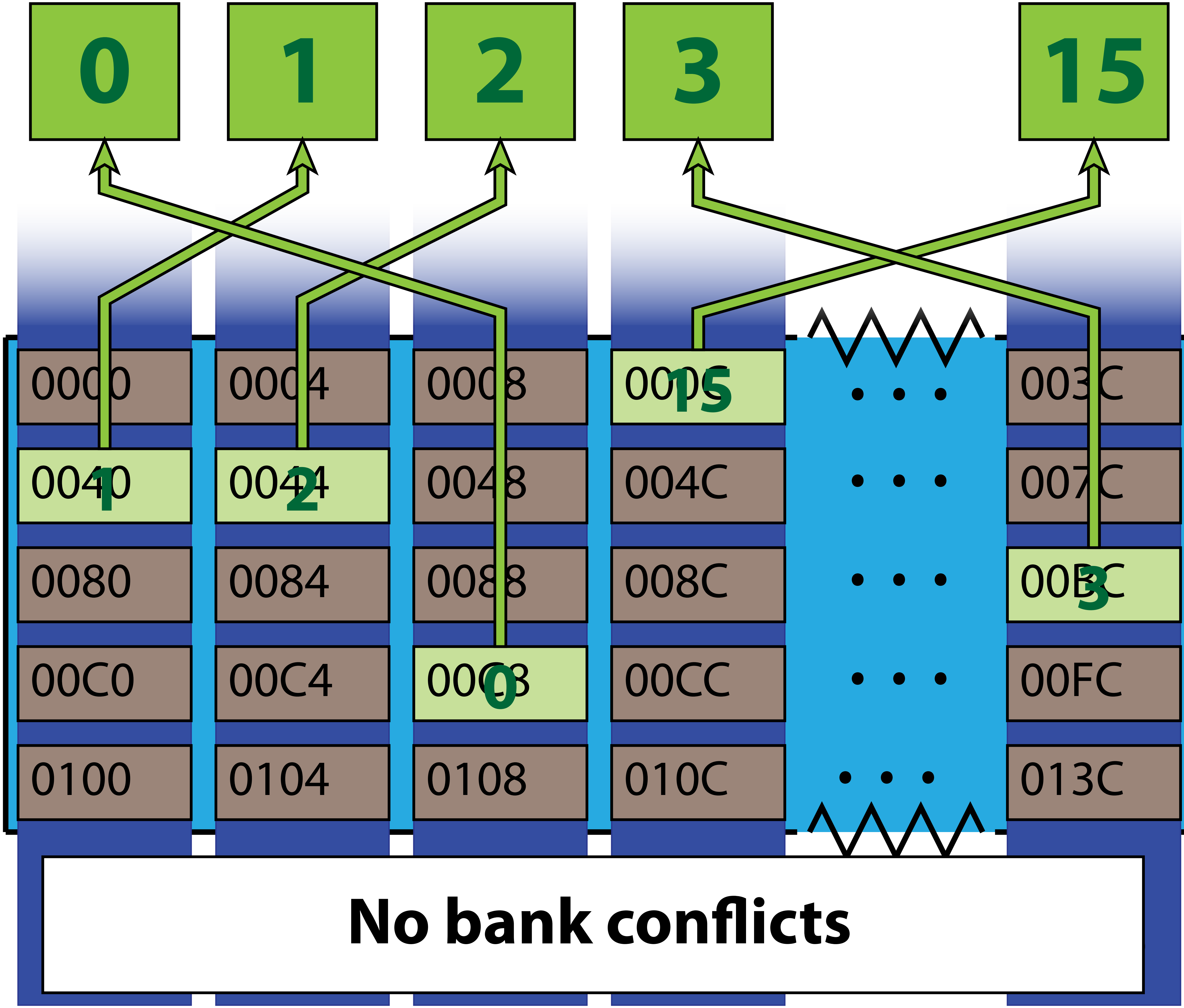
- memory banks

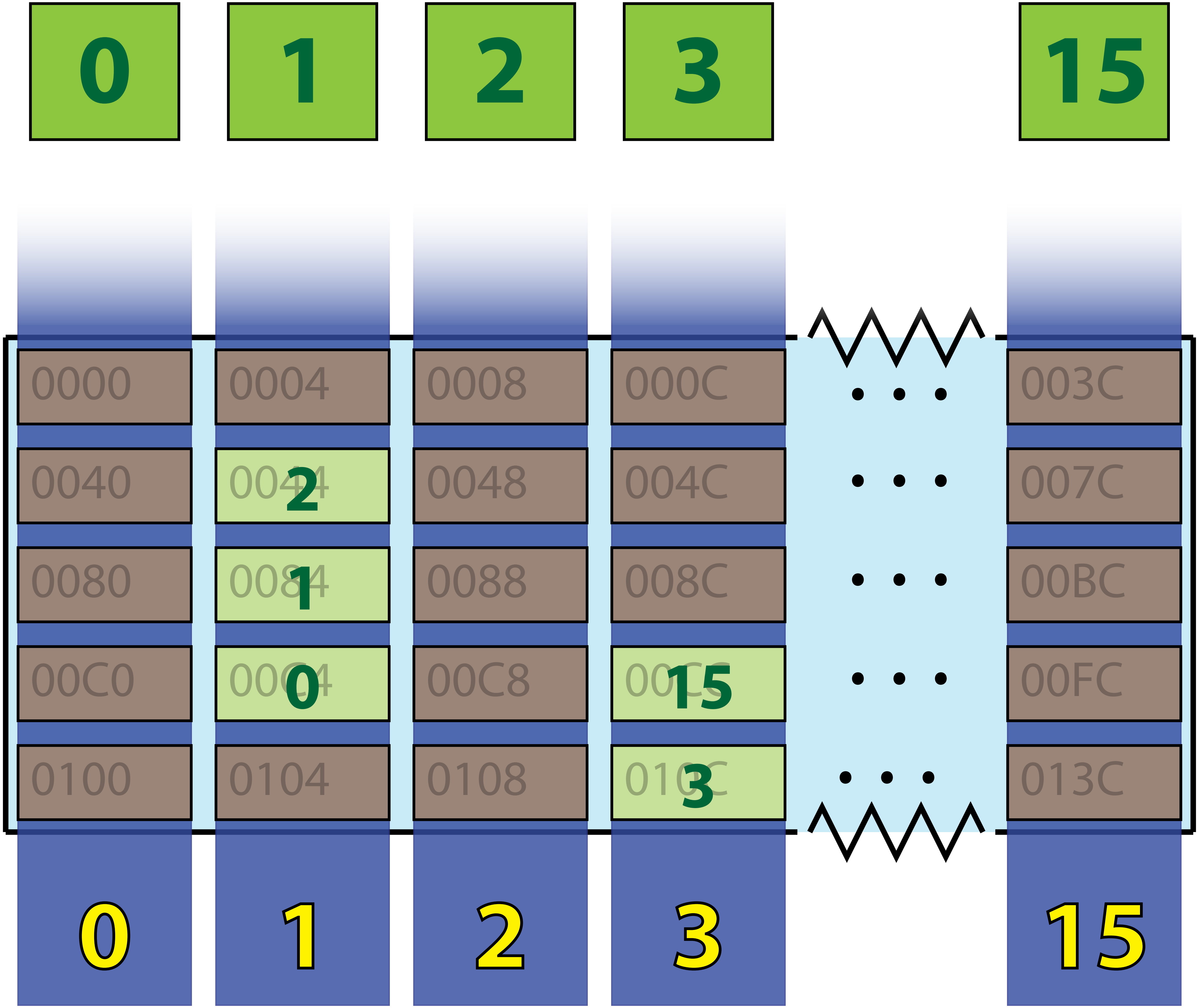



0
1
2
3
15
0000
0004
0008
000C
003C
0040
0044
0048
004C
007C
0080
0084
0088
008C
00BC
00C0
00C4
00C8
00CC
00FC
0100
0104
0108
010C
013C
15
1
2
3
0
0
1
2
3
15

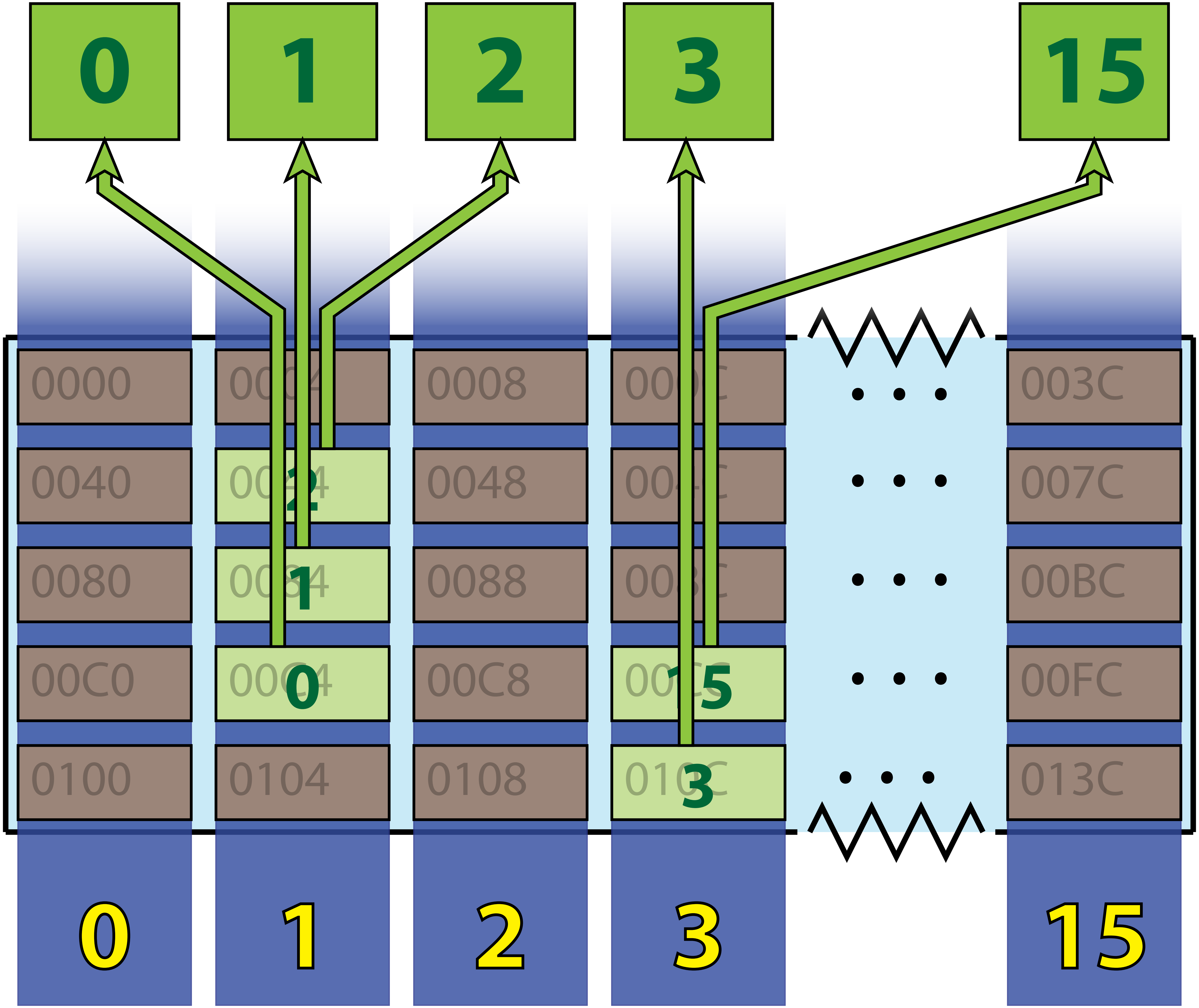


0
1
2
3
15
0000
0004
0008
000C
15
003C
0040
1
0044
2
0048
004C
007C
0080
0084
0088
008C
00BC
3
00C0
00C4
00C8
0
00CC
00FC
0100
0104
0108
010C
013C
0
1
2
3
15

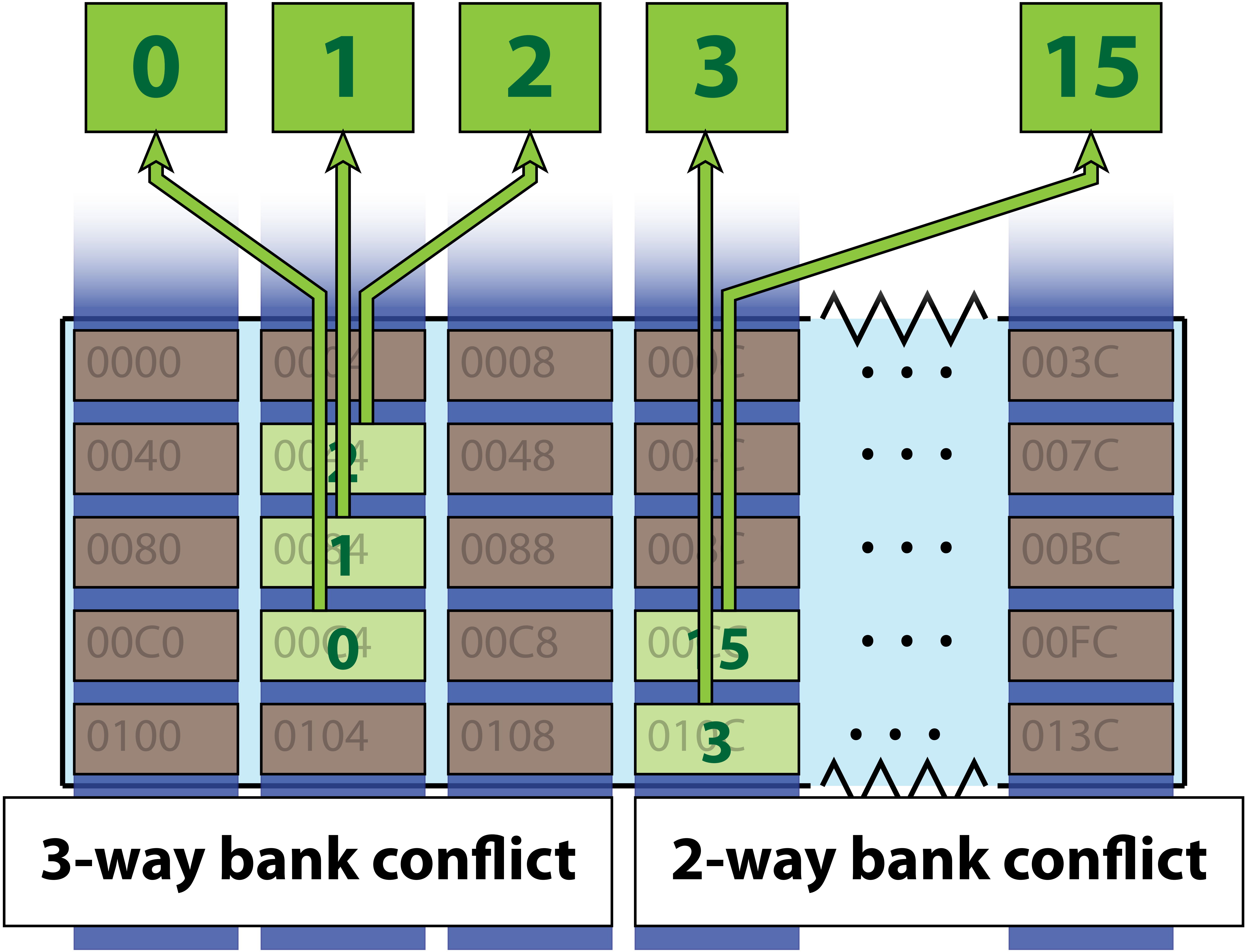

0
1
2
3
15
0000
0004
0008
0010
15
003C
0040
1
0044
2
0048
004C
007C
0080
0084
0088
008C
00BC
3
00C0
00C4
00C8
0
00CC
00FC
0100
0104
0108
010C
013C
No bank conflicts



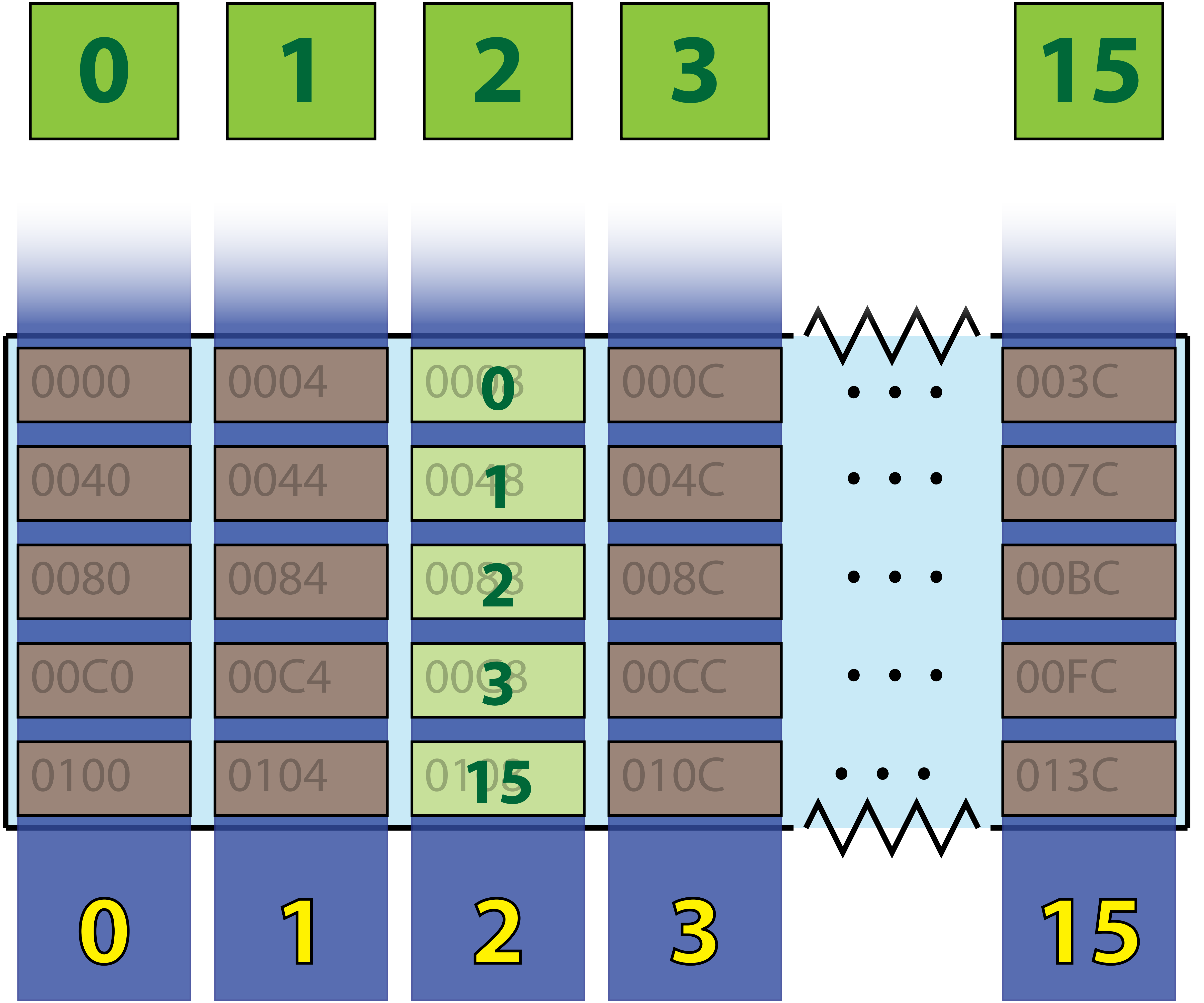
0
1
2
3
15
0000
0004
0008
000C
003C
0040
0044
2
0048
004C
007C
0080
0084
1
0088
008C
00BC
00C0
00C4
0
00C8
00CC
15
00FC
0100
0104
0108
013C
3
013C
0
1
2
3
15



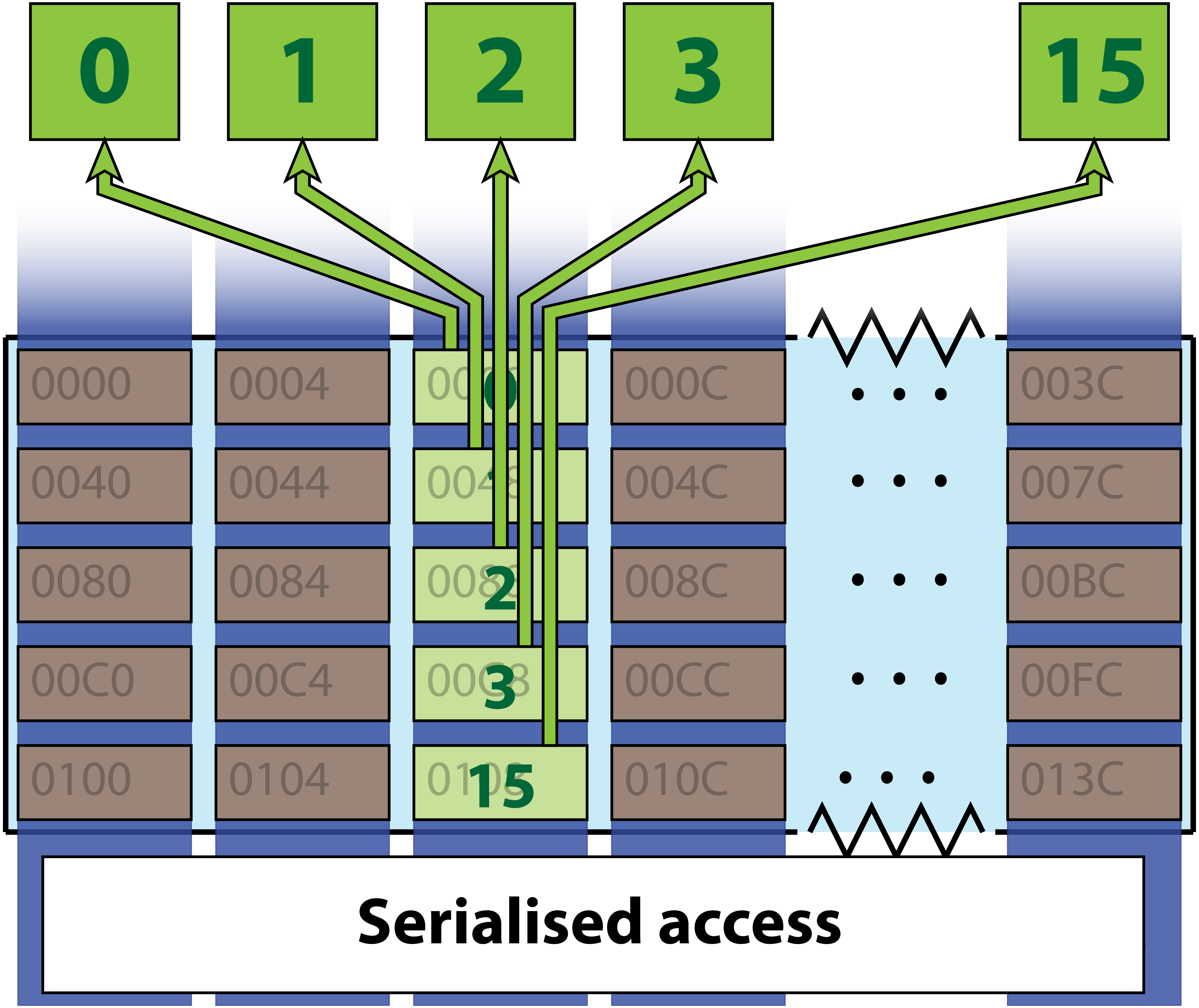
0
1
2
3
15
0000
0008
003C
0040
0048
007C
0080
0088
00BC
00C0
00C8
00FC
0100
0104
0108
013C

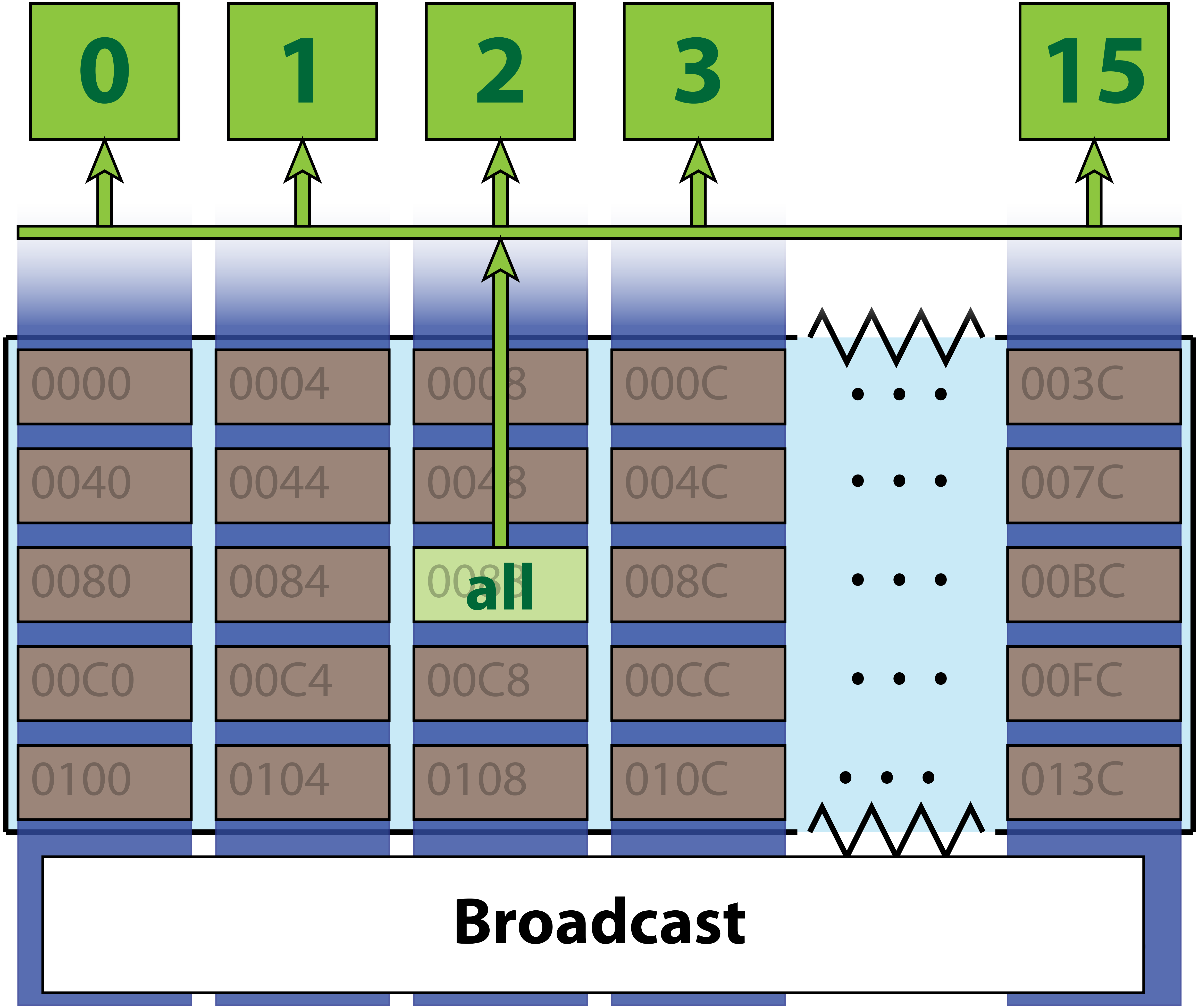


0
1
2
3
15
0000
0008
003C
0040
0048
007C
0080
0088
00BC
00C0
00C8
00FC
0100
0104
0108
013C
3-way bank conflict
2-way bank conflict



0
1
2
3
15
0000
0004
0008
000C
003C
0040
0044
0048
004C
007C
0080
0084
0088
008C
00BC
00C0
00C4
00C8
00CC
00FC
0100
0104
0108
010C
013C
0
1
2
3
15


0
1
2
3
15
0000
0004
000C
003C
0040
0044
004C
007C
0080
0084
008C
00BC
00C0
00C4
00CC
00FC
0100
0104
010C
013C
Serialised access

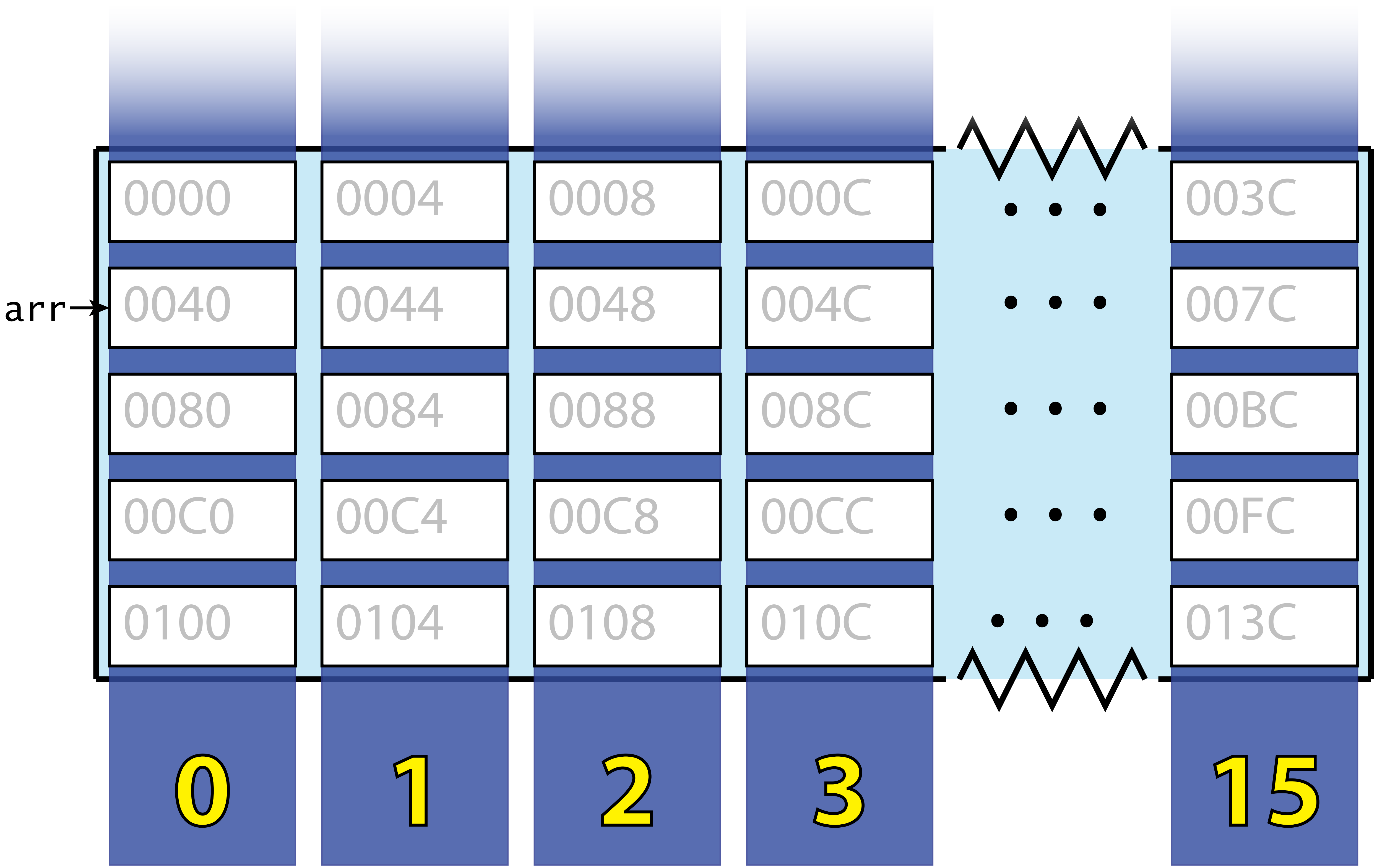


0
1
2
3
15
0000
0004
0008
000C
003C
0040
0044
0048
004C
007C
0080
0084
0088
all
008C
00BC
00C0
00C4
00C8
00CC
00FC
0100
0104
0108
010C
013C
Broadcast

Shared memory

- in practice?

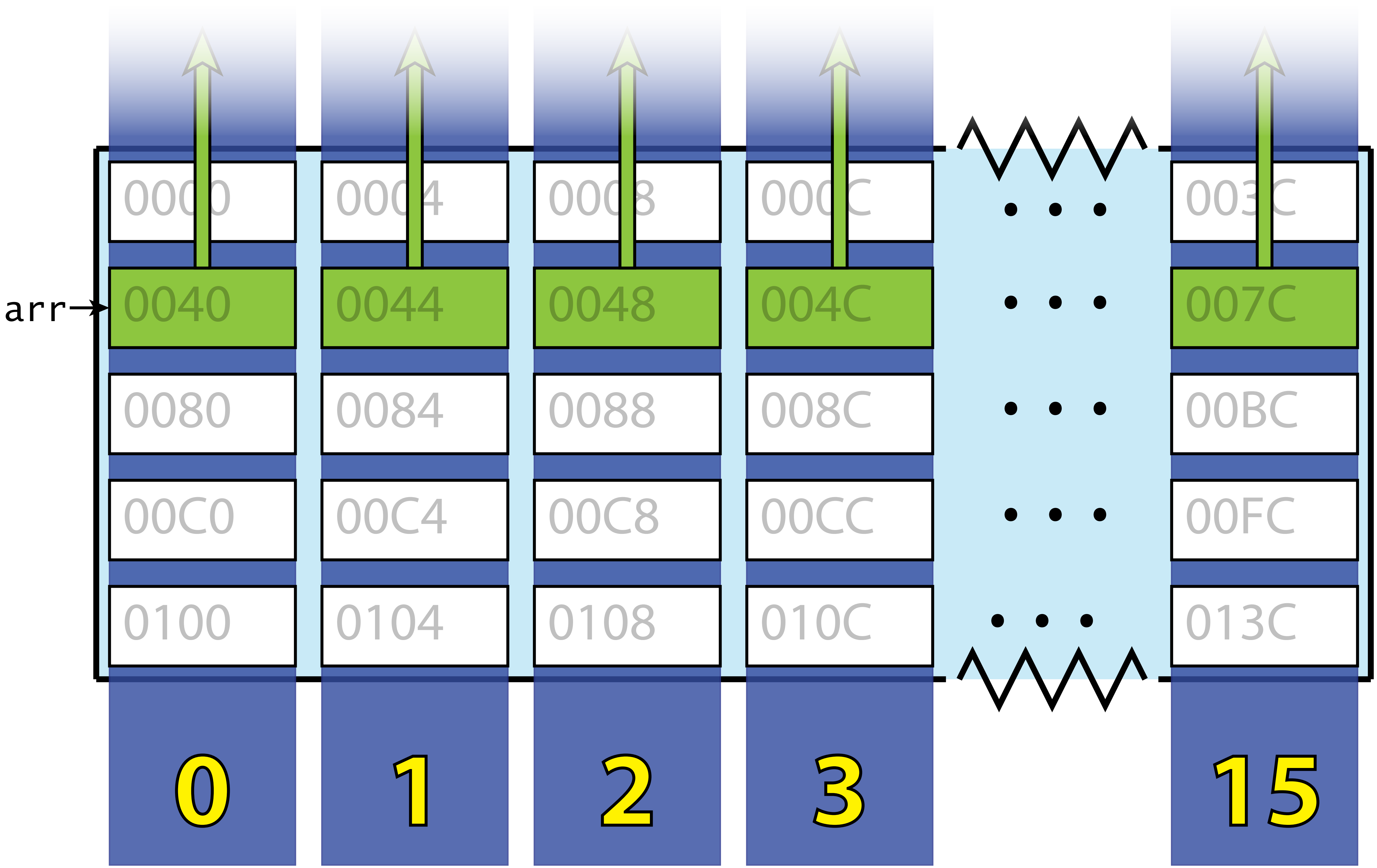
```
__shared__ int arr[128];
...
int v = arr[ ?? ]
```



Shared memory

- in practice?

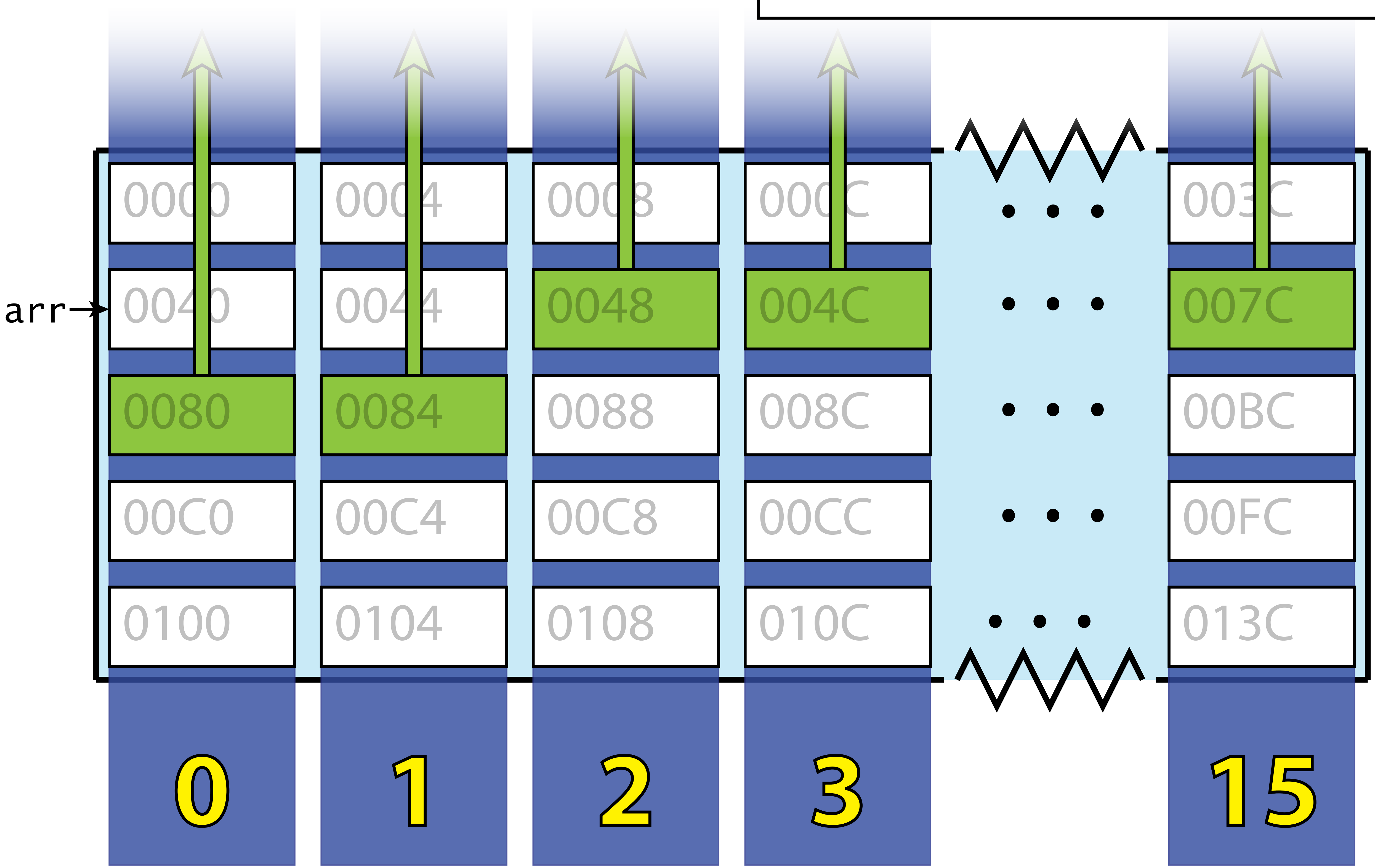
```
__shared__ int arr[128];
...
int v = arr[ threadIdx.x ]
```



Shared memory

- in practice?

```
__shared__ int arr[128];
...
int v = arr[ threadIdx.x ];
v=arr[threadIdx.x+2];
```

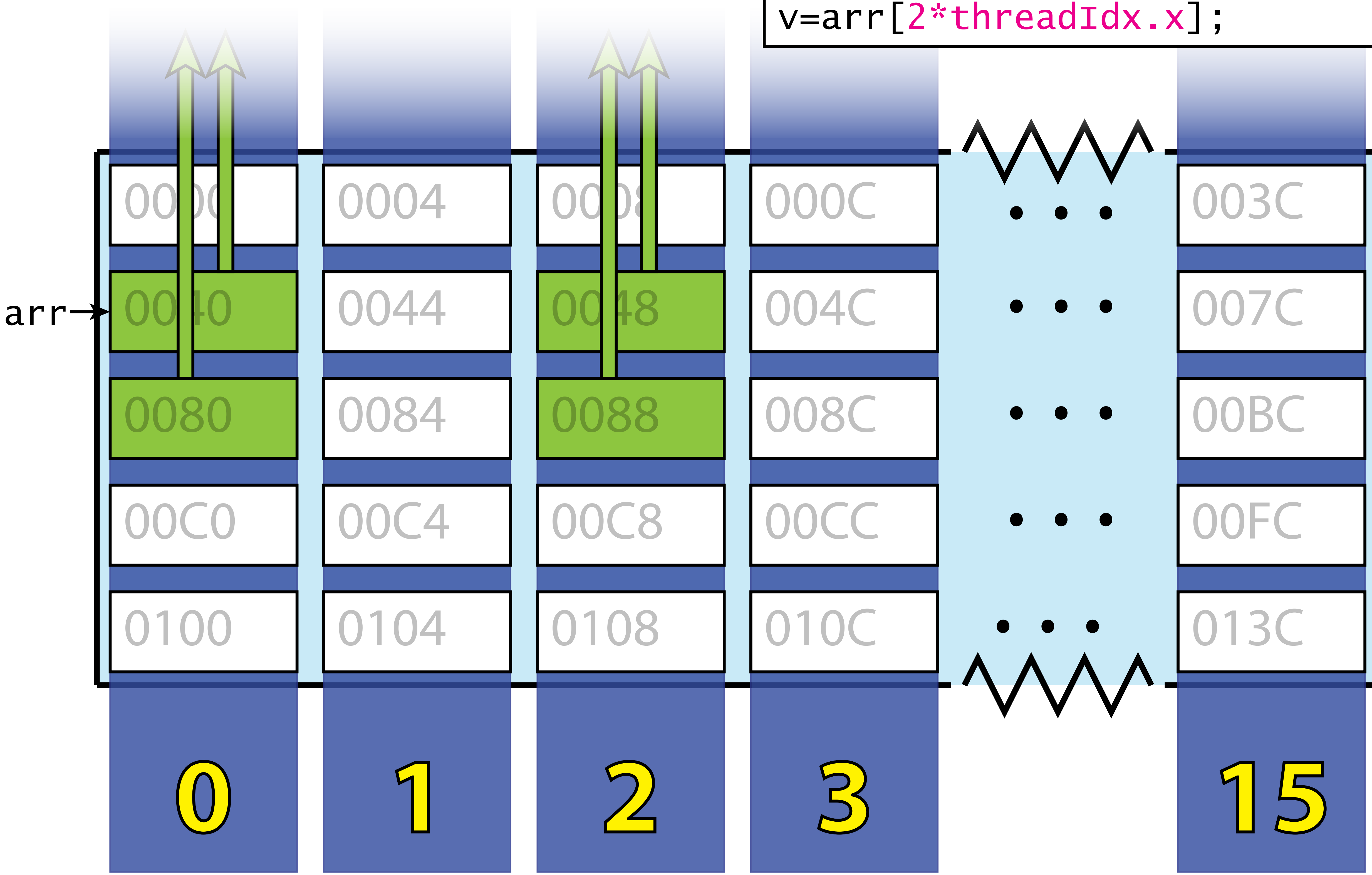


Shared memory

- in practice?

```
__shared__ int arr[128];
...
int v = arr[ threadIdx.x ];
v=arr[threadIdx.x+2];
v=arr[2*threadIdx.x];
```

2-way



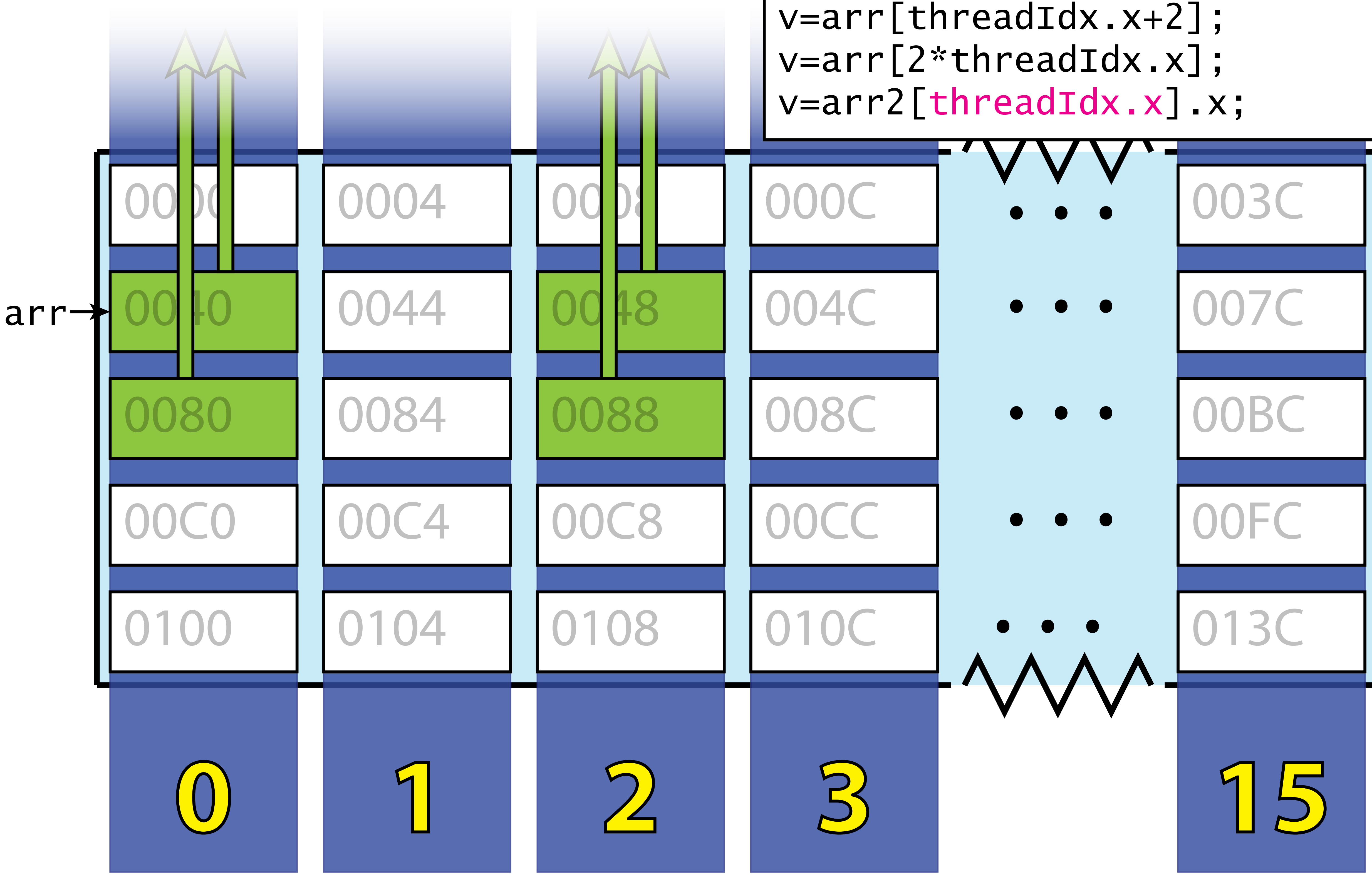
Shared memory

- in practice?

```
__shared__ int arr[128];
__shared__ int2 arr2[128];
...
int v = arr[ threadIdx.x ];
v=arr[threadIdx.x+2];
v=arr[2*threadIdx.x];
v=arr2[threadIdx.x].x;
```

2-way

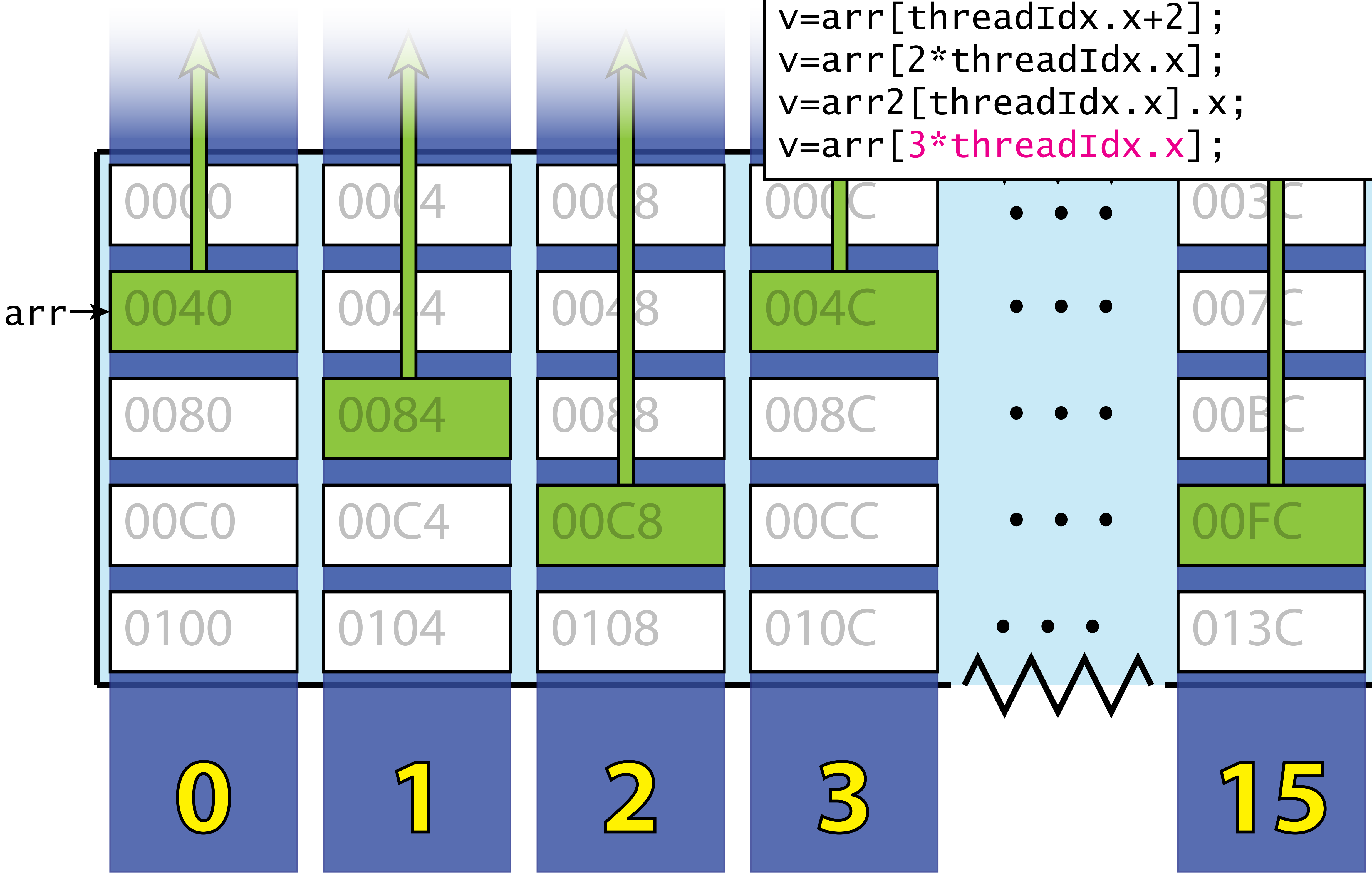
2-way



Shared memory

- in practice?

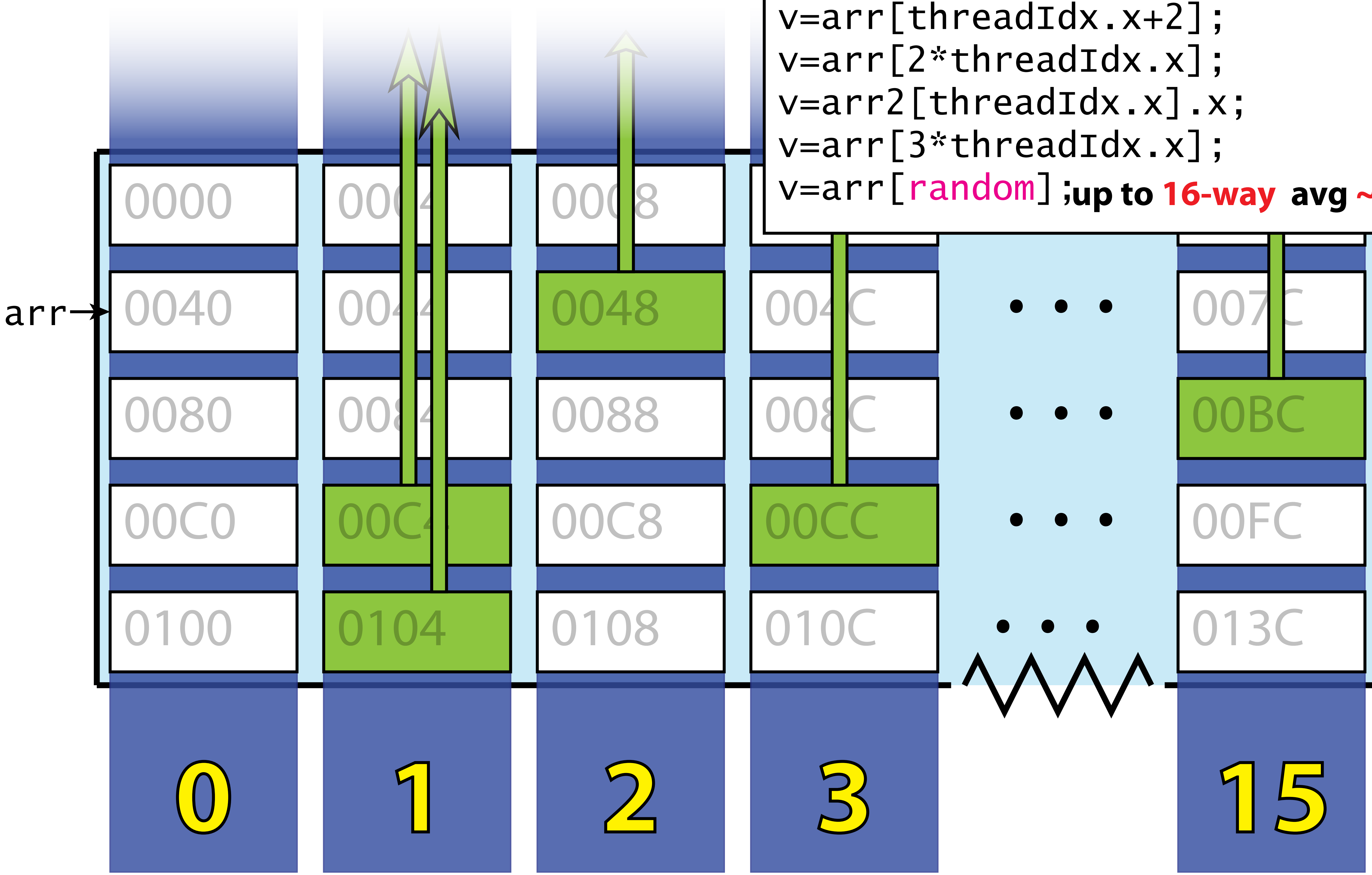
```
__shared__ int arr[128];
__shared__ int2 arr2[128];
...
int v = arr[ threadIdx.x ];
v=arr[threadIdx.x+2];
v=arr[2*threadIdx.x];      2-way
v=arr2[threadIdx.x].x;     2-way
v=arr[3*threadIdx.x];
```



Shared memory

- in practice?

```
__shared__ int arr[128];
__shared__ int2 arr2[128];
...
int v = arr[ threadIdx.x ];
v=arr[threadIdx.x+2];
v=arr[2*threadIdx.x];          2-way
v=arr2[threadIdx.x].x;         2-way
v=arr[3*threadIdx.x];
v=arr[random];  up to 16-way  avg ~5-way
```



Shared memory

- in practice?
- in general?

```
__shared__ int arr[128];
__shared__ int2 arr2[128];
...
int v = arr[ threadIdx.x ];
v=arr[threadIdx.x+2];
v=arr[2*threadIdx.x];            2-way
v=arr2[threadIdx.x].x;           2-way
v=arr[3*threadIdx.x];
v=arr[random];up to 16-way avg ~5-way
```

Array element size

- 32-bit types
- *p**32-bit types (*p* is odd)
- 2-way 64-bit types (`double`)
- 4-way 128-bit types (`float4`)
- 2-way 16-bit types (`short`)

Shared memory

- in practice?
- in general?

```
__shared__ int arr[128];
__shared__ int2 arr2[128];
...
int v = arr[ threadIdx.x ];       ✓
v=arr[threadIdx.x+2];             ✓
v=arr[2*threadIdx.x];             2-way
v=arr2[threadIdx.x].x;            2-way
v=arr[3*threadIdx.x];             ✓
v=arr[random];up to 16-way avg ~5-way
```

Array element size + Access pattern

Array element size	Access pattern
✓ 32-bit types	✓ `arr[threadIdx.x+offset]`
✓ *p**32-bit types (*p* is odd)	✓ `arr[p*threadIdx.x]` (*p* is odd)
2-way 64-bit types (`double`)	2^e-way `arr[2^e*threadIdx.x]`
4-way 128-bit types (`float4`)	
2-way 16-bit types (`short`)	

Shared memory

- in practice?
- in general?

```
__shared__ int arr[128];
__shared__ int2 arr2[128];
...
int v = arr[ threadIdx.x ];
v=arr[threadIdx.x+2];
v=arr[2*threadIdx.x];          2-way
v=arr2[threadIdx.x].x;         2-way
v=arr[3*threadIdx.x];
v=arr[random] ;up to 16-way  avg ~5-way
```

Array element size + Access pattern

32-bit types | arr[threadIdx.x+offset]

p*32-bit types (p is odd) | arr[p*threadIdx.x] (p is odd)

2-way 64-bit types (double) | 2^e-way arr[2^e*threadIdx.x]

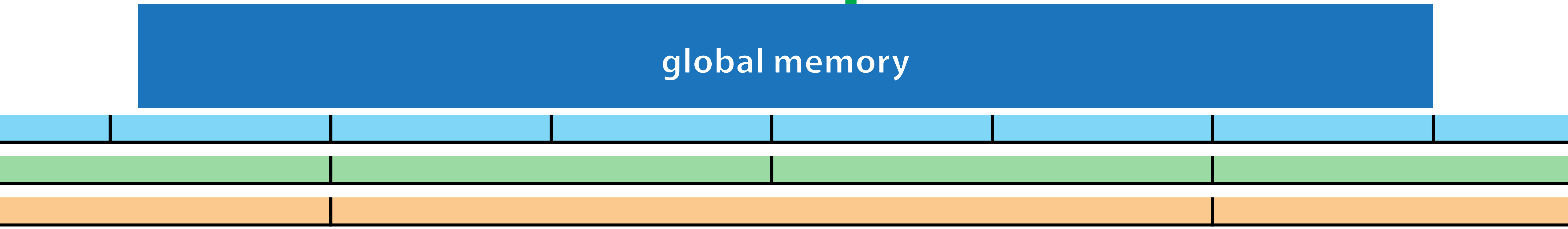
4-way 128-bit types (float4) 2-way

2-way 16-bit types (short)

[pre-Fermi] 16 banks

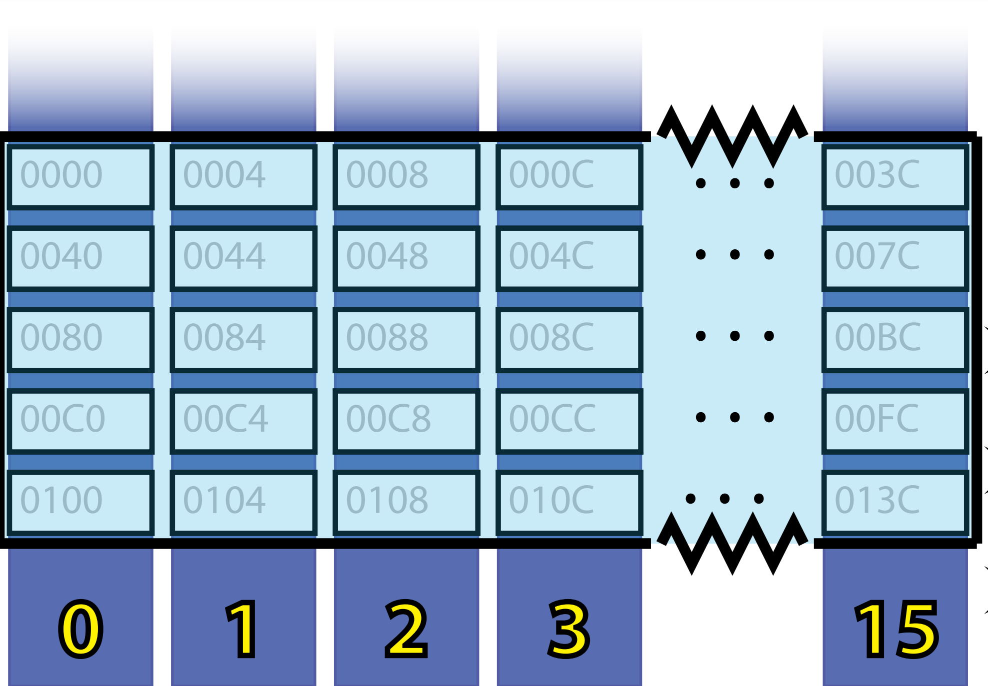
[Fermi] [Kepler] 32 banks

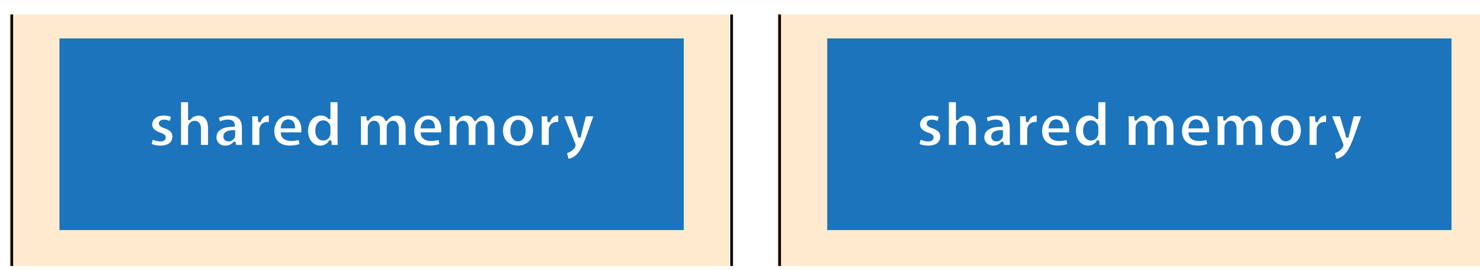
Recap



- segmented into 32B, 64B and 128B chunks
- each warp should access as few chunks as possible

- n addresses = n memory transactions
- all threads in a warp should read the same thing





- “vertical” memory banks
- each thread should access different bank
- or should read the same thing (broadcast)

Thank you

Questions?

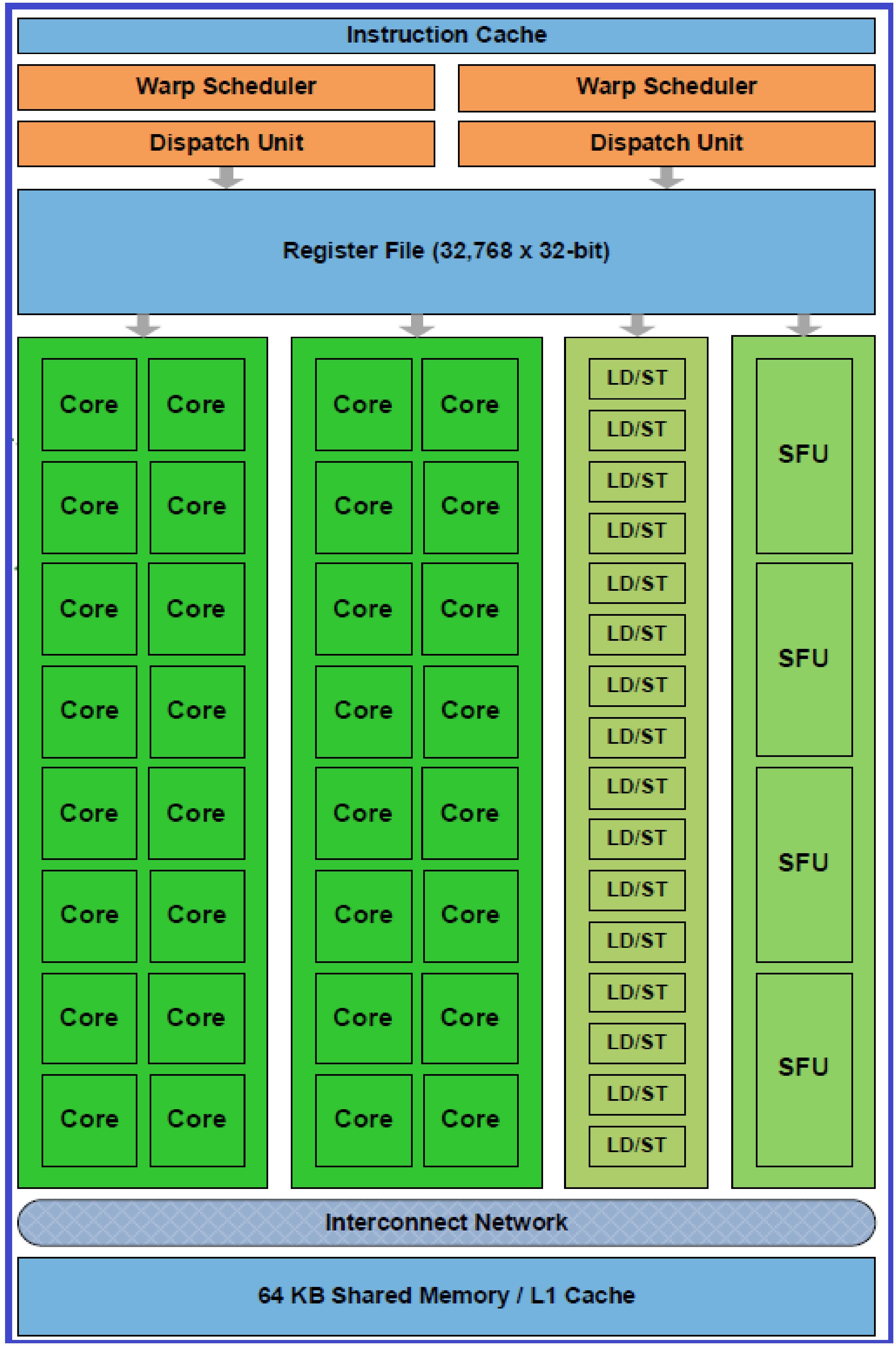
2.CUDA线程交互

Fermi架构中的SM

SM含：32个SP，16个存储器读写单元LD/ST，
4个特殊功能计算单元SFU
（用于计算超越函数等复杂操作），
64KB的共享存储器，32768个32位寄存器，
2个Warp调度器与2个指令分派单元。

线程执行：
SM调度时将32个线程组成一组
（即一个warp）并发执行。
2个warp调度器与2个指令分派单
元能够将2个warp同时进行发射和执行：
双warp调度器先选择两个warp
然后从每个warp发射一条指令到
一个十六核心的组，
或是十六个读写单元
或是四个SFU。

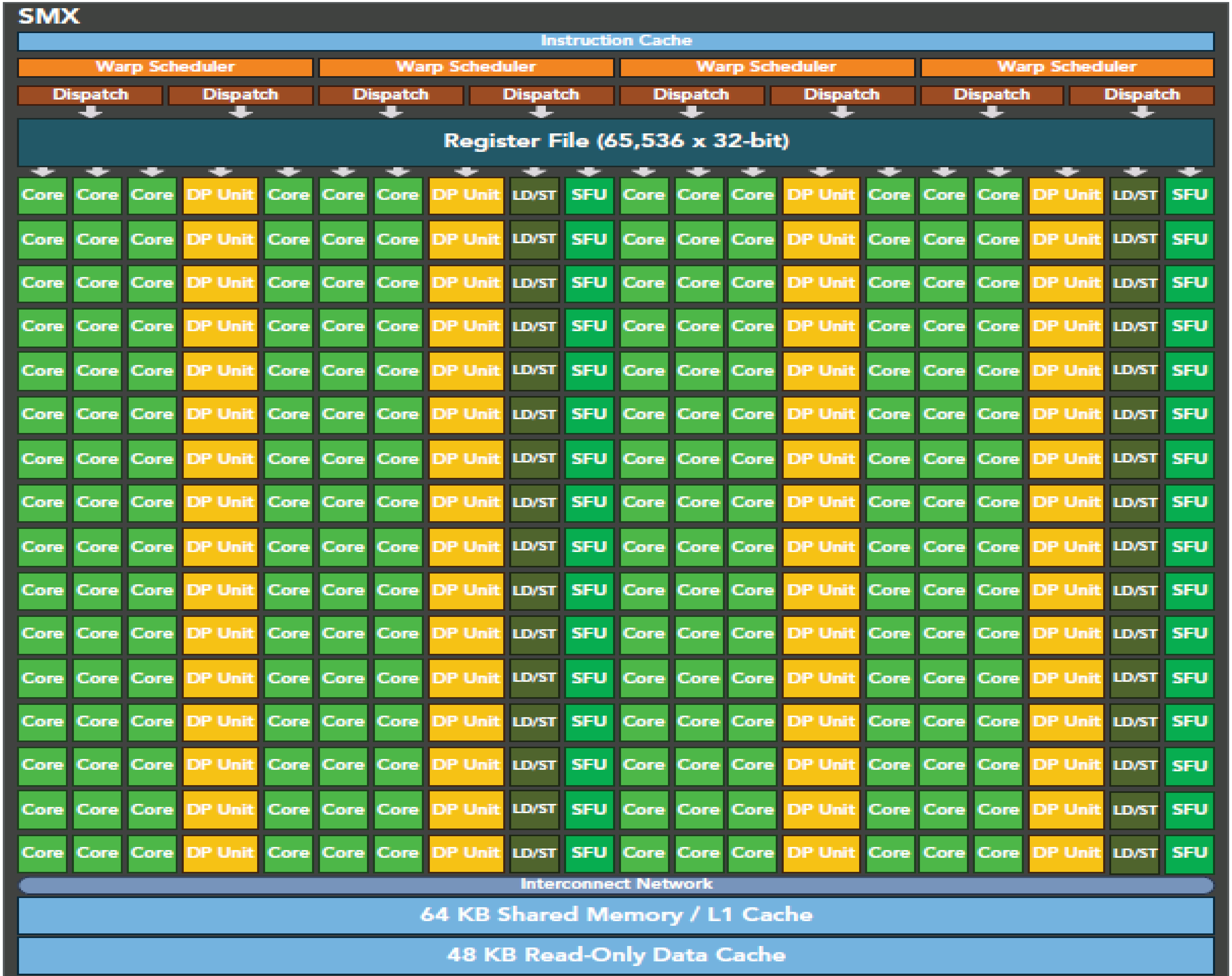
warp执行是独立的
调度器无需检查指令流内的依赖性



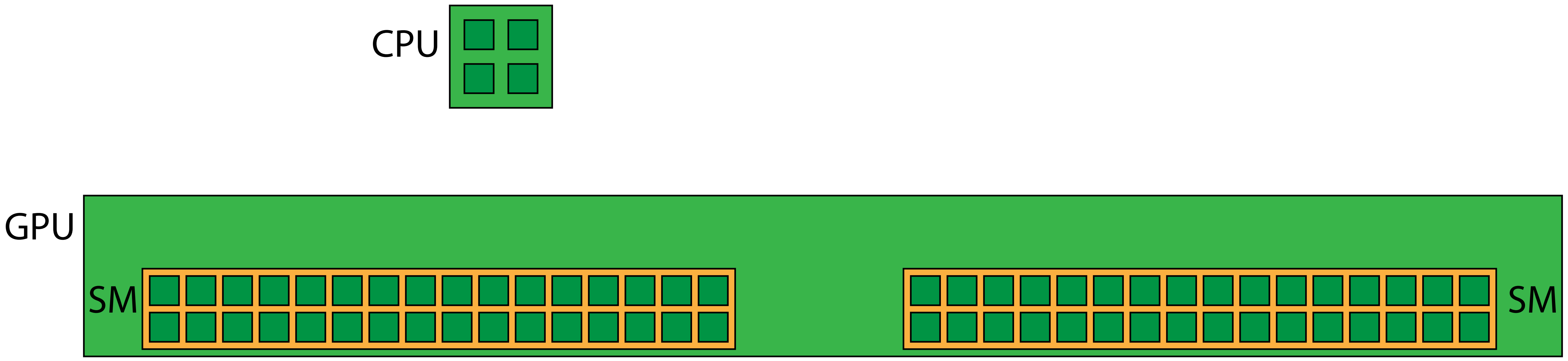
一种典型架构的SM

每一个SM有两个SM Processing Block (SMP) ，每个SMP里有CUDA Core: Streaming Processor (SP) 。
每一个SM有自己的指令缓存，L1缓存，共享内存。每个SMP有自己的Warp Scheduler、Register File等。
注意：CUDA Core是Single Precision的，也就是计算float单精度的。双精度Double Precision是那些黄色的模块
这个SM里边由32个DP Unit，由64个CUDA Core，所以单精度双精度单元数量比是2:1。
LD/ST 是load store unit，用来内存操作的。SFU是Special function unit，用来做cuda的intrinsic function (内嵌函数) 的，
类似于__cos()这种。

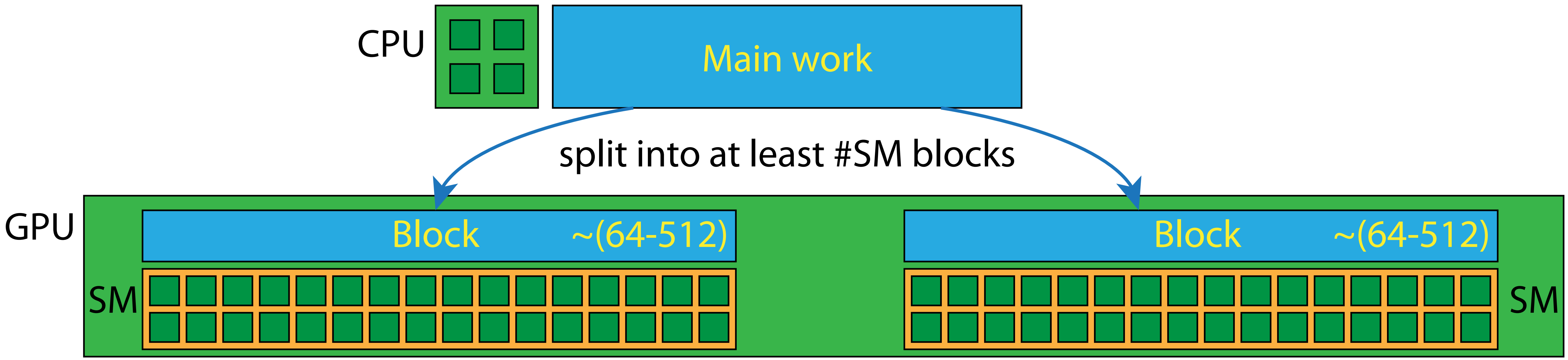
开普勒（Kepler）构架中的SMX（即SM）



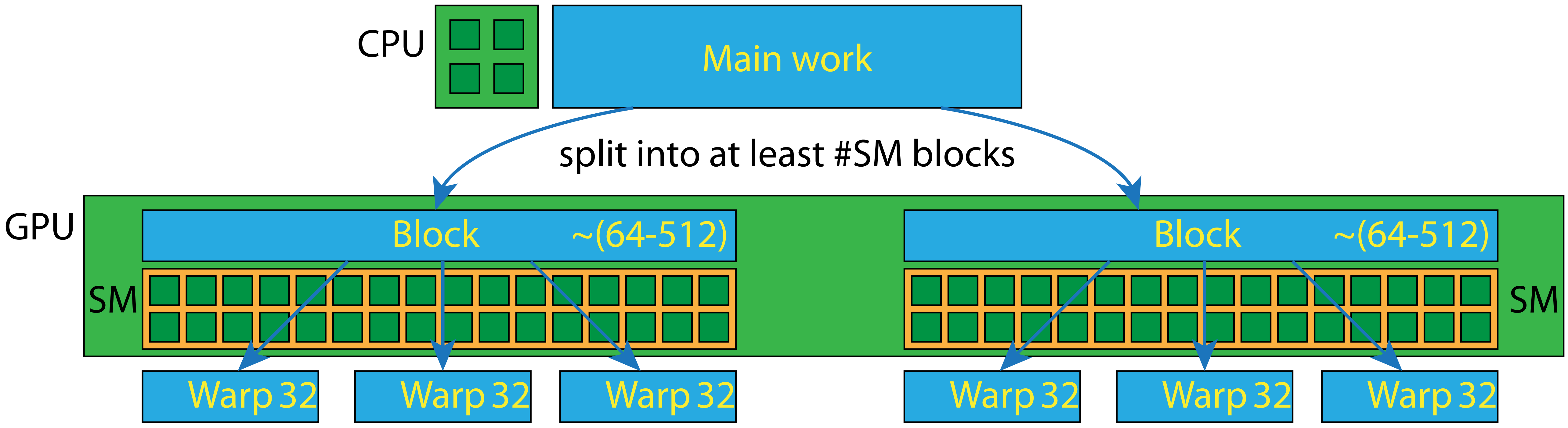
Work hierarchy



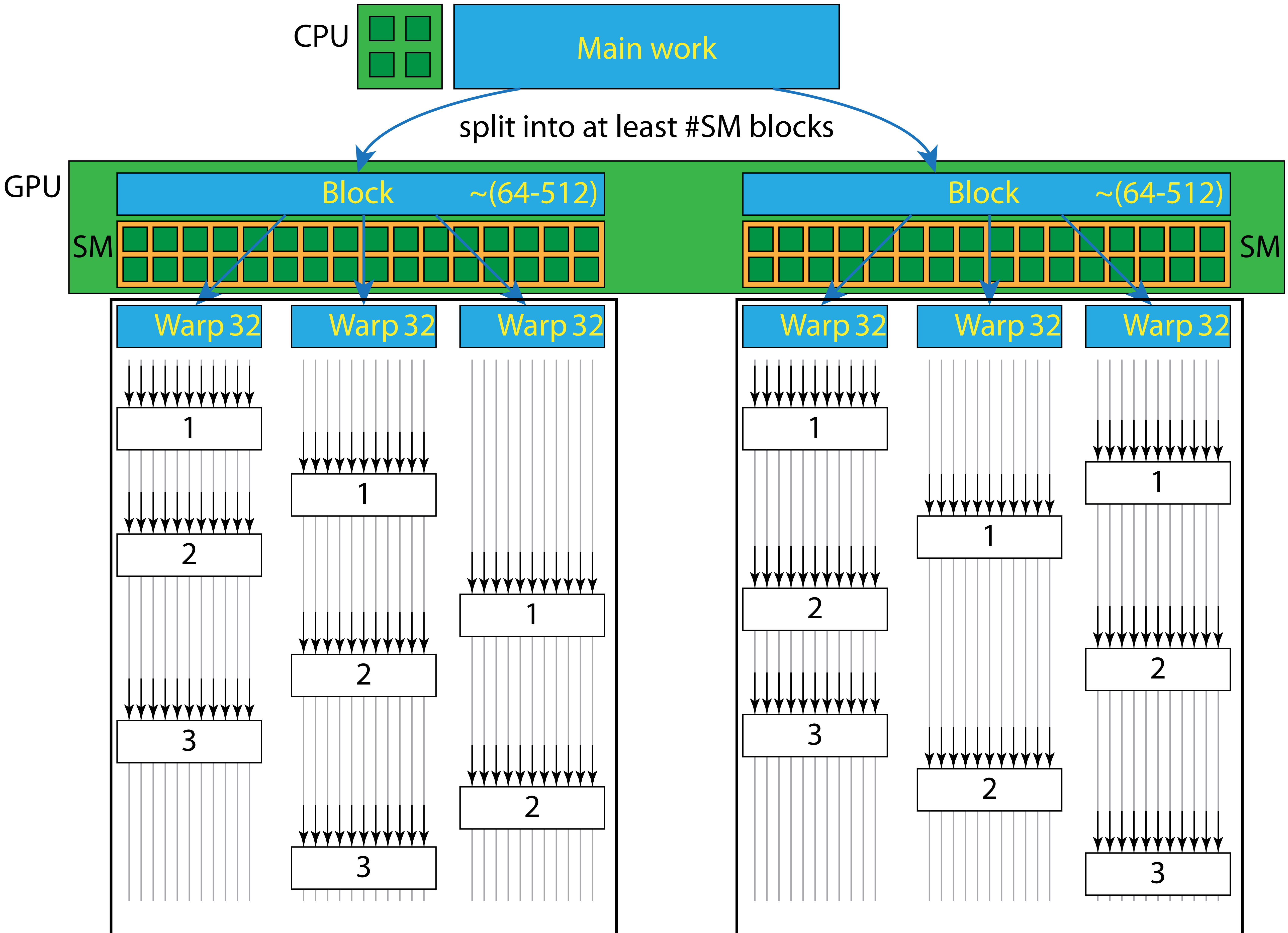
Work hierarchy



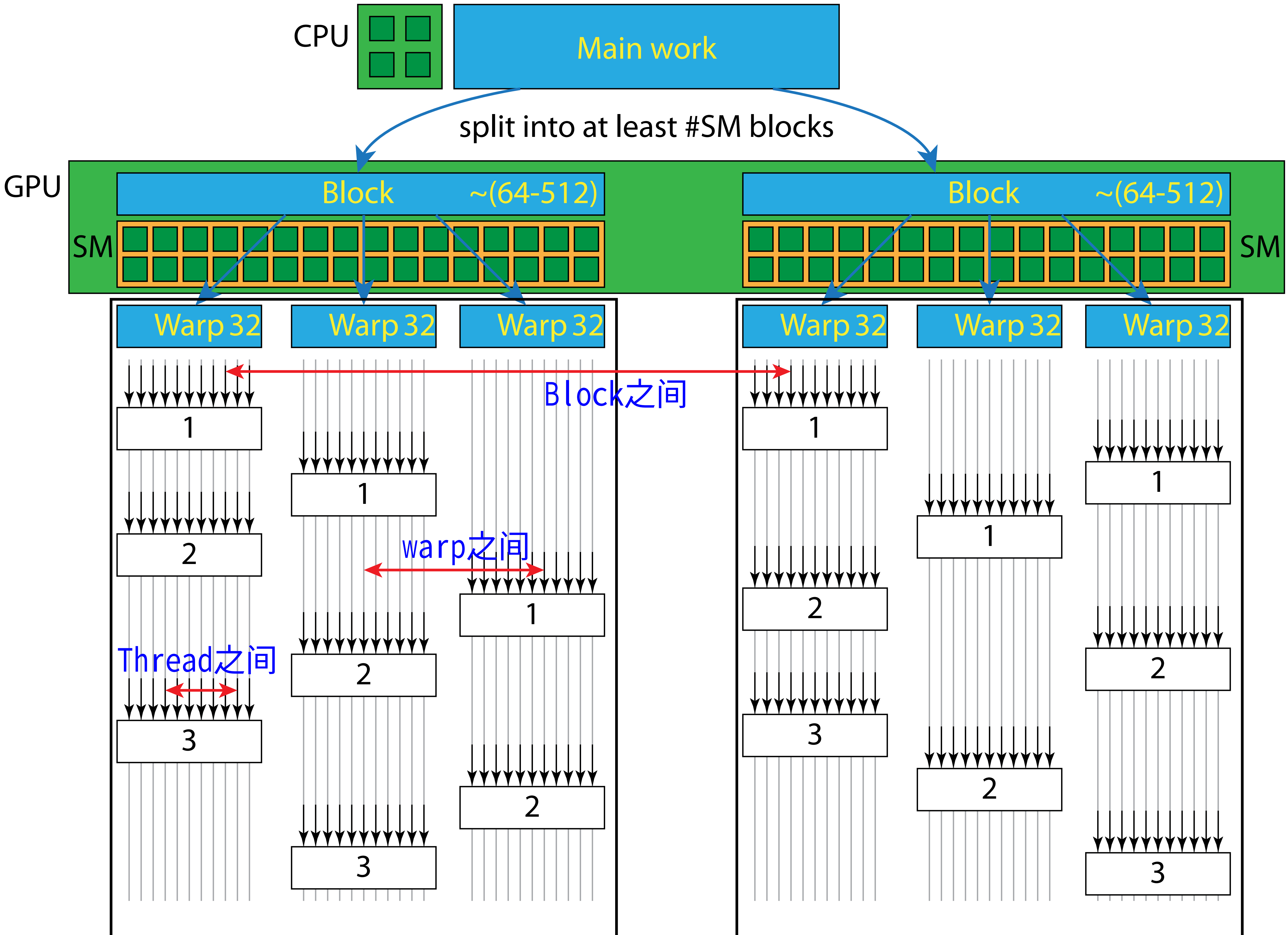
Work hierarchy



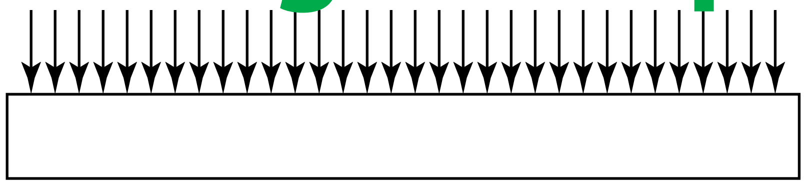
Work hierarchy



Work hierarchy



Single warp



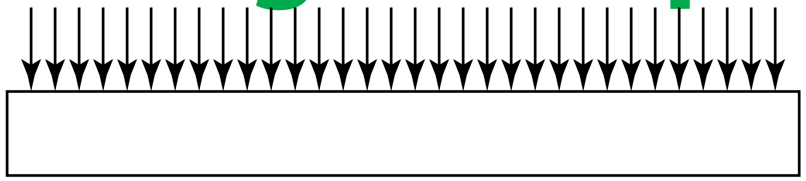
```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

- 32 threads [pre-Fermi] [Fermi] [Kepler]
- in-sync 同步的
- SIMD

将要讨论：

- Shared memory reads/writes
- Branching
- Warp functions

Single warp

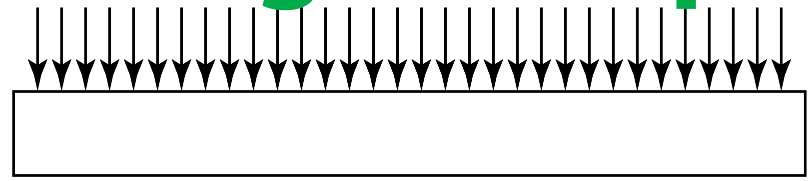


```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

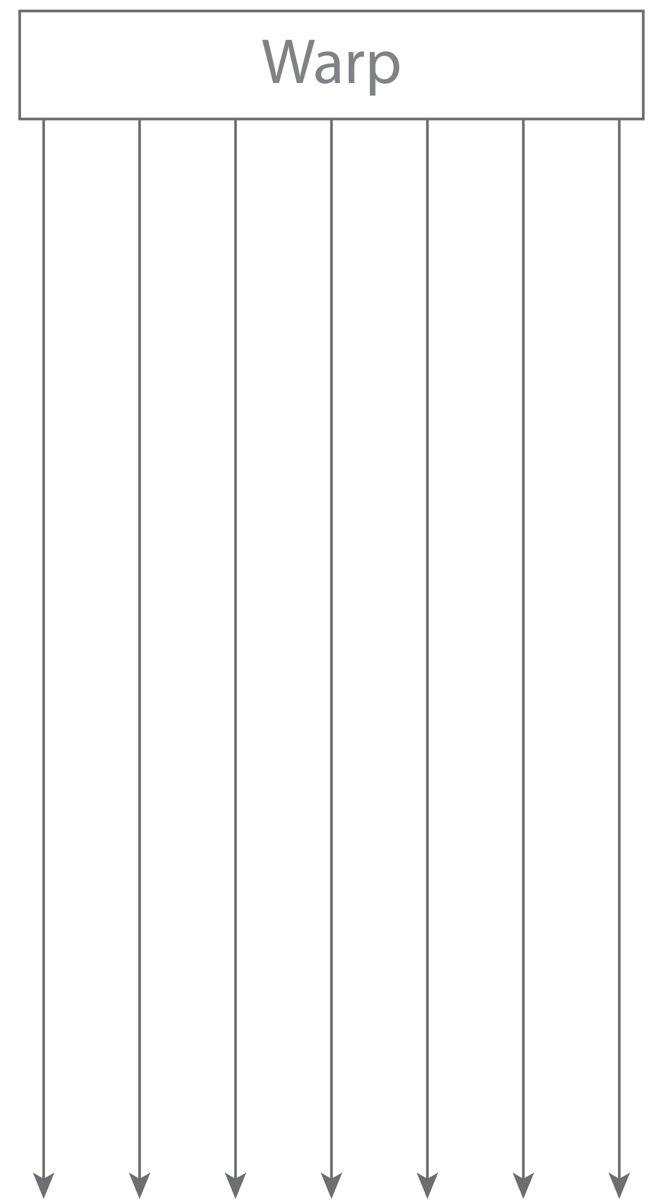
```
if (warpIdx == 0) {
  int v = 0;
  ++v;
  v == ?
}
```

```
if (warpIdx == 0) {
  __shared__ int v;
  v = 0;
  ++v;
  v == ?
}
```

Single warp



```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

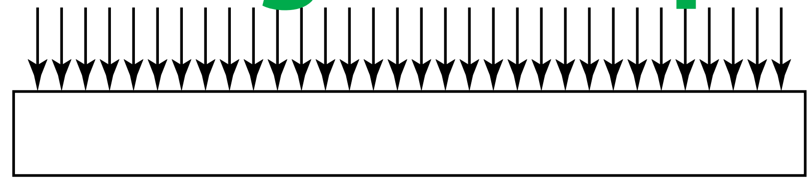



```
if (warpIdx == 0) {
  int v = 0;  第0号warp的每个线程都有局部的v
  ++v;
  v == ?
}
```

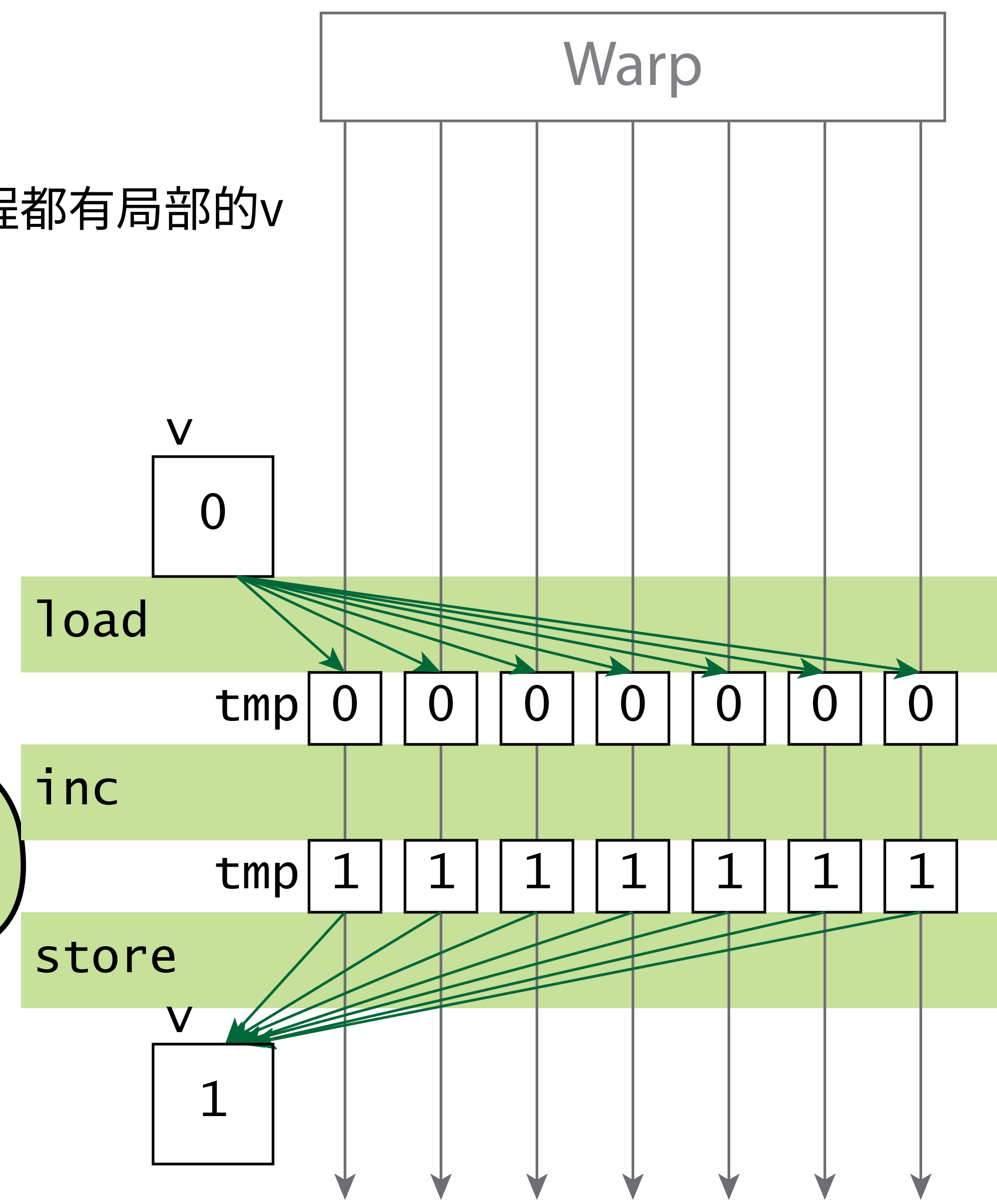
```
if (warpIdx == 0) {
  __shared__ int v;
  v = 0;
  ++v;
  v == ?
}
```

```
load v→tmp;
inc tmp; tmp + 1
store v←tmp;
```

Single warp



```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```



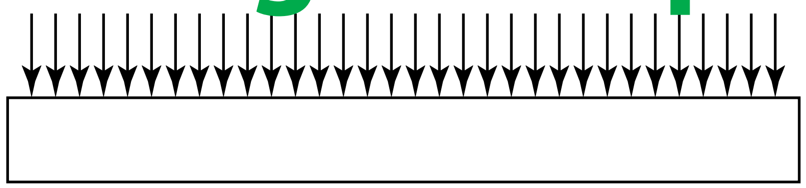

```
if (warpIdx == 0) {
  int v = 0;
  ++v;
  v == ?
}
```

第0号warp的每个线程都有局部的v

```
if (warpIdx == 0) {
  __shared__ int v;
  v = 0;
  ++v;
  v == ?
}
```

```
load v→tmp;
inc tmp; tmp + 1
store v←tmp;
```

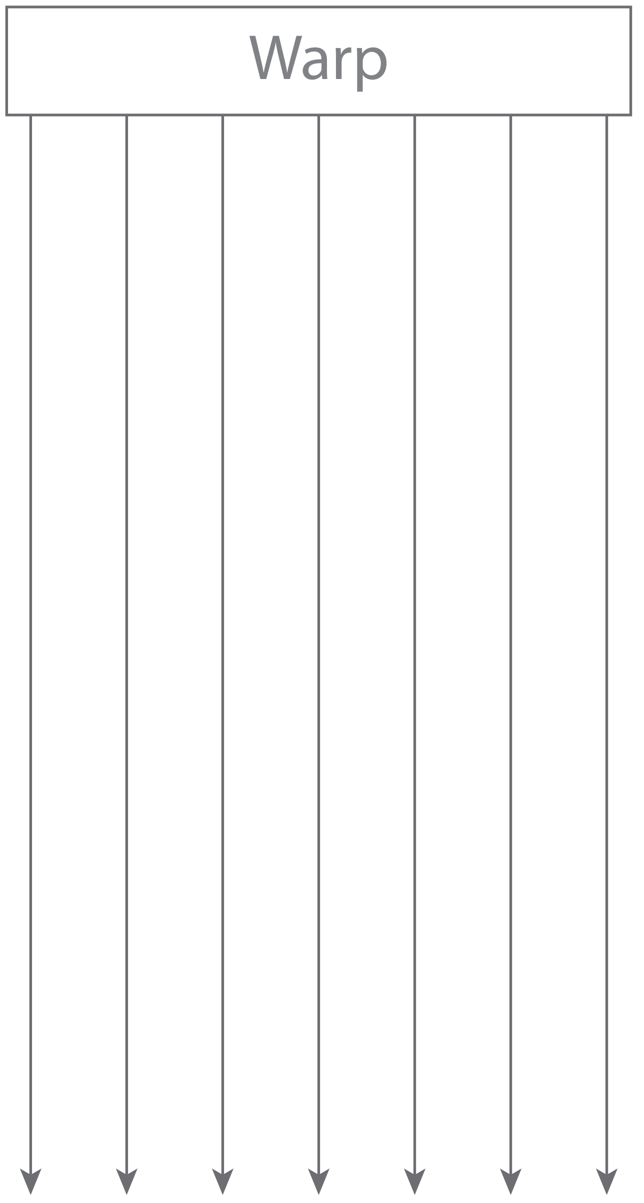
Single warp



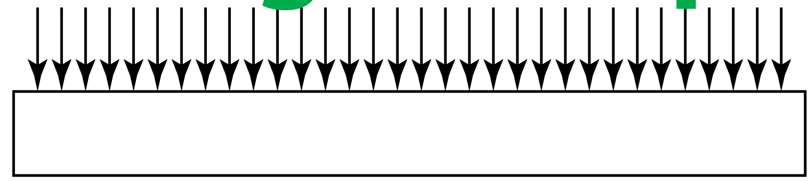
```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

```
if (warpIdx == 0) {
  int v = 0;
  ++v;
  v == ?
}
```

```
if (warpIdx == 0) {
  __shared__ int v;
  v = 0;
  atomicAdd(&v,1);
  v == ?
}
```



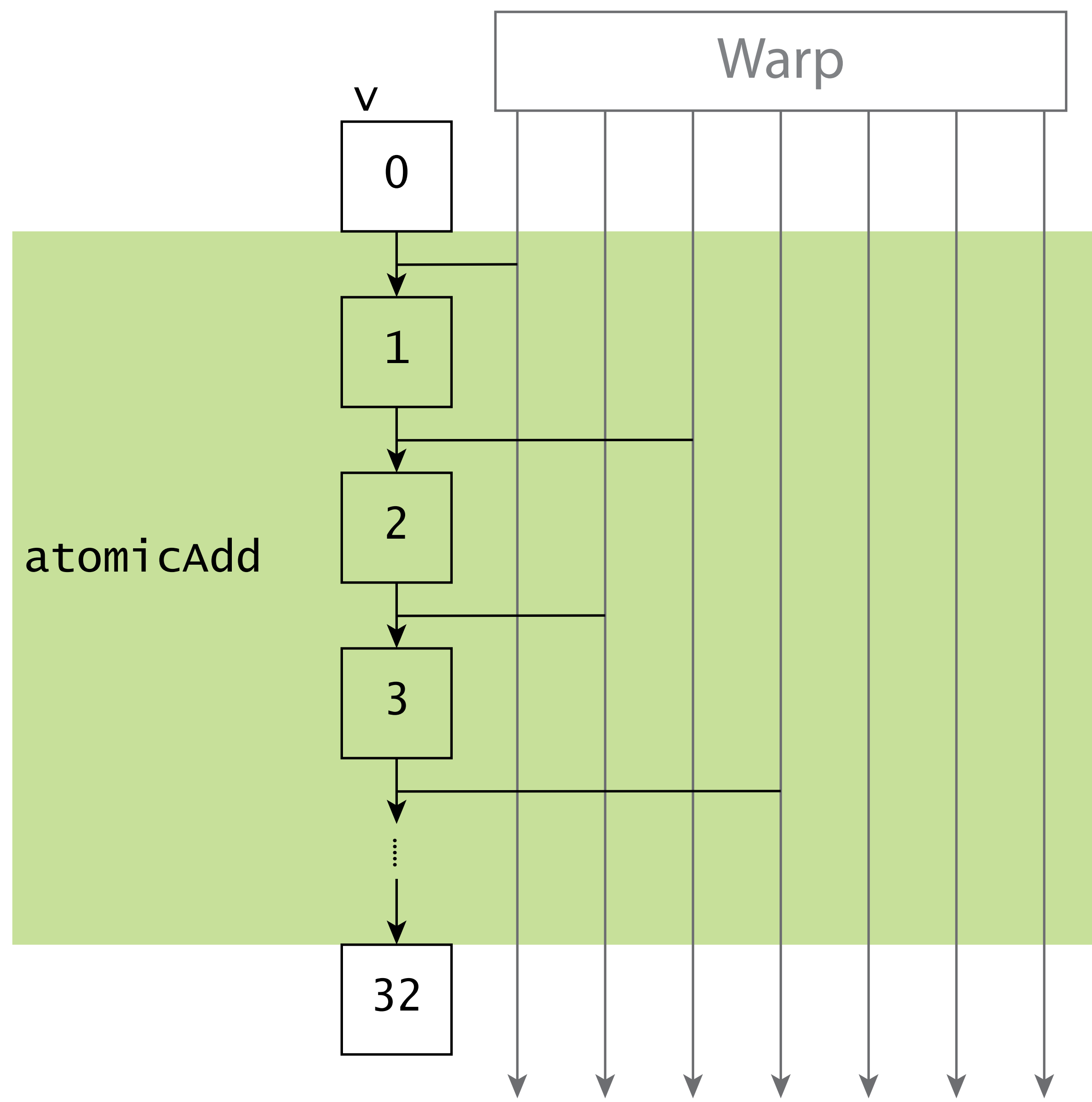
Single warp



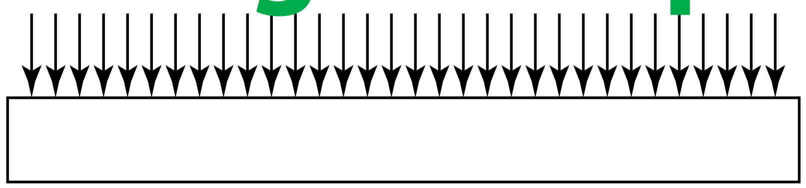
```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

```
if (warpIdx == 0) {
  int v = 0;
  ++v;
  v == ?
}
```

```
if (warpIdx == 0) {
  __shared__ int v;
  v = 0;
  atomicAdd(&v,1);
  v == ?
}
```



Single warp



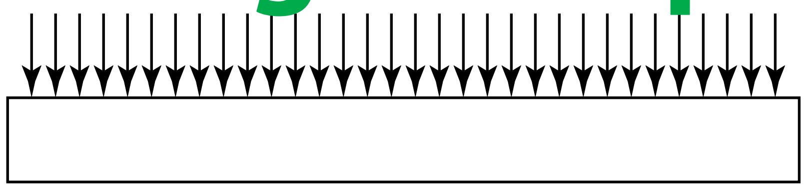
```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

```
if (warpIdx == 0) {
  int v = 0;
  ++v;
  v == ?
}
```

```
if (warpIdx == 0) {
  __shared__ int v;
  v = 0;
  atomicAdd(&v,1);
  v == ?
}
```

```
atomicAdd(T* addr, T val);
atomicSub(T* addr, T val);
atomicExch(T* addr, T val);
atomicMin(T* addr, T val);
atomicMax(T* addr, T val);
atomicInc(T* addr, T val);
atomicDex(T* addr, T val);
atomicCAS(T* addr, T cmp, T val);
atomicAnd(T* addr, T val);
atomicOr(T* addr, T val);
atomicXor(T* addr, T val);
```

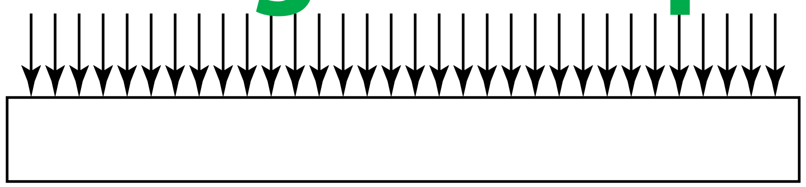
Single warp



```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

- No atomic for longer section of code 一个更大代码段没有原子操作，
- Semaphors, critical section, could be implemented but do not do it! 可用信号量、临界区等实现，但不要这样做：

Single warp



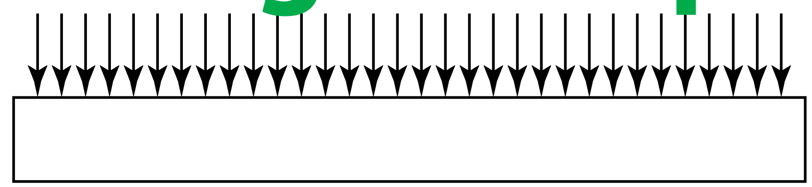
```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

- No atomic for longer section of code —个更大代码段没有原子操作，
- Semaphors, critical section, could be implemented but do not do it!　可用信号量、临界区等实现，但不要这样做：
  - It would be very slow
  - Can lead to a deadlock due to SIMD

If you need critical section:

  - Try to redesign the algorithm
  - Consider using CPU

Single warp

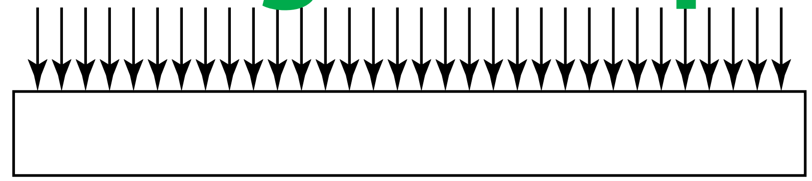


```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

- 32 threads [pre-Fermi] [Fermi] [Kepler]
- in-sync
- SIMD

- Shared memory reads/writes
- now Branching
- Warp functions

Single warp

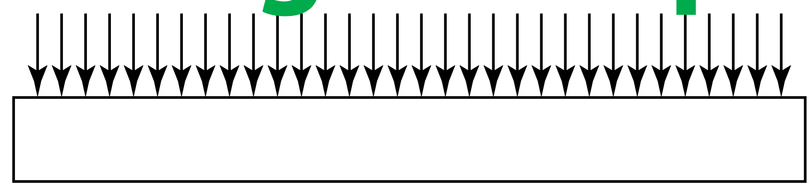


```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

- 32 threads [pre-Fermi] [Fermi] [Kepler]
- in-sync
- SIMD = Single Instruction Multiple Data

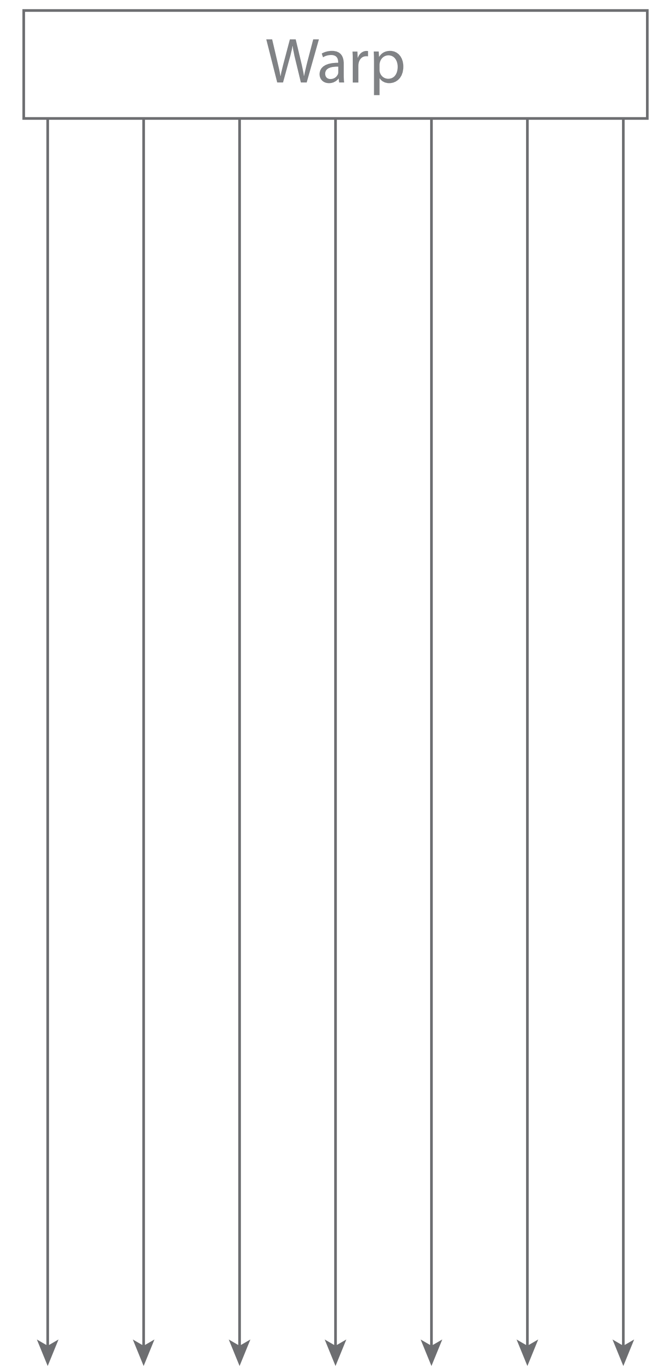
- Shared memory reads/writes
- now Branching
- Warp functions

Single warp

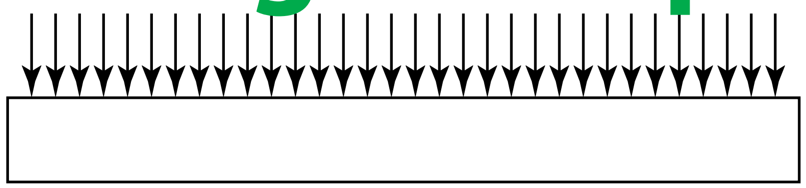


```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

```
if (warpIdx == 0) {
  if (laneIdx % 2 == 0) {
    doA();
  } else {
    doB();
  }
}
```

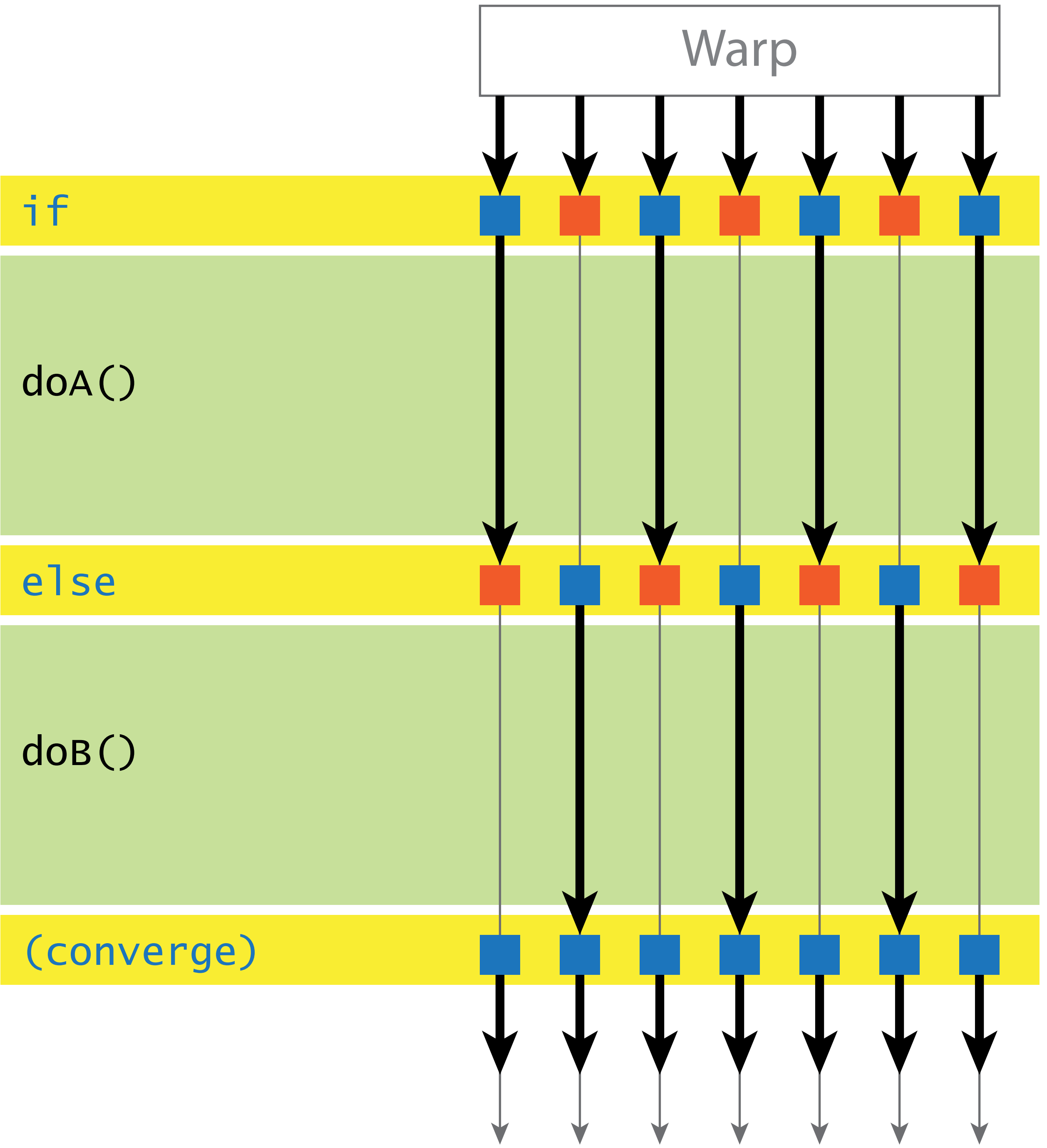


Single warp

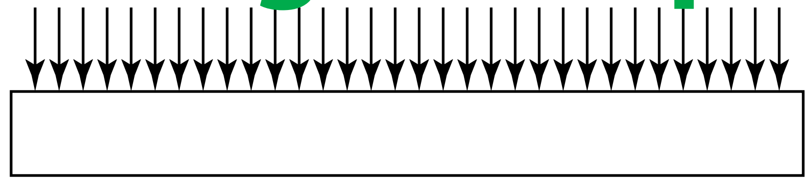


```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

```
if (warpIdx == 0) {
  if (laneIdx % 2 == 0) {
    doA();
  } else {
    doB();
  }
}
```



Single warp



Divergent code
all branches taken
(unless no thread in a warp takes it)

⇓

Branch-heavy code inefficient

Weird control flow
(multi-level `break`, `continue`, `return`)

⇓

conservative convergence

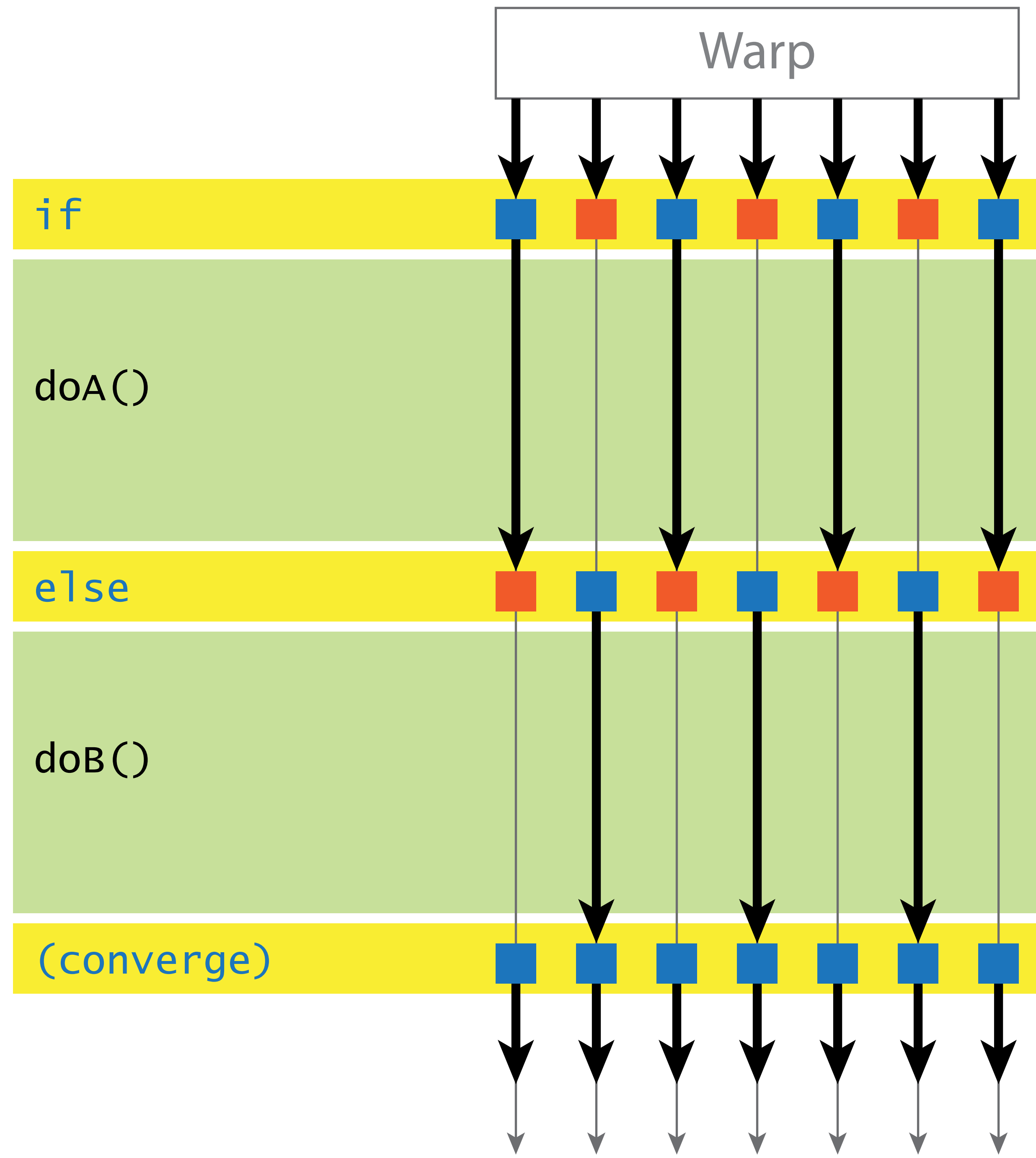
⇓

inefficient, bug prone

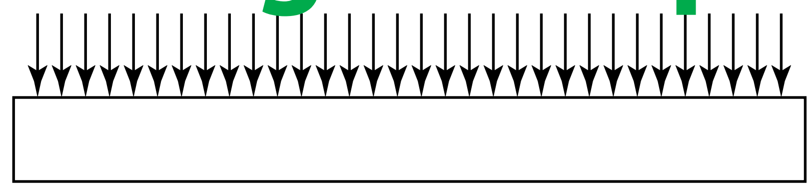
No `throw`

No `goto`

```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```



Single warp

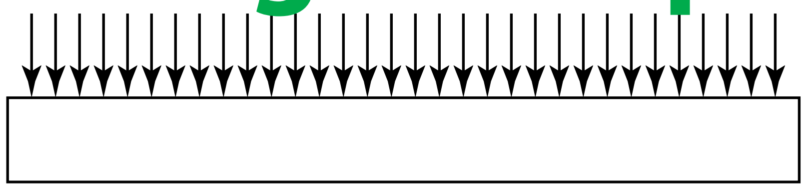


```
int warpIdx = threadIdx.x / 32;
int laneIdx = threadIdx.x % 32;
```

- 32 threads [pre-Fermi] [Fermi] [Kepler]
- in-sync
- SIMD

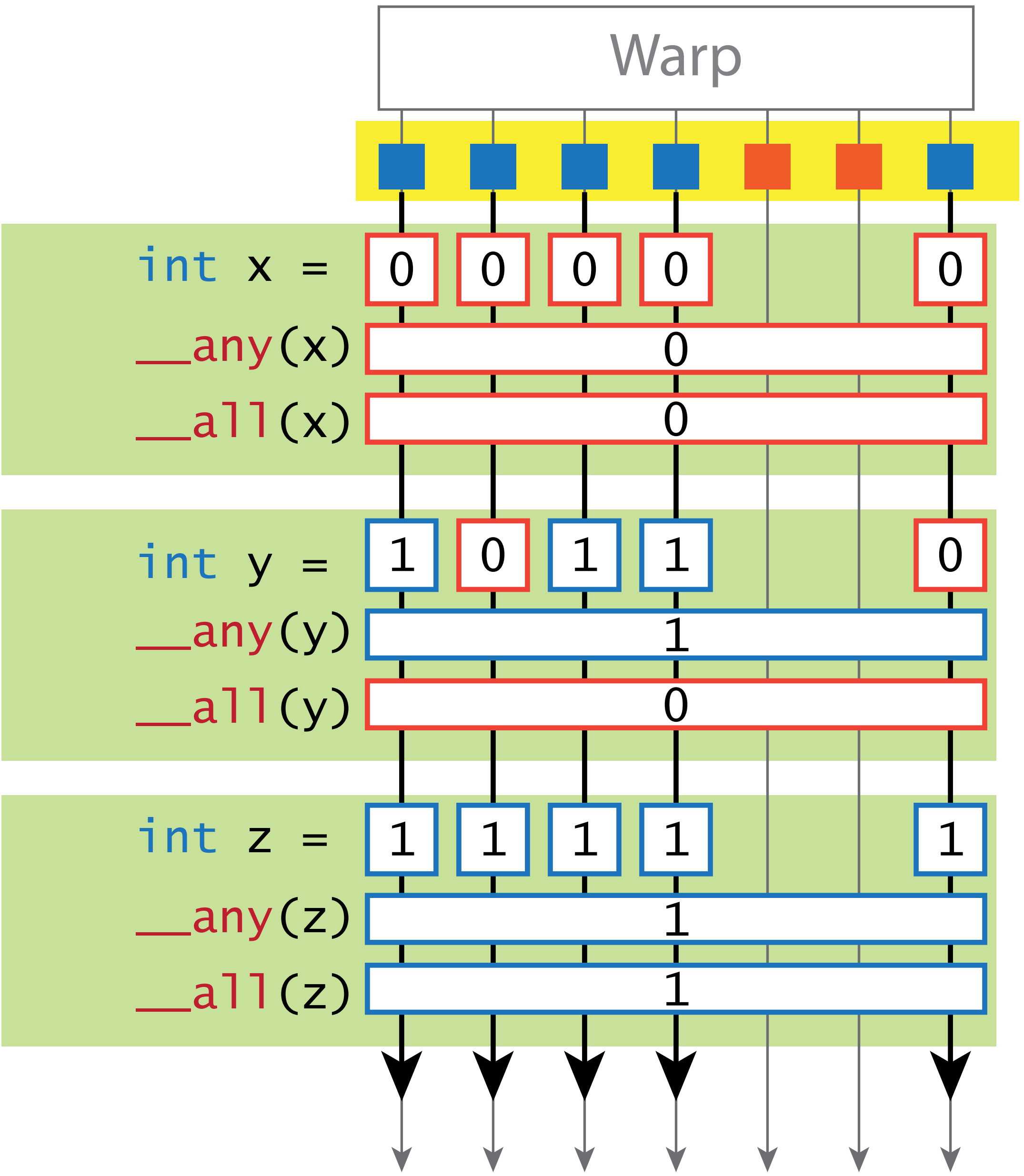
- Shared memory reads/writes
- Branching
- now Warp functions

Single warp

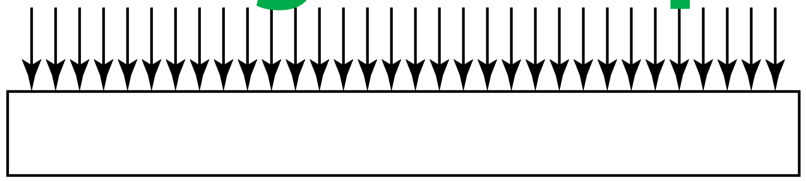


any, all [CC 1.2+] - [Fermi] [Kepler], some [pre-Fermi]

```
int __any(int predicate);
int __all(int predicate);

__all(predicate):
   Evaluate predicate for all active threads
   of the warp and return non-zero if and
   only if predicate evaluates to non-zero
   for all of them.
__any(predicate):
   Evaluate predicate for all active threads
   of the warp and return non-zero if and
   only if predicate evaluates to non-zero
   for any of them.
```



Single warp

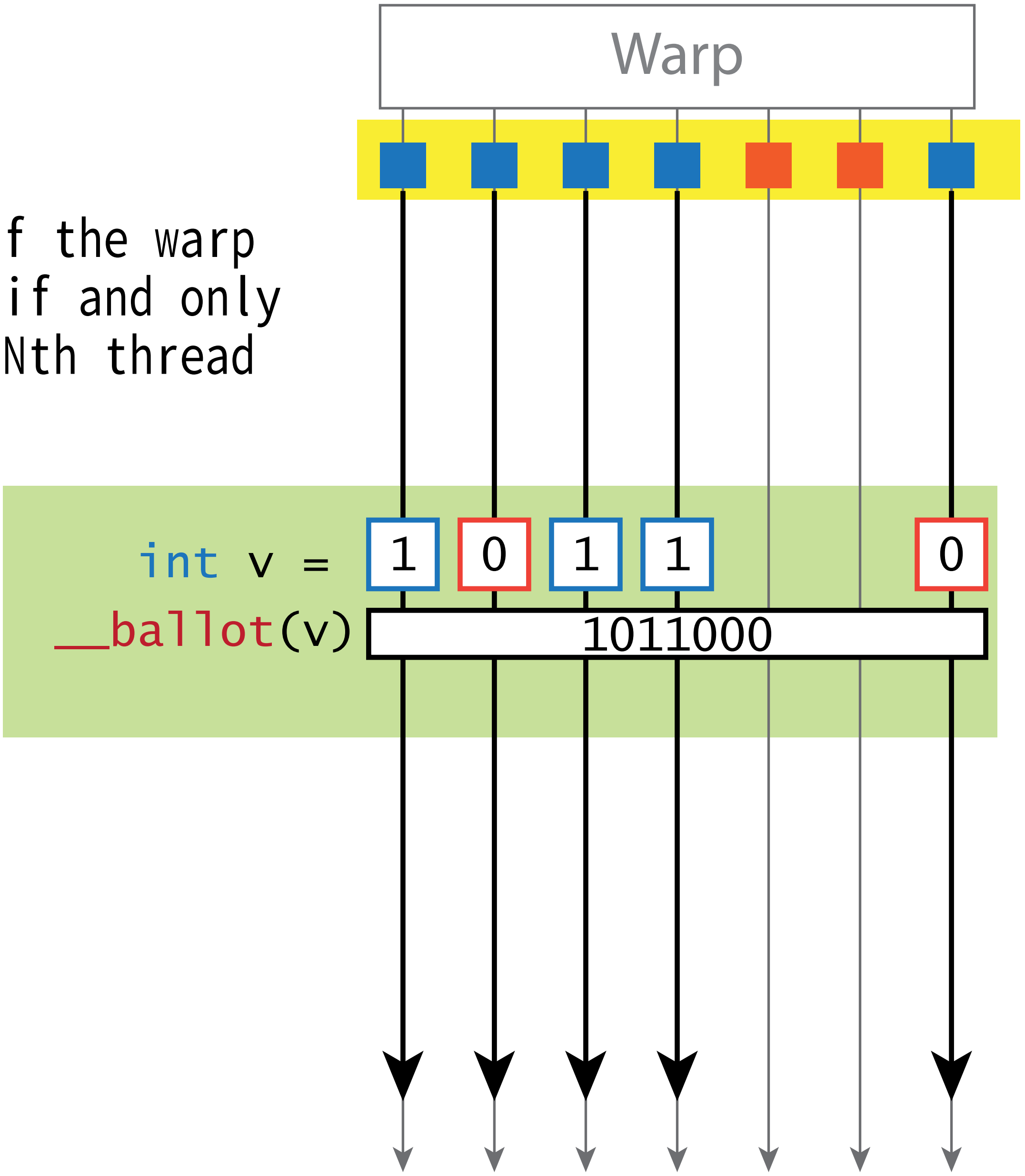


any, all [CC 1.2+] - [Fermi] [Kepler], some [pre-Fermi]

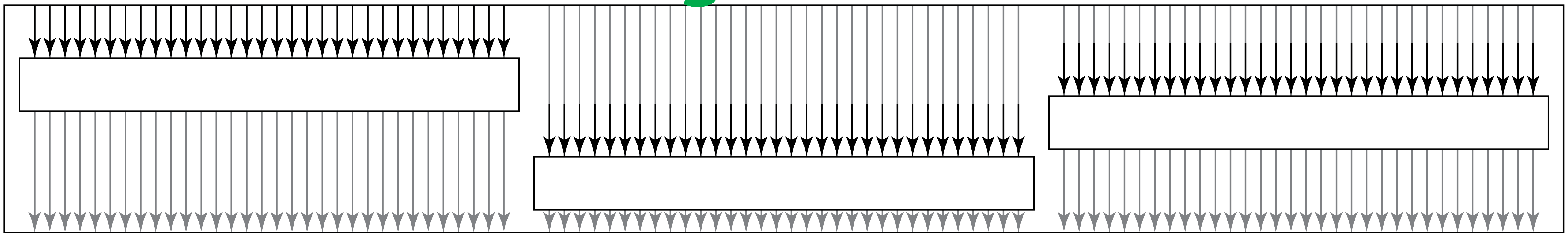
ballot [CC 2.0+] - [Fermi] [Kepler]

```
unsigned int __ballot(int predicate);
```

```
__ballot(predicate):
   Evaluate predicate for all active threads of the warp
   and return an integer whose Nth bit is set if and only
   if predicate evaluates to non-zero for the Nth thread
   of the warp and the Nth thread is active.
```

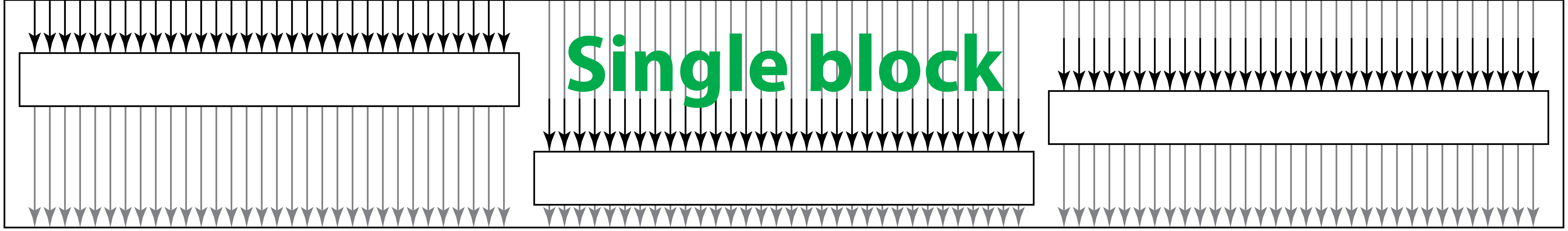


Single block



- hundreds of threads split into warps
- warps execute concurrently
- all waprs live

- Concurrency hazards
- Branching

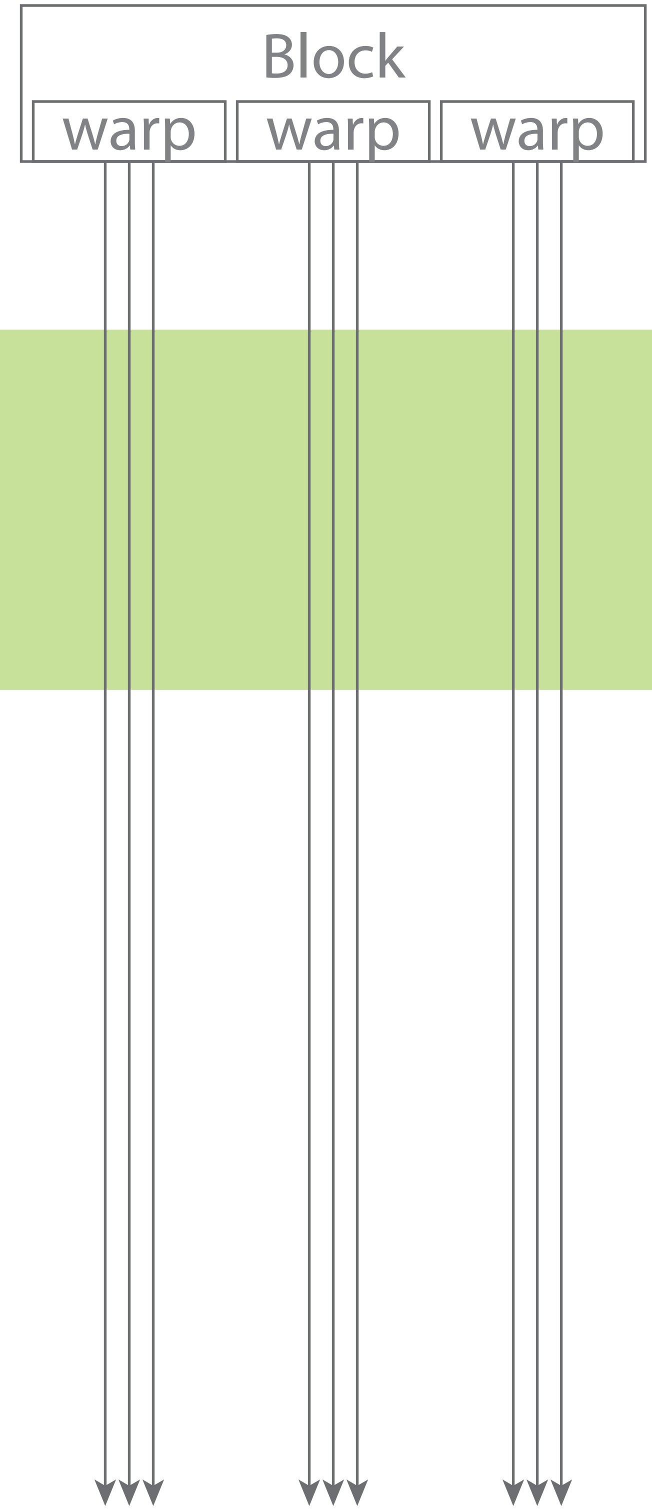


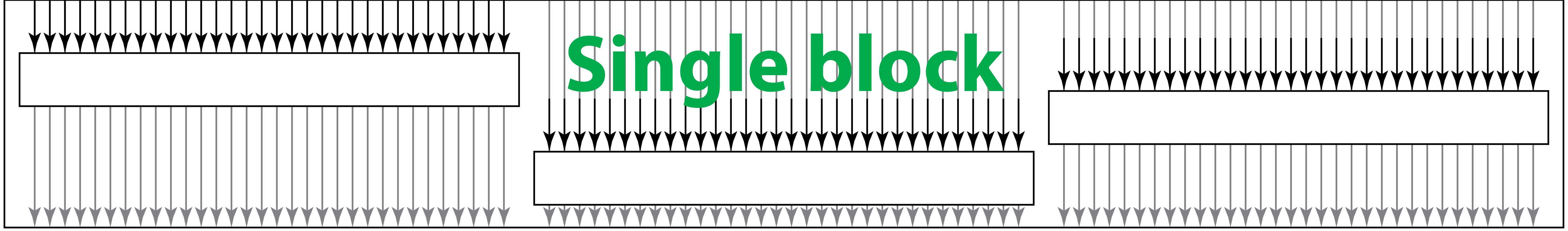

```
__shared__ int shared[3];
shared[warpIdx]=0;
```

shared

0	0	0

```
1: shared[warpIdx]=1;
2: int x = shared[warpIdx+1];
   x == ?
```

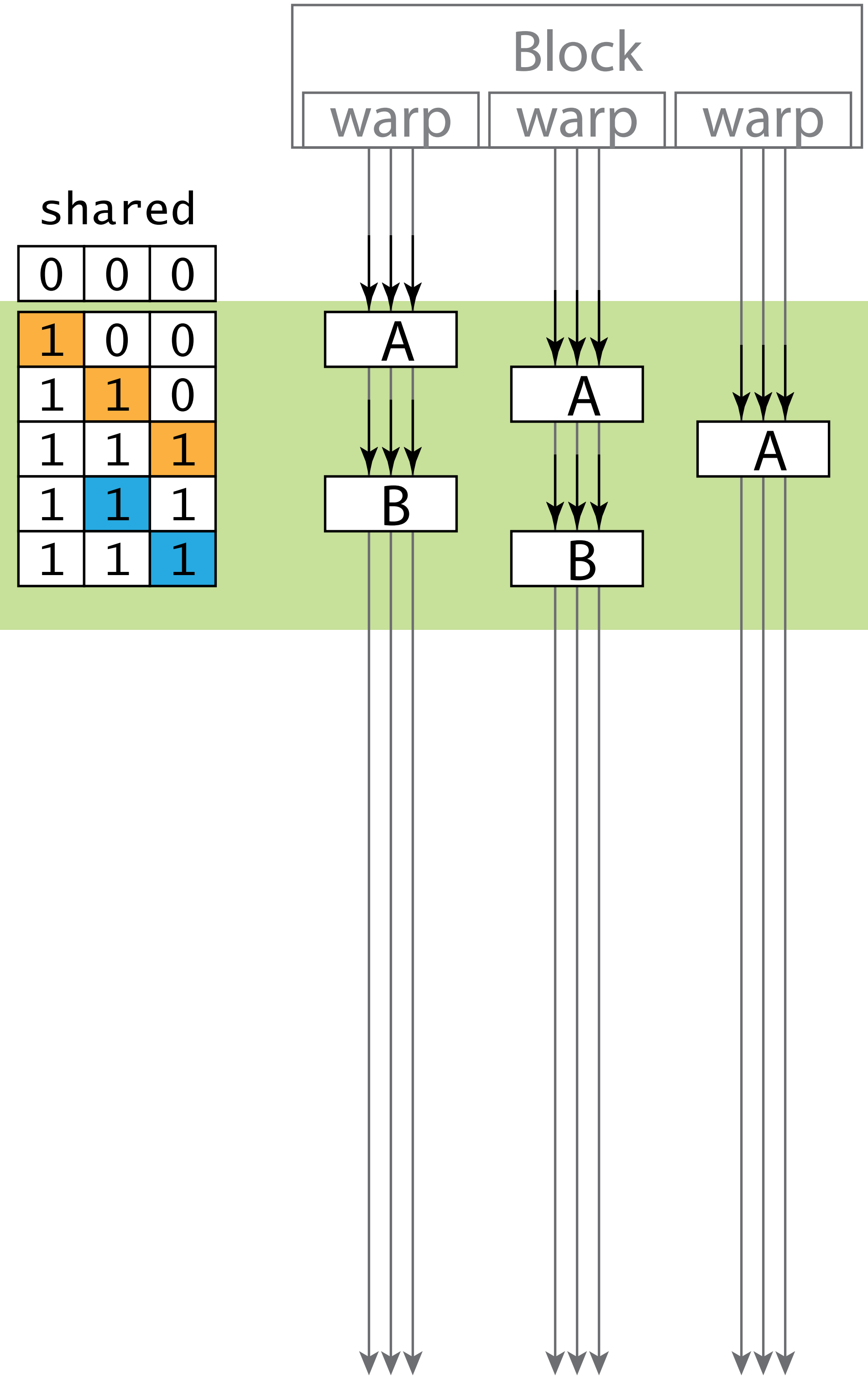


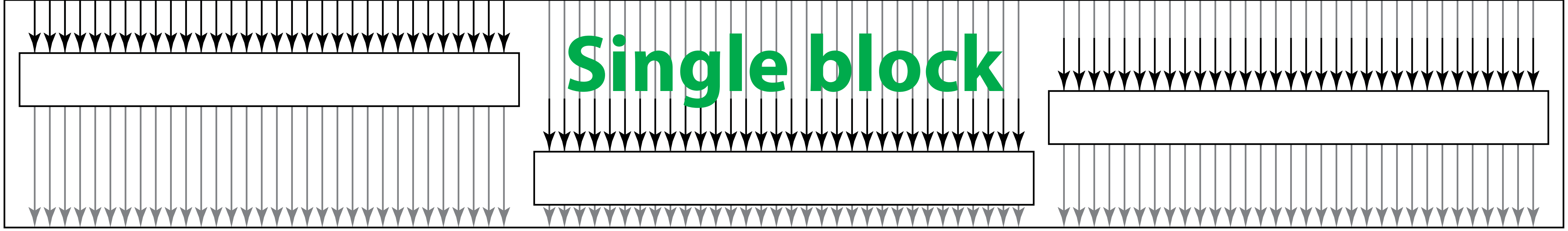



```
__shared__ int shared[3];
shared[warpIdx]=0;
```

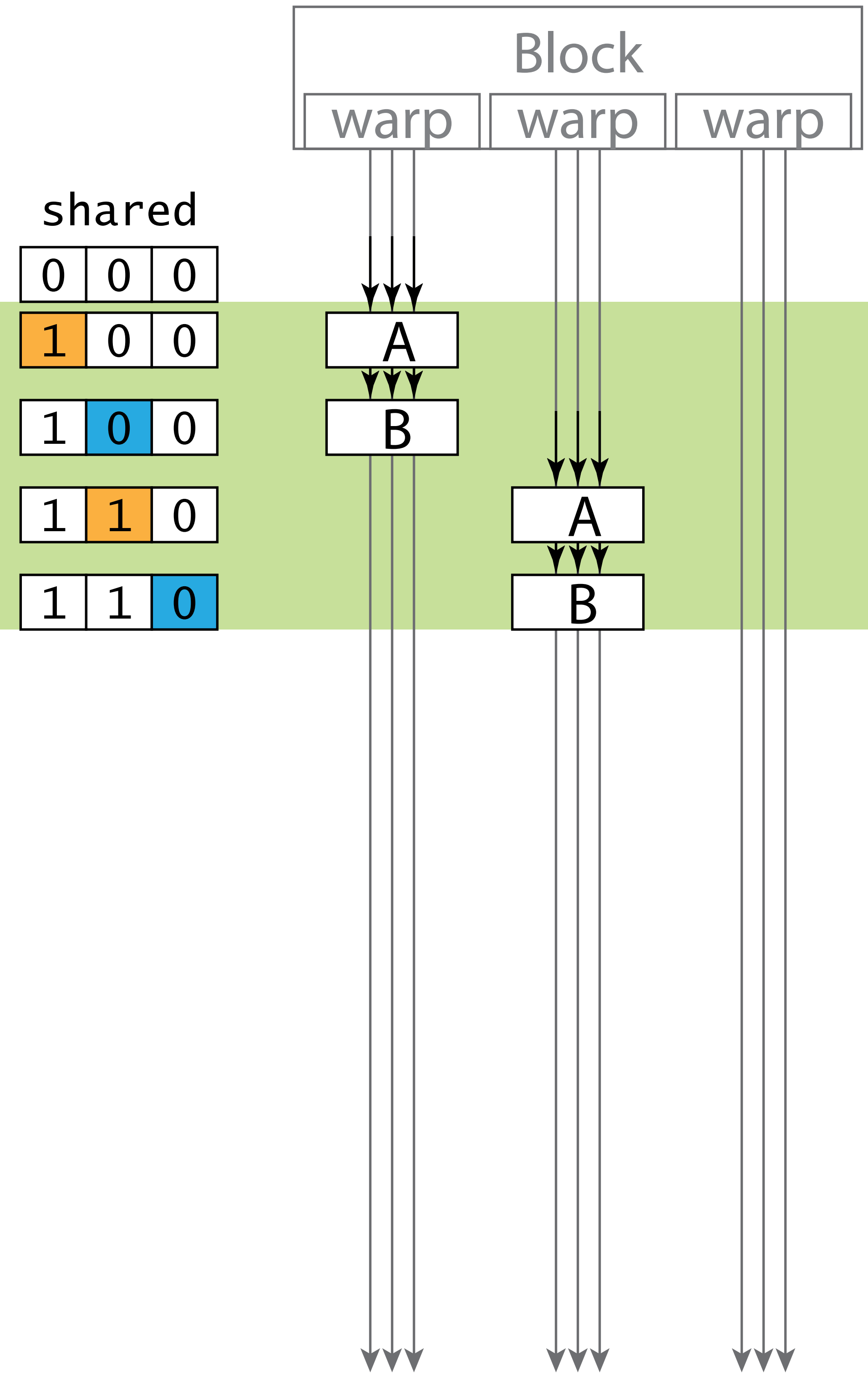
```
A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
```

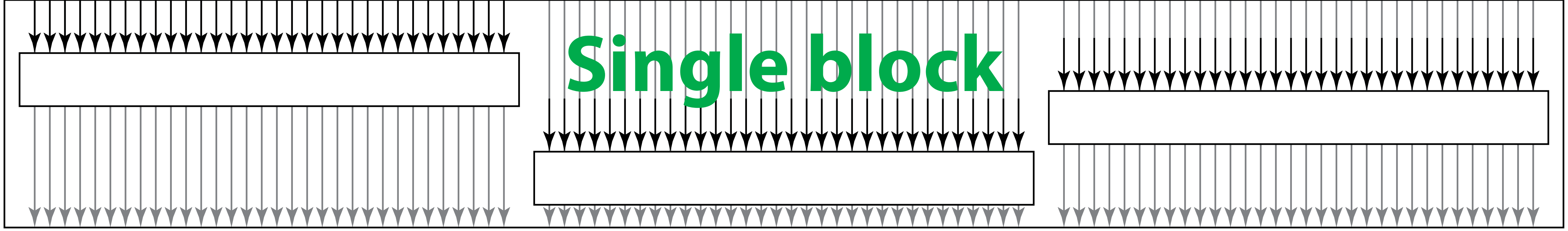
x == 1






```
__shared__ int shared[3];
shared[warpIdx]=0;

A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
              x == 1
              x == 0
```



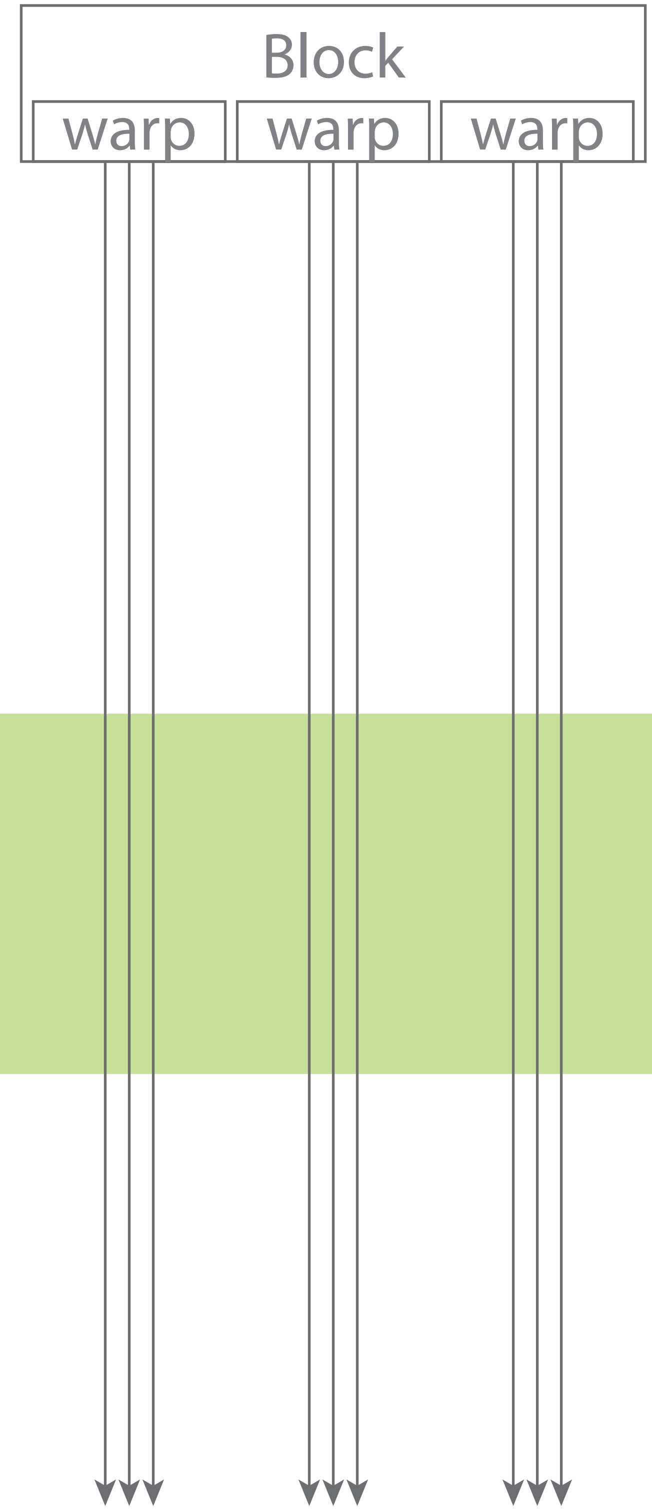


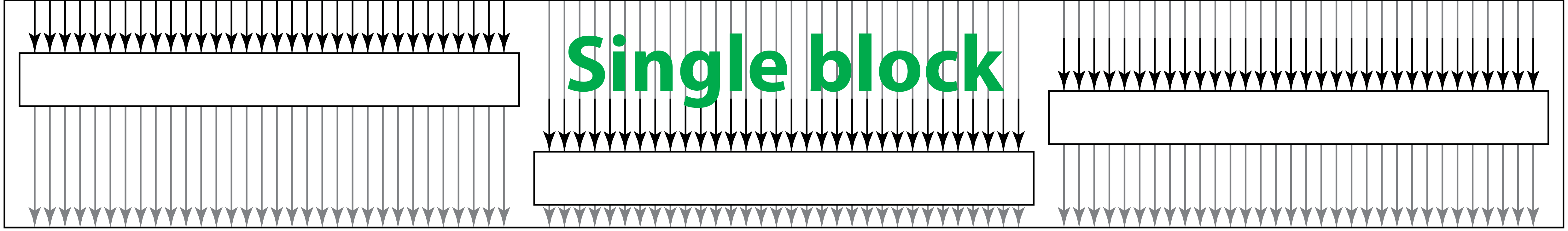

```
__shared__ int shared[3];
shared[warpIdx]=0;

A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
            x == 1
            x == 0
```

shared

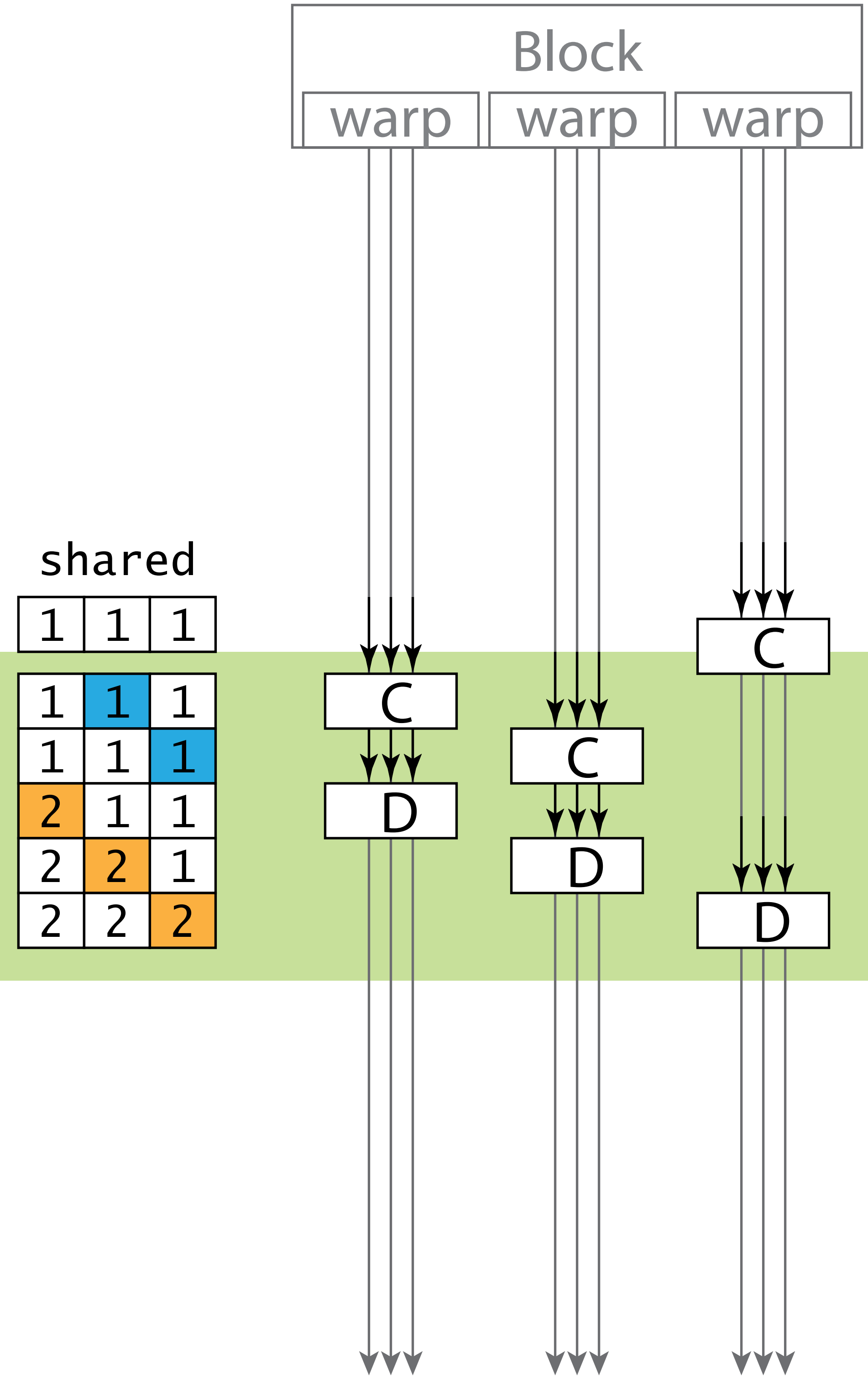
1	1	1

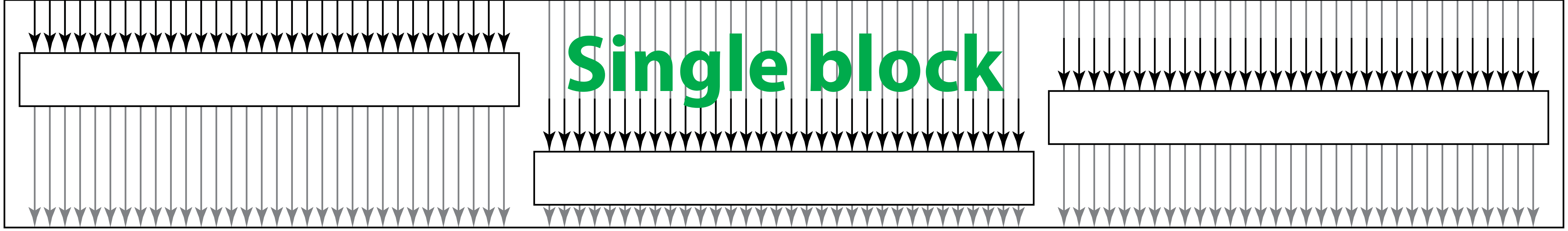
```
C: int y = shared[warpIdx+1];
D: shared[warpIdx]=2;
   y == ?
```



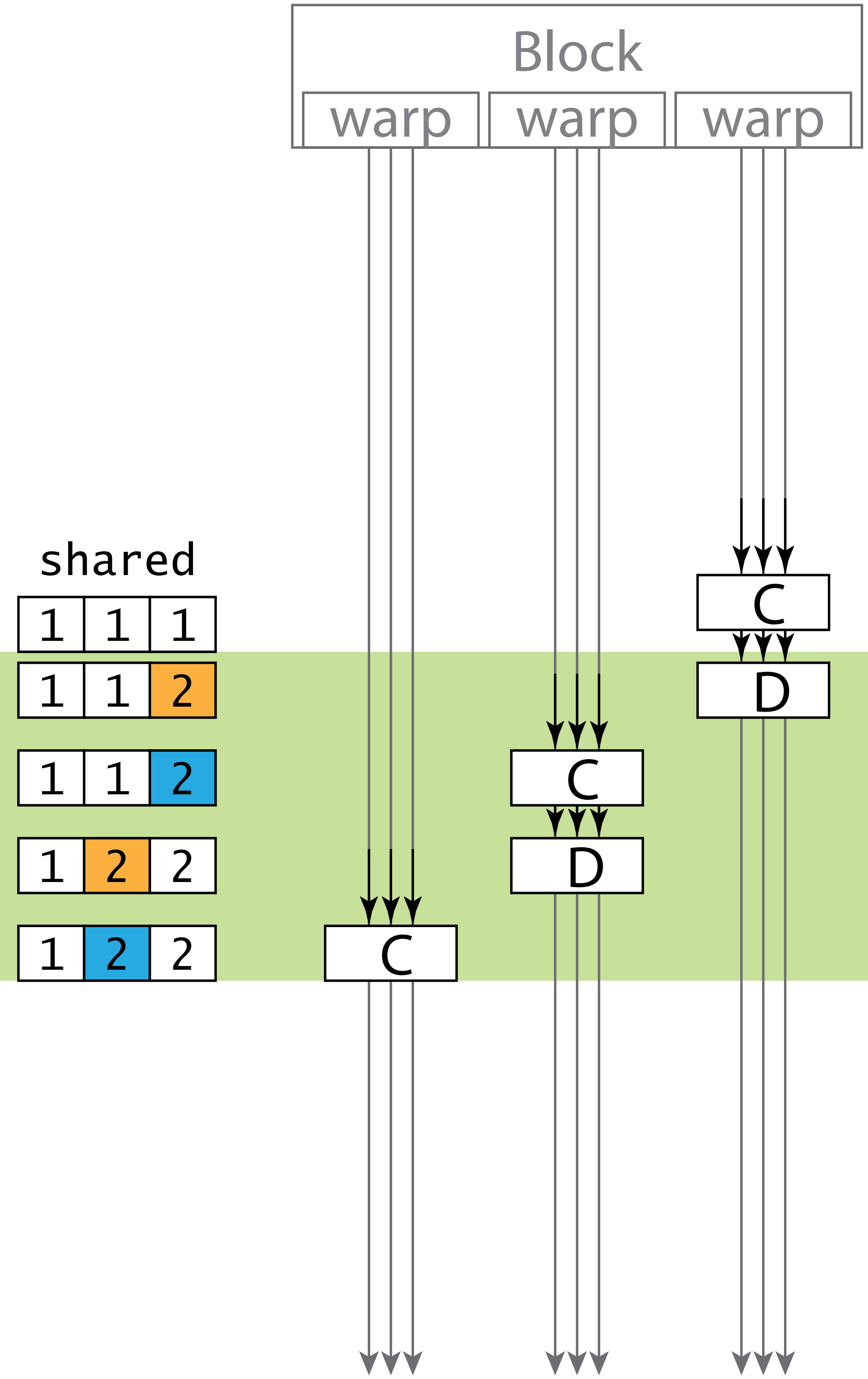


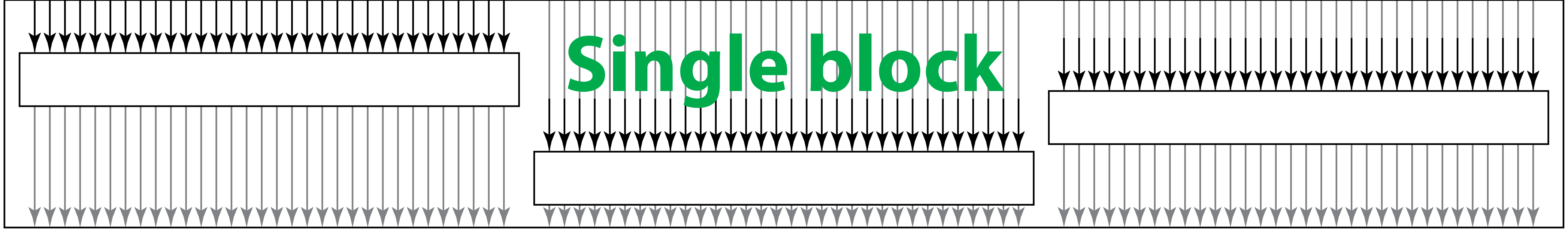

```
__shared__ int shared[3];
shared[warpIdx]=0;



A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
             x == 1
             x == 0


C: int y = shared[warpIdx+1];
D: shared[warpIdx]=2;
   y == ?
             y == 1
```






```
__shared__ int shared[3];
shared[warpIdx]=0;

A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
               x == 1
               x == 0

C: int y = shared[warpIdx+1];
D: shared[warpIdx]=2;
   y == ?
               y == 1
               y == 2
```

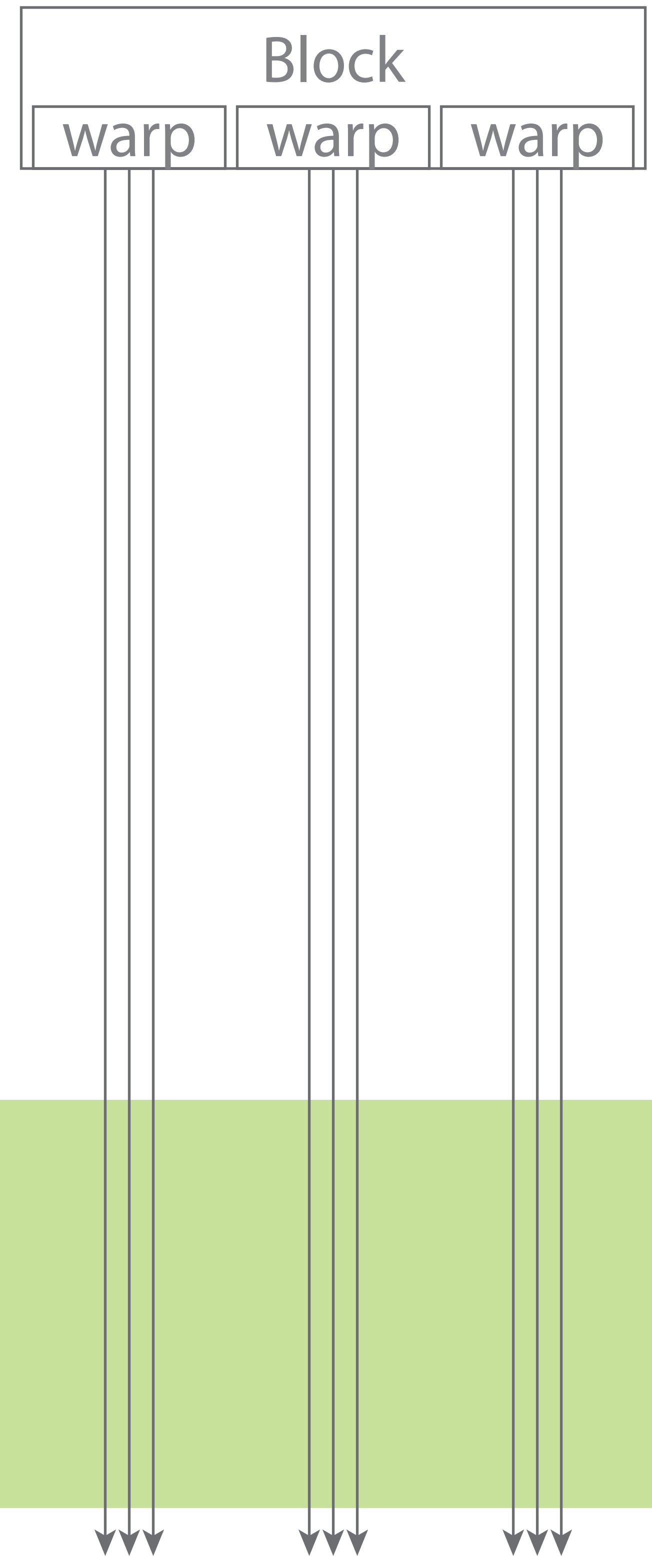


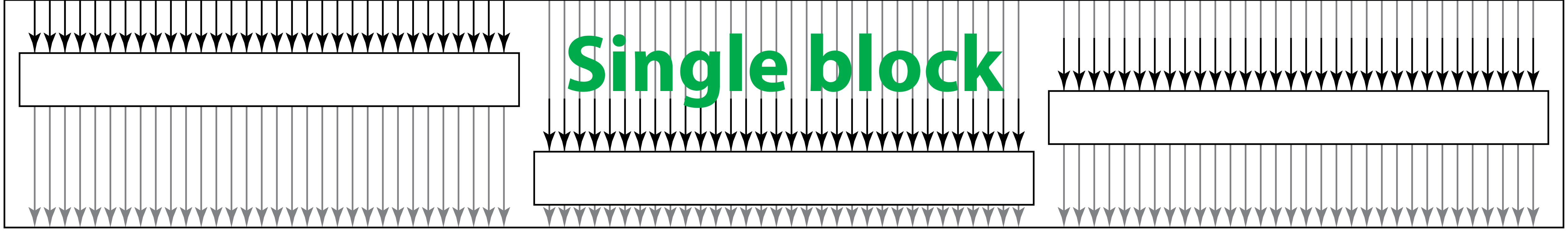



```
__shared__ int shared[3];
shared[warpIdx]=0;


A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
             x == 1
             x == 0


C: int y = shared[warpIdx+1];
D: shared[warpIdx]=2;
   y == ?
             y == 1
             y == 2


E: shared[warpIdx]=3;
F: shared[warpIdx+1]=4;
   shared == ?
```

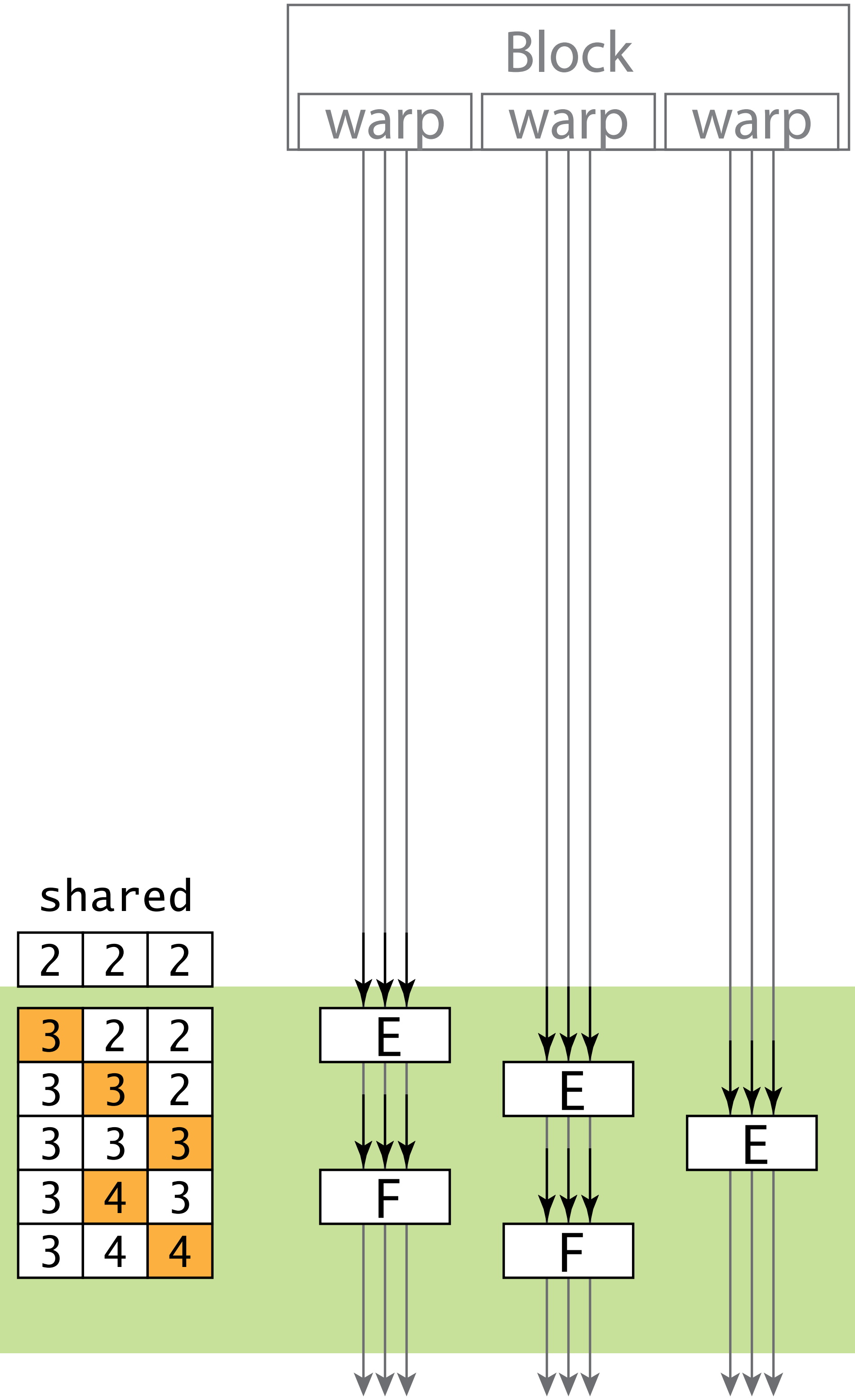
shared

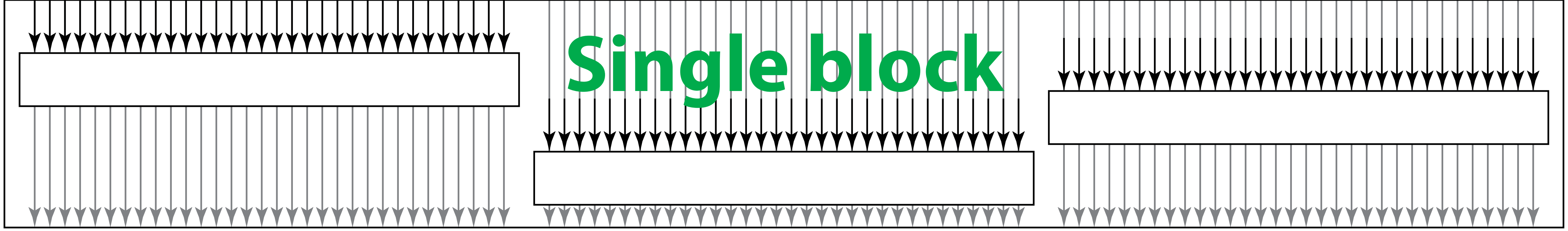
2	2	2



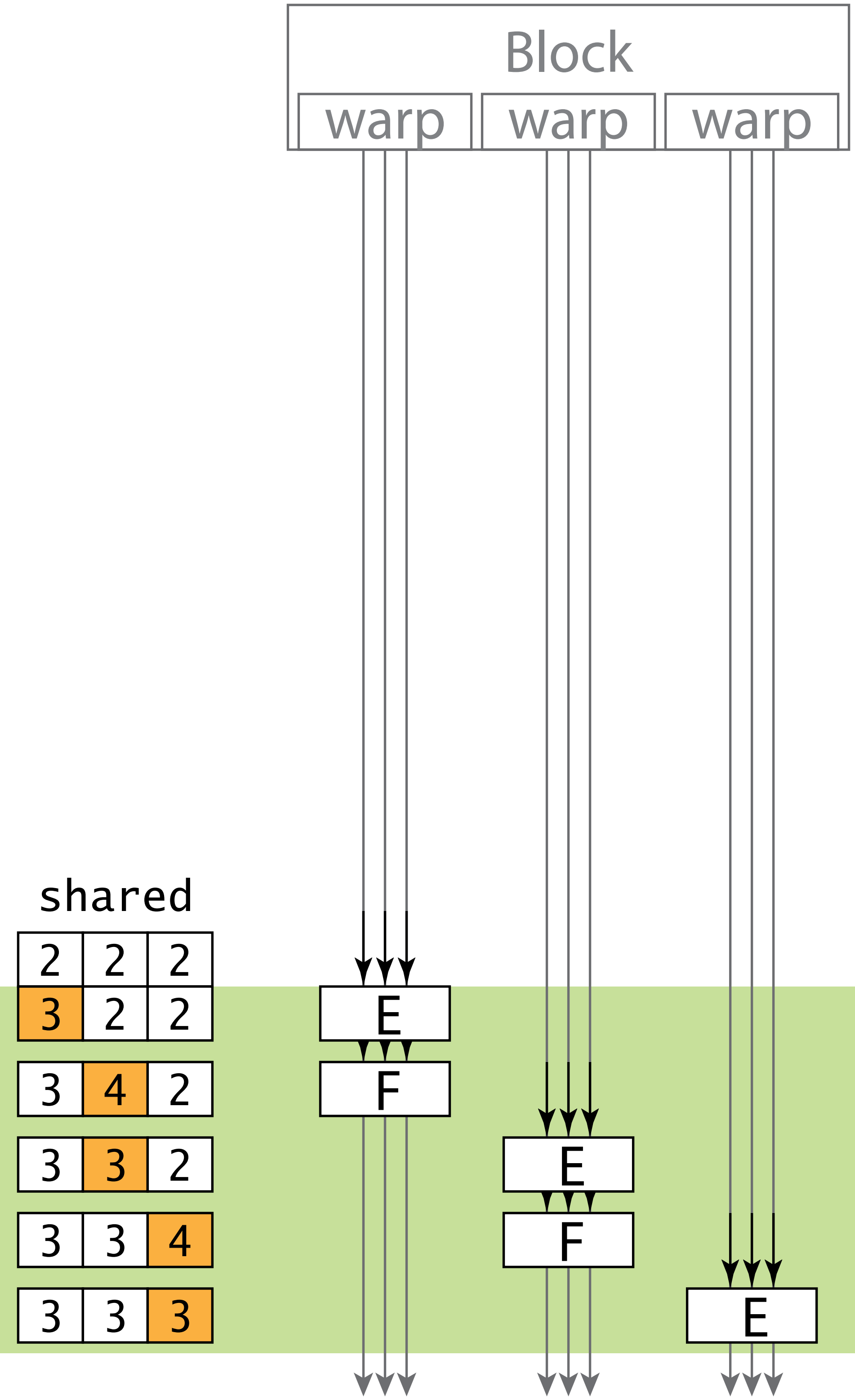


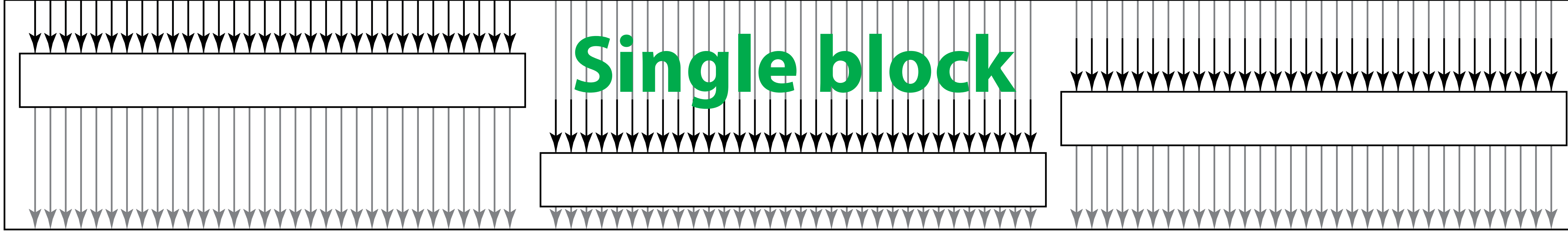

```
__shared__ int shared[3];
shared[warpIdx]=0;


A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
             x == 1
             x == 0


C: int y = shared[warpIdx+1];
D: shared[warpIdx]=2;
   y == ?
             y == 1
             y == 2


E: shared[warpIdx]=3;
F: shared[warpIdx+1]=4;
   shared == ?
             shared == {4, 4, 3}
```



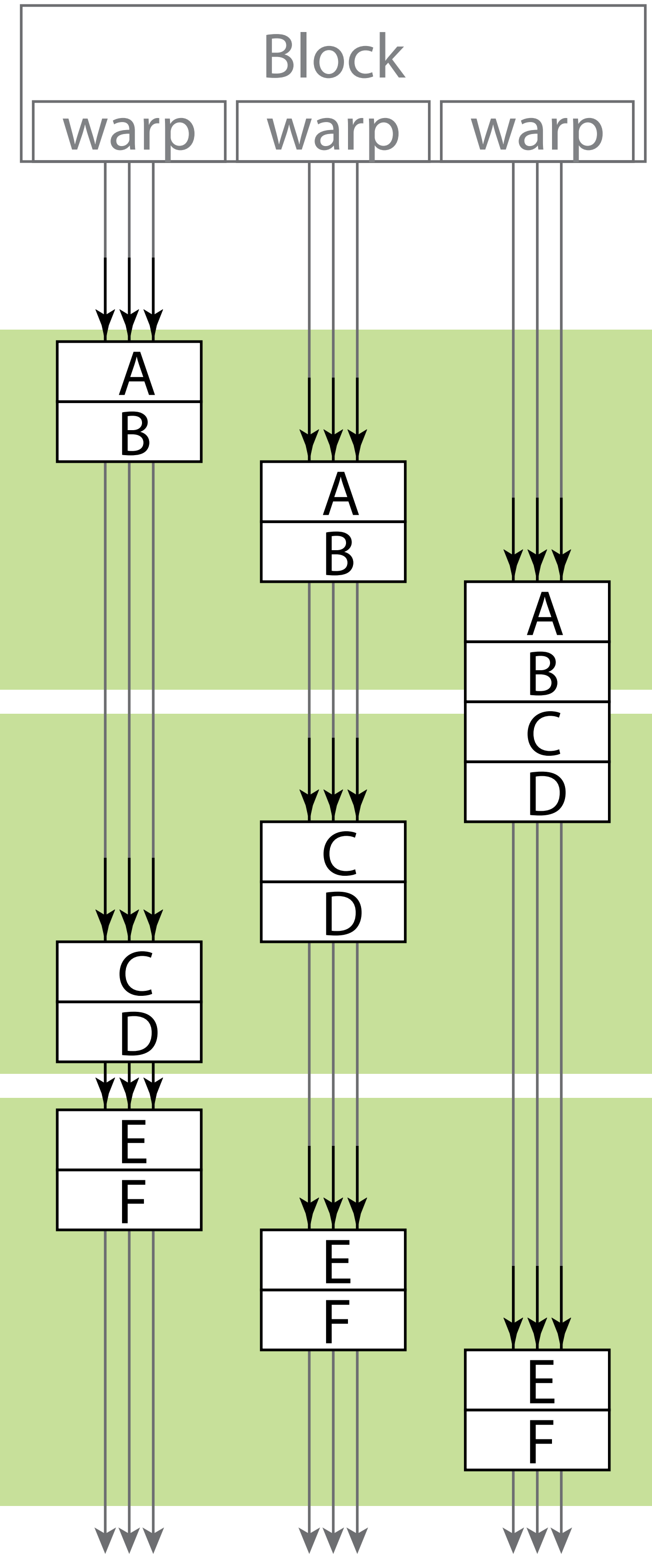



```
__shared__ int shared[3];
shared[warpIdx]=0;


A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
             x == 1
             x == 0


C: int y = shared[warpIdx+1];
D: shared[warpIdx]=2;
   y == ?
             y == 1
             y == 2


E: shared[warpIdx]=3;
F: shared[warpIdx+1]=4;
   shared == ?
             shared == {4, 4, 3}
             shared == {3, 3, 3}
```






```
__shared__ int shared[3];
shared[warpIdx]=0;
```



RaW

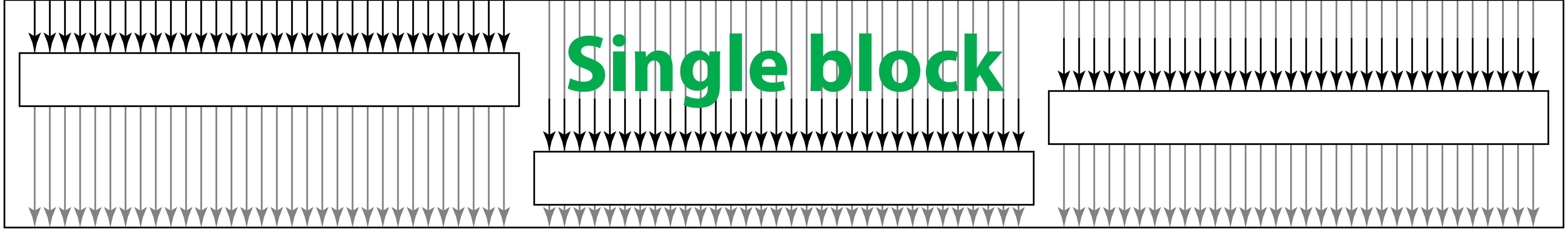
```
A: shared[warpIdx]=1;
B: int x = shared[warpIdx+1];
   x == ?
              x == 1
              x == 0
```

WaR

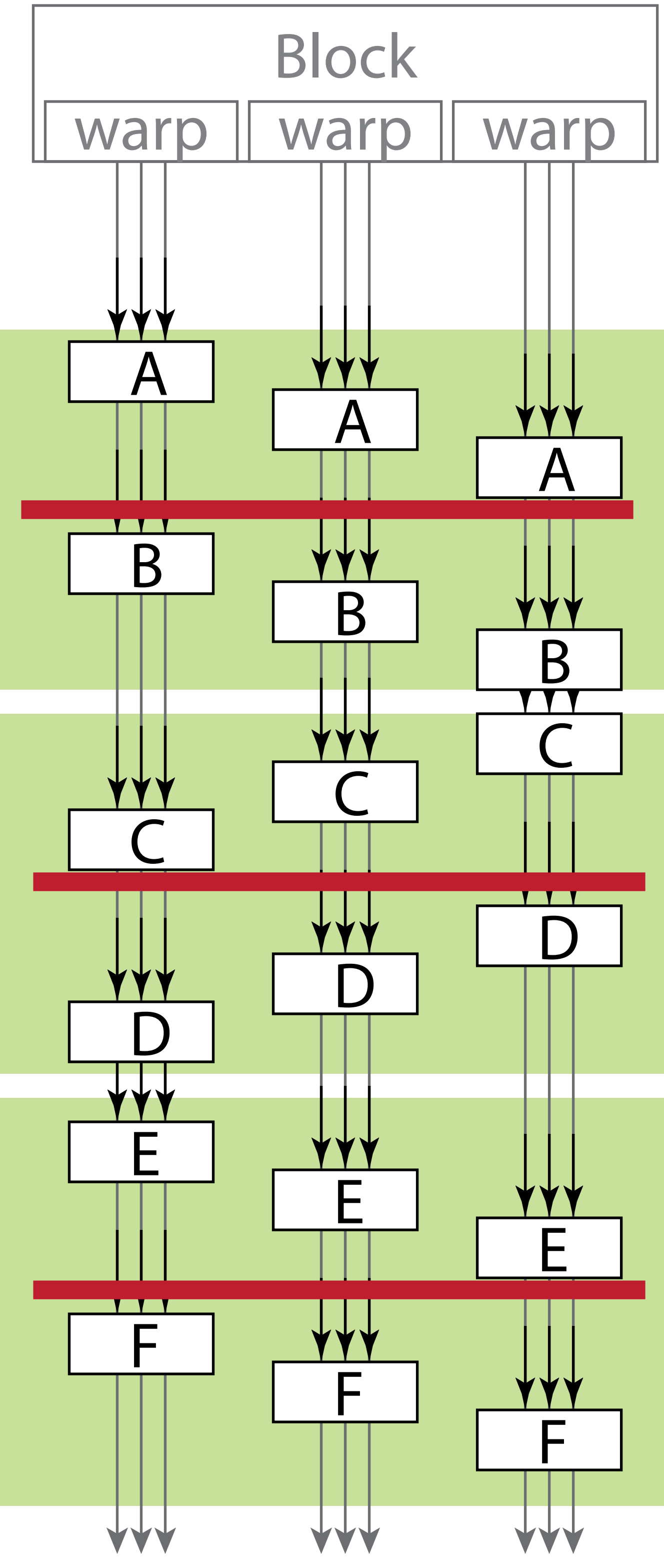
```
C: int y = shared[warpIdx+1];
D: shared[warpIdx]=2;
   y == ?
              y == 1
              y == 2
```

WaW

```
E: shared[warpIdx]=3;
F: shared[warpIdx+1]=4;
   shared == ?

              shared == {4, 4, 3}
              shared == {3, 3, 3}
```



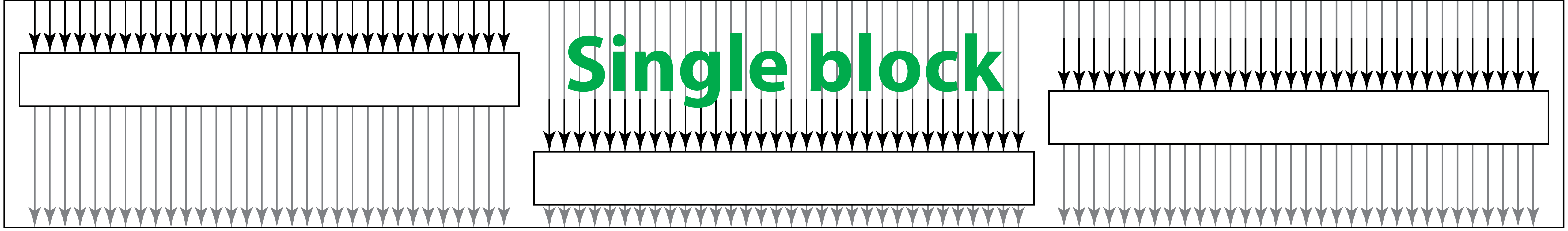

```
__shared__ int shared[3];
shared[warpIdx]=0;
```



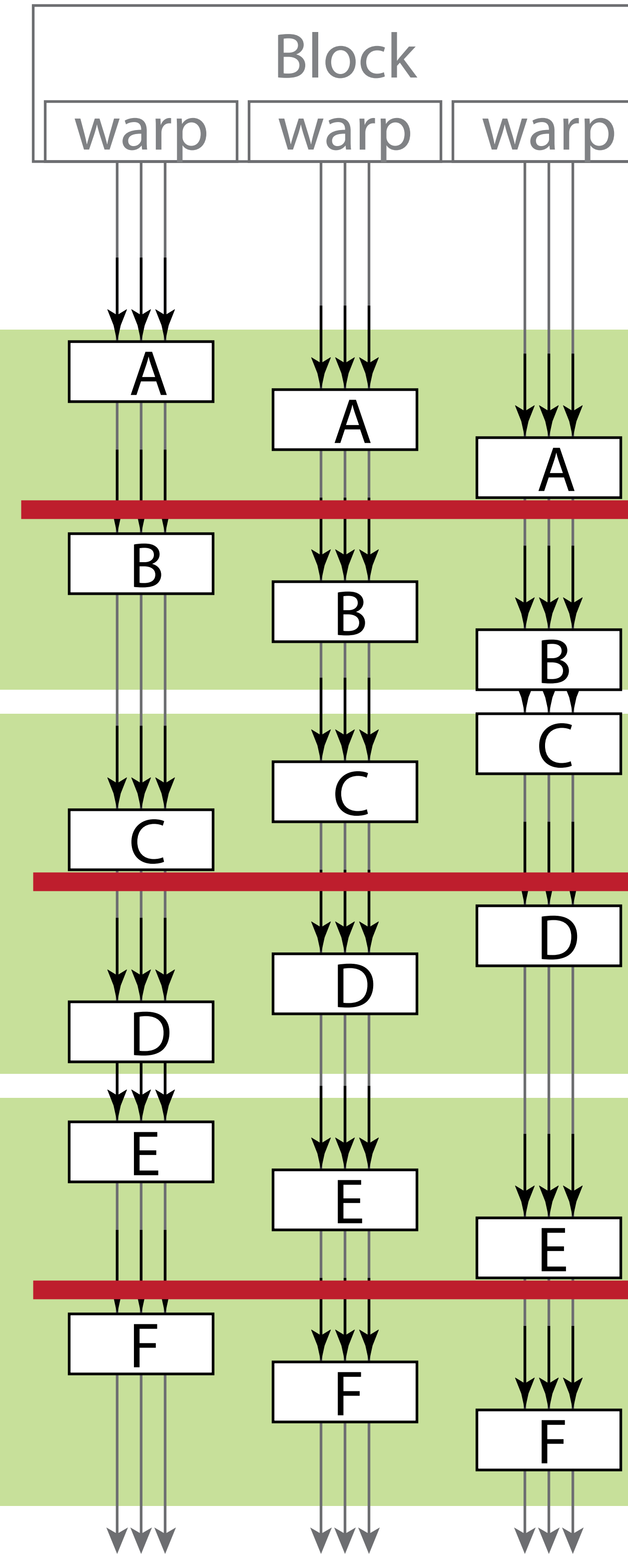

```
A: shared[warpIdx]=1;
   __syncthreads();
B: int x = shared[warpIdx+1];
   x == ?
             x == 1
```

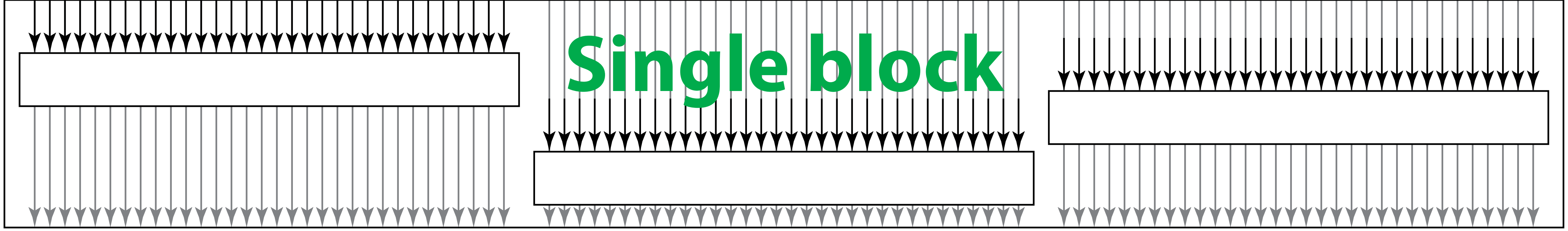
```
C: int y = shared[warpIdx+1];
   __syncthreads();
D: shared[warpIdx]=2;
   y == ?
             y == 1
```

```
E: shared[warpIdx]=3;
   __syncthreads();
F: shared[warpIdx+1]=4;
   shared == ?

             shared == {4, 4, 3}
```




```
    __shared__ int shared[3];
    shared[warpIdx]=0;

A:  shared[warpIdx]=1;
    __syncthreads();
B:  int x = shared[warpIdx+1];

C:  int y = shared[warpIdx+1];
    __syncthreads();
D:  shared[warpIdx]=2;

E:  shared[warpIdx]=3;
    __syncthreads();
F:  shared[warpIdx+1]=4;
```






```
    __shared__ int shared[3];
    shared[warpIdx]=0;
```

WaW but same warp

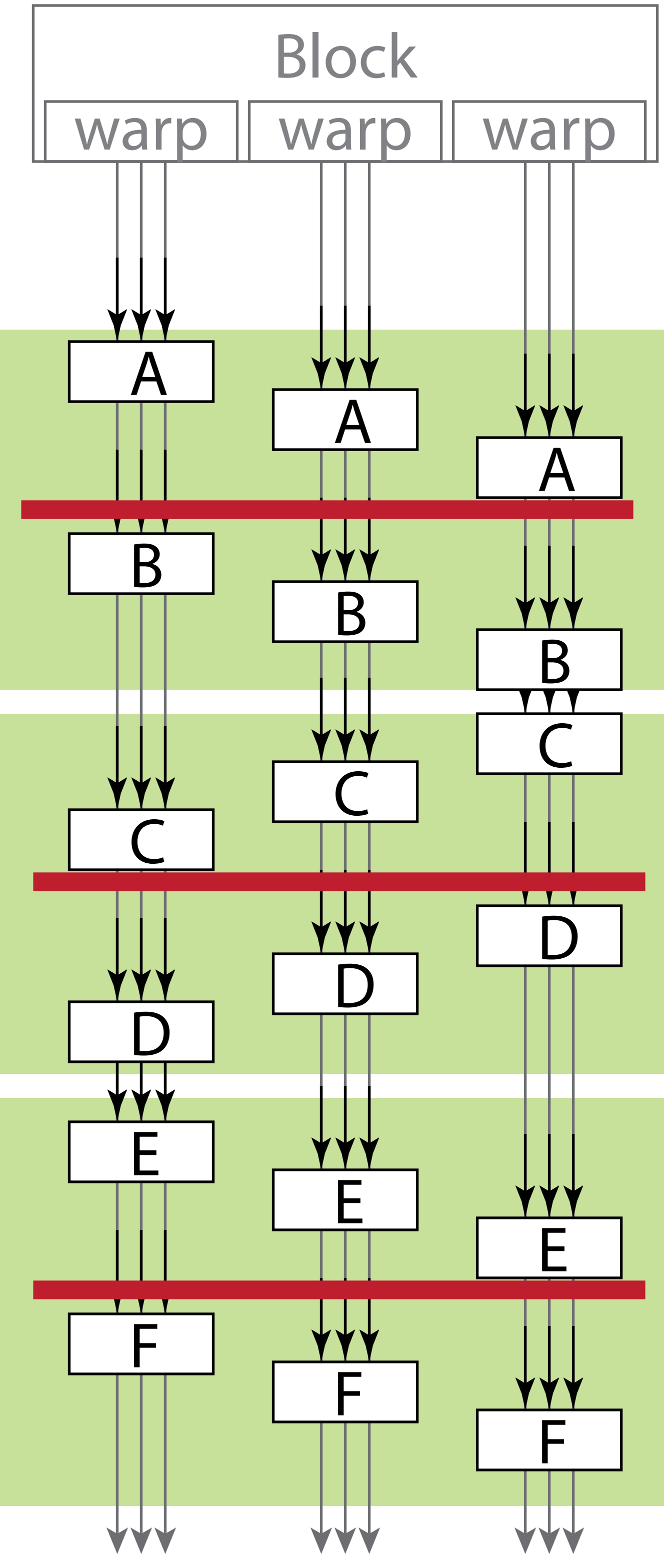
```
A:  shared[warpIdx]=1;
    __syncthreads();
B:  int x = shared[warpIdx+1];
```

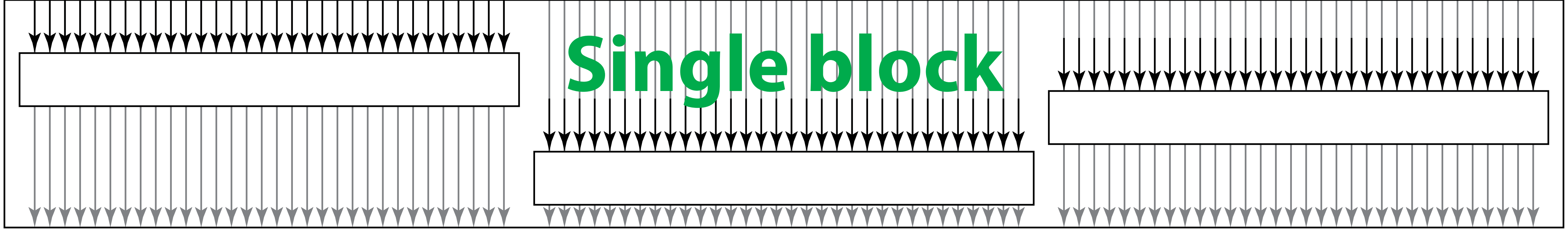
RaR and same warp

```
C:  int y = shared[warpIdx+1];
    __syncthreads();
D:  shared[warpIdx]=2;
```

WaW but same warp

```
E:  shared[warpIdx]=3;
    __syncthreads();
F:  shared[warpIdx+1]=4;
```

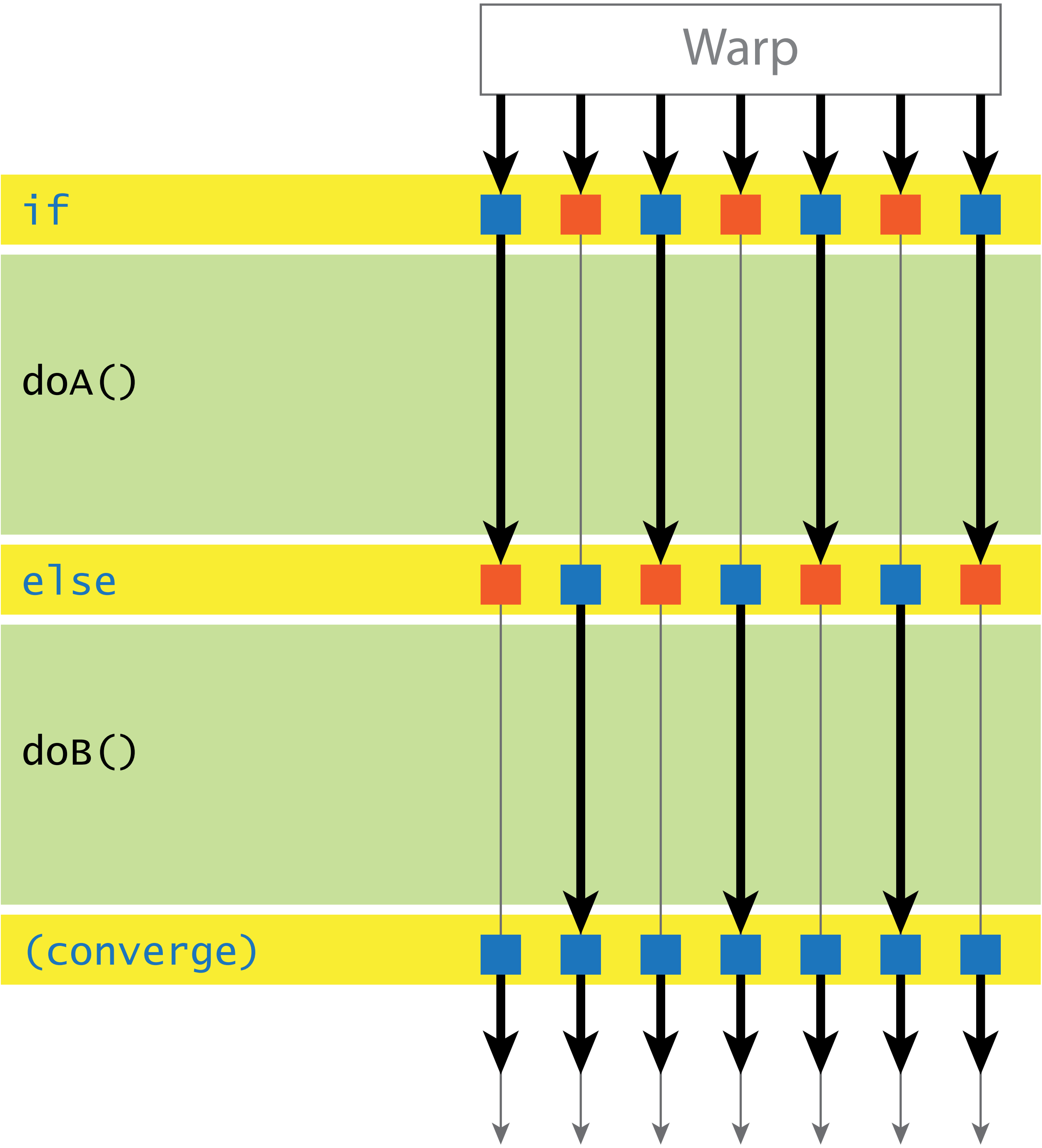


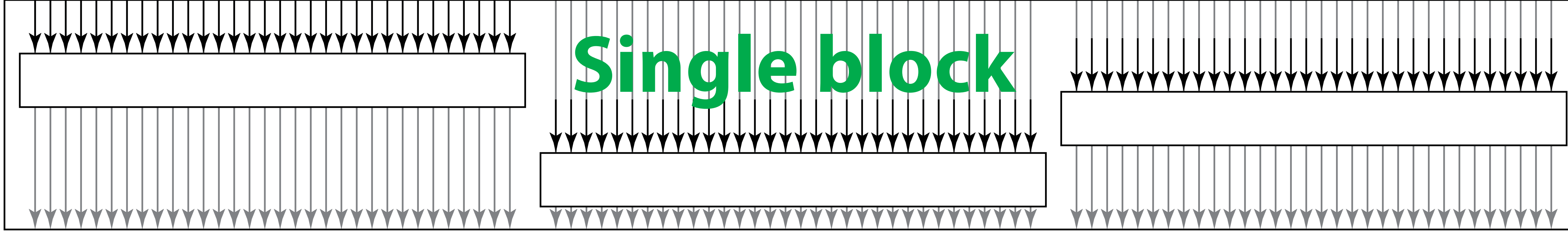


Recall for single warp:

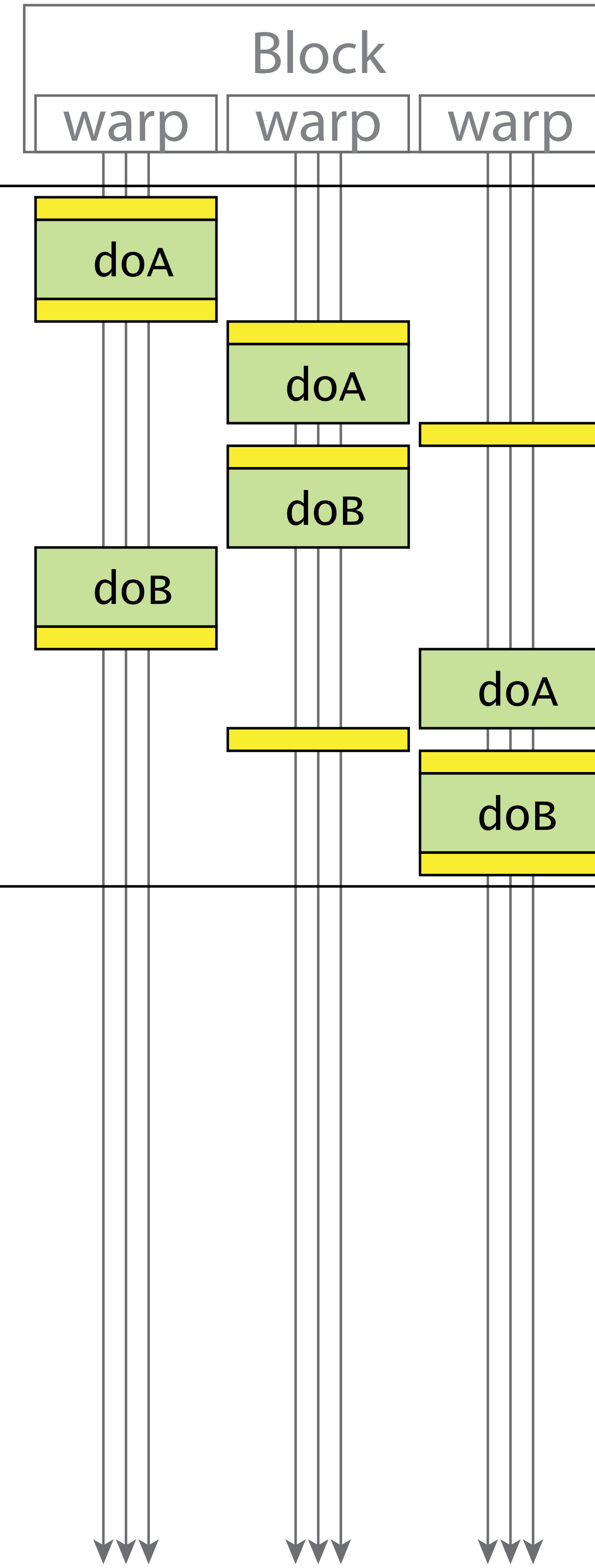
warp中第偶数位置的线程

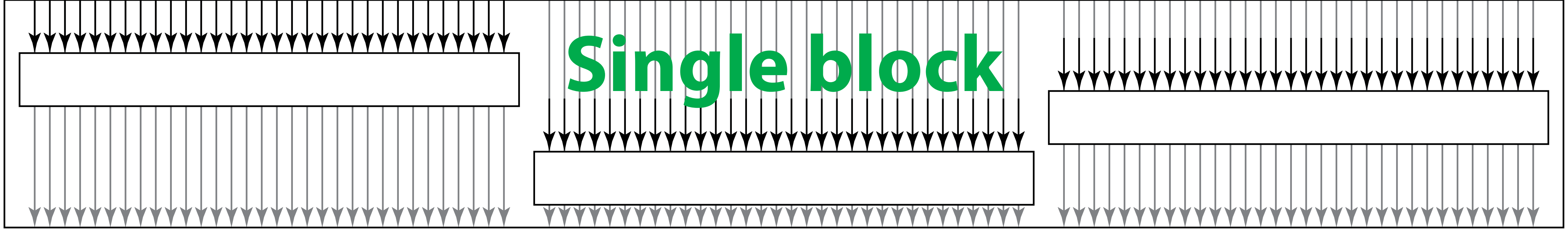
```
if (laneIdx % 2 == 0) {
  doA();
} else {
  doB();
}
```

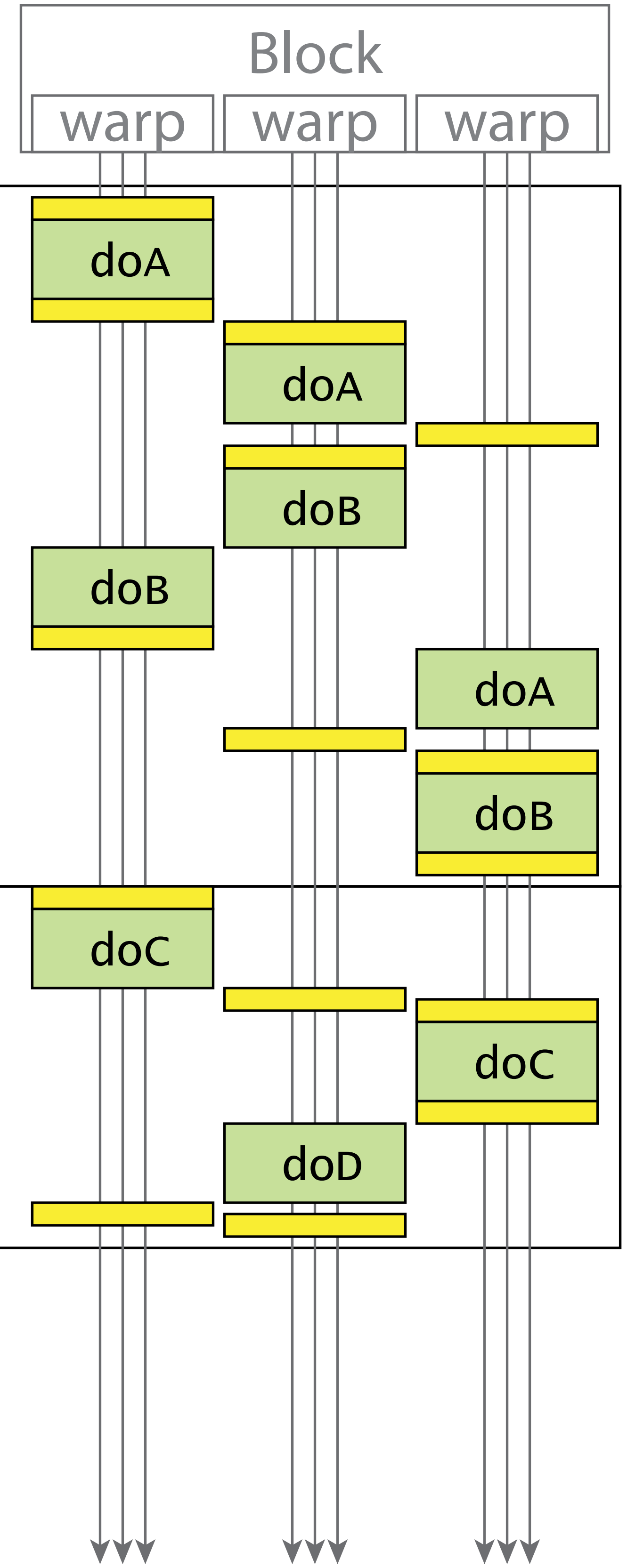





```
if (laneIdx % 2 == 0) {
  doA();
} else {
  doB();
}
```



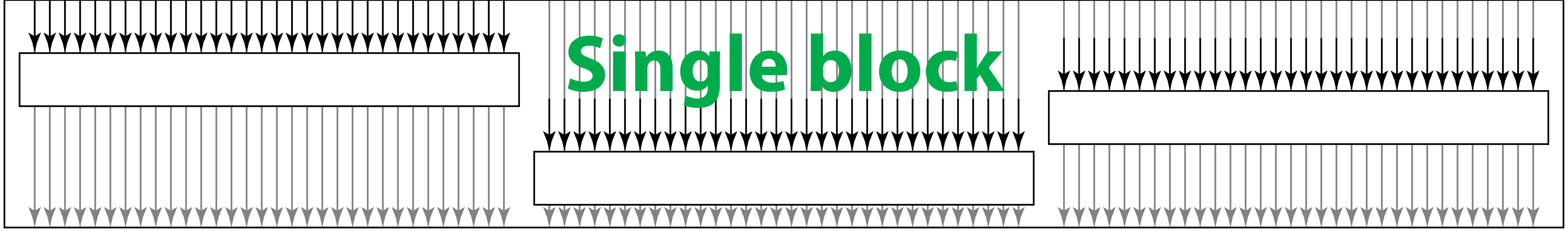





```
if (laneIdx % 2 == 0) {
  doA();
} else {
  doB();
}
```

第偶数warp的线程中

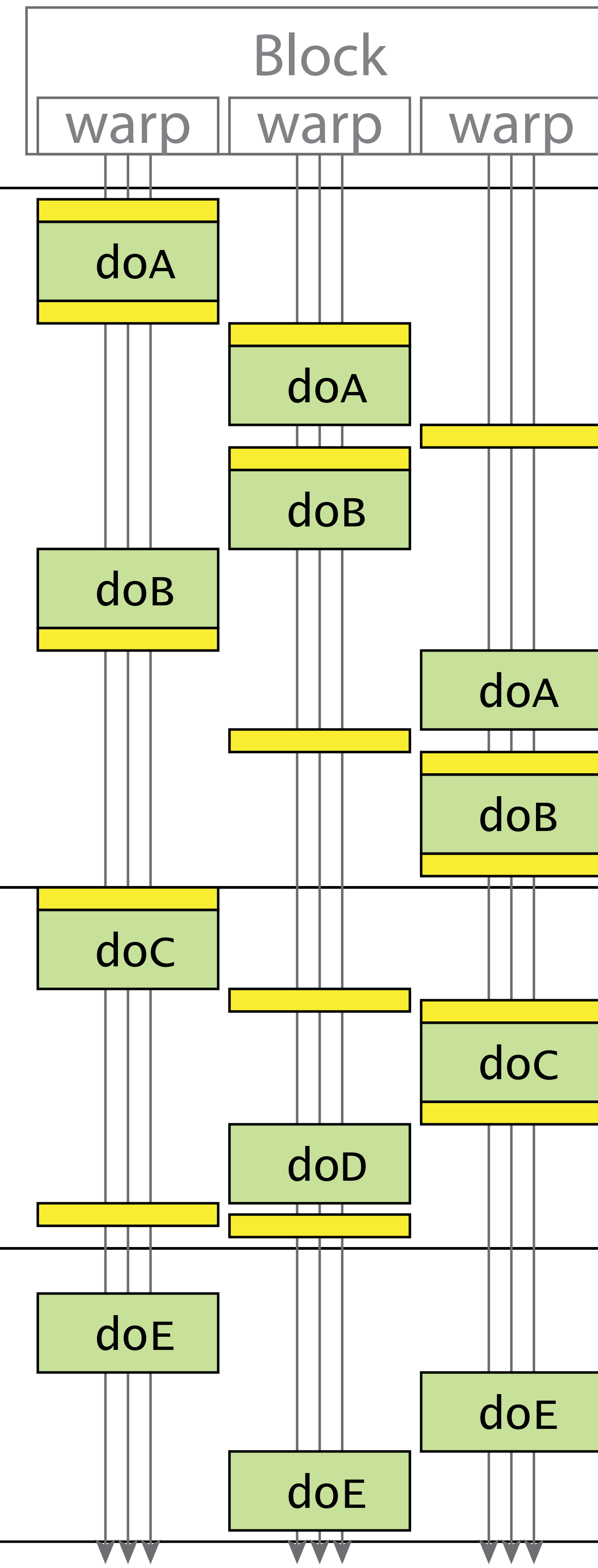
```
if (warpIdx % 2 == 0) {
  doC();
} else {
  doD();
}
```

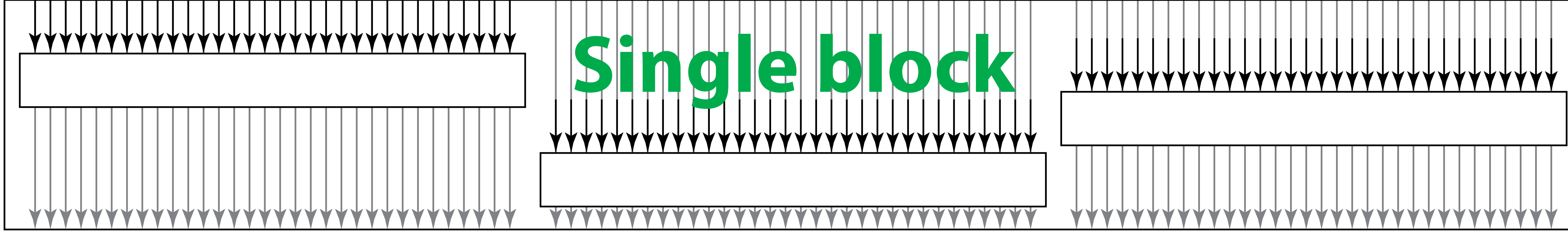



```
if (laneIdx % 2 == 0) {
  doA();
} else {
  doB();
}
```

```
if (warpIdx % 2 == 0) {
  doC();
} else {
  doD();
}
```

```
if (blockIdx.x % 2 == 0) {
  doE();
} else {
  doF();
}
```






```
if (laneIdx % 2 == 0) {
  doA();
} else {
  doB();
}
```

warp-divergent `if`

(performance loss)

```
if (warpIdx % 2 == 0) {
  doC();
} else {
  doD();
}
```

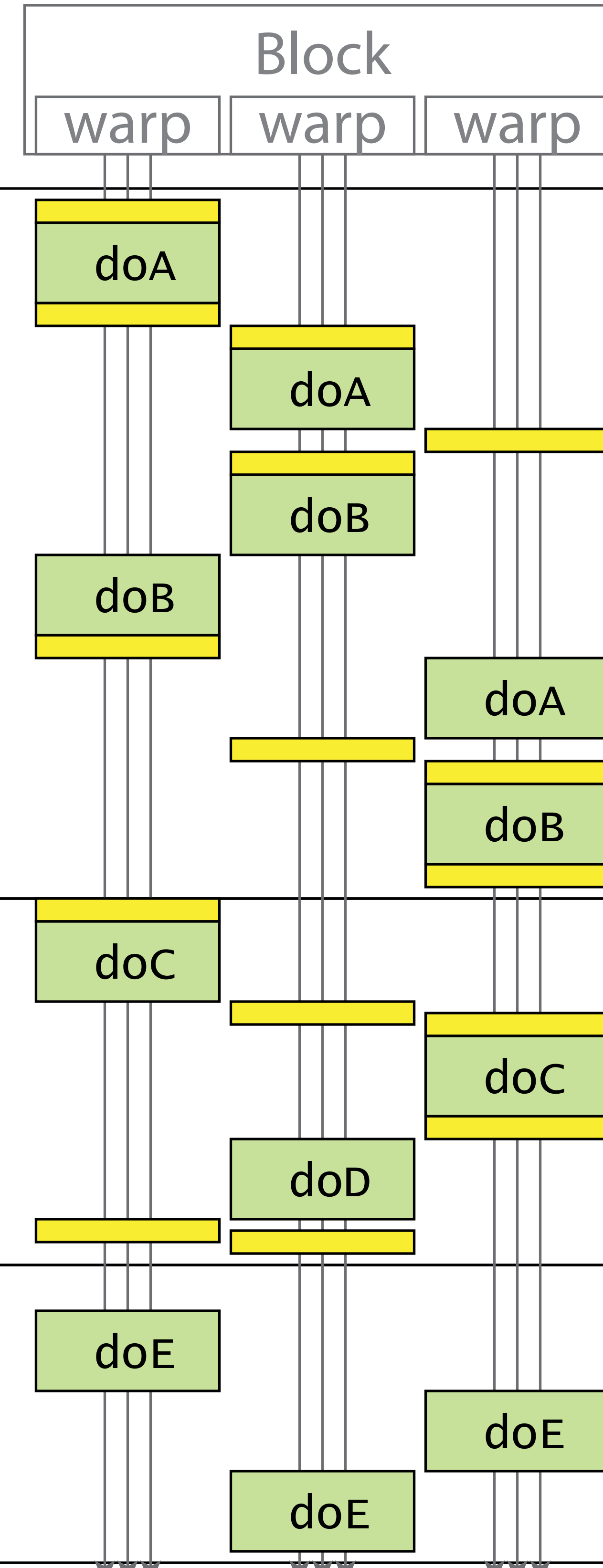
warp-uniform `if`

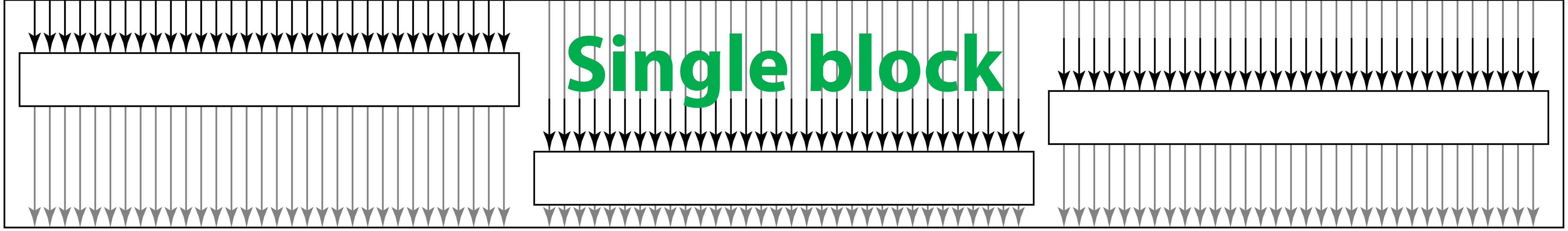
(little performance loss)

```
if (blockIdx.x % 2 == 0) {
  doE();
} else {
  doF();
}
```

block-uniform `if`

(no performance loss)






```
if (laneIdx % 2 == 0) {
   doA();
   __syncthreads();
} else {
   doB();
}
```

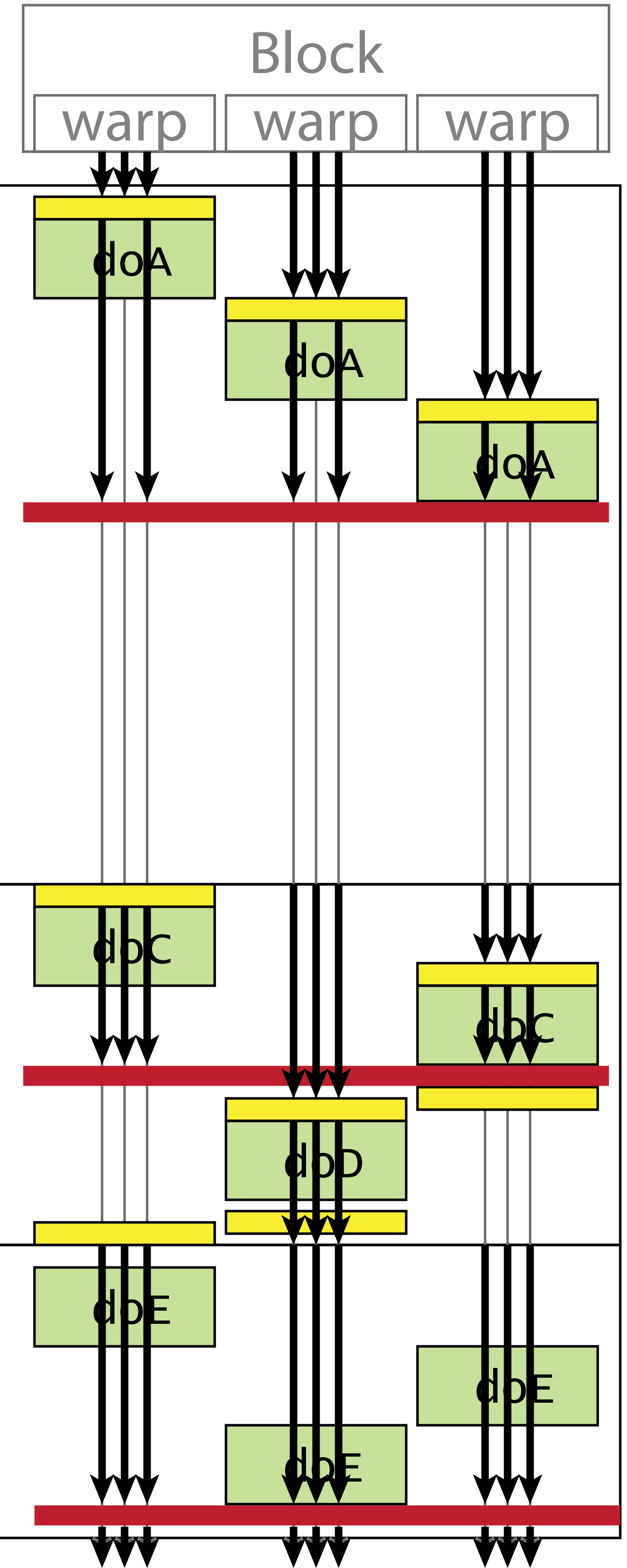
✗ warp-divergent `if`
(performance loss)

```
if (warpIdx % 2 == 0) {
   doC();
   __syncthreads();
} else {
   doD();
}
```

✗ warp-uniform `if`
(little performance loss)

```
if (blockIdx.x % 2 == 0) {
   doE();
   __syncthreads();
} else {
   doF();
}
```

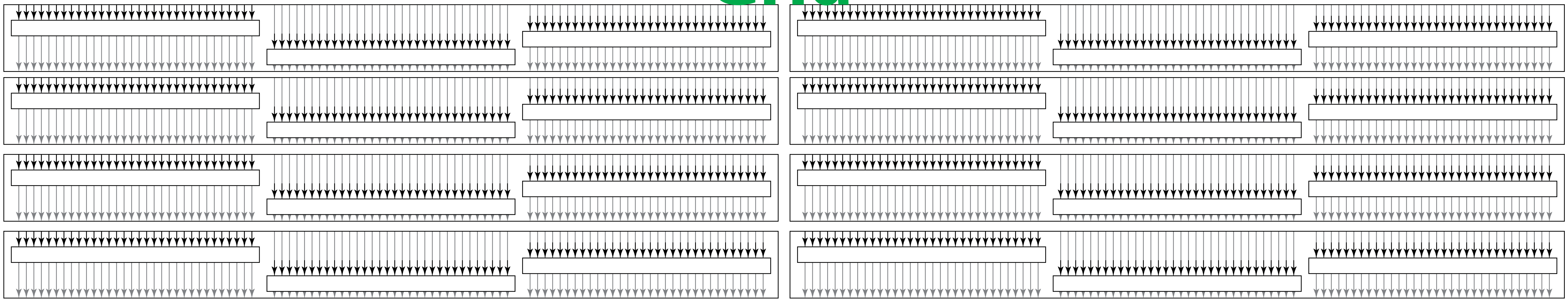
✓ block-uniform `if`
(no performance loss)



Grid

- thousands of threads split into blocks
- blocks execute independently
- some blocks may not be live

Grid



some blocks may not be live

⇓

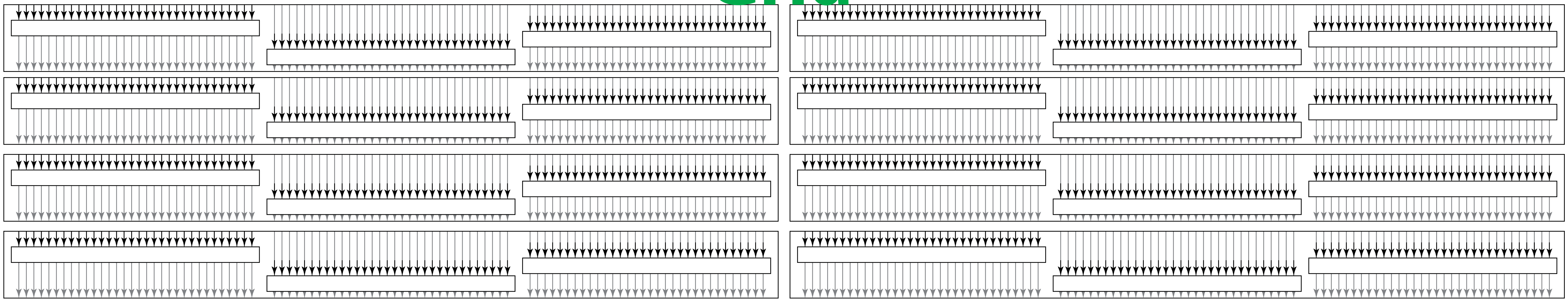
blocks cannot synchronize

⇓

concurency hazards unavoidable

- write to different array that you read from
- atomic on global memory
- terminate kernel, start new one

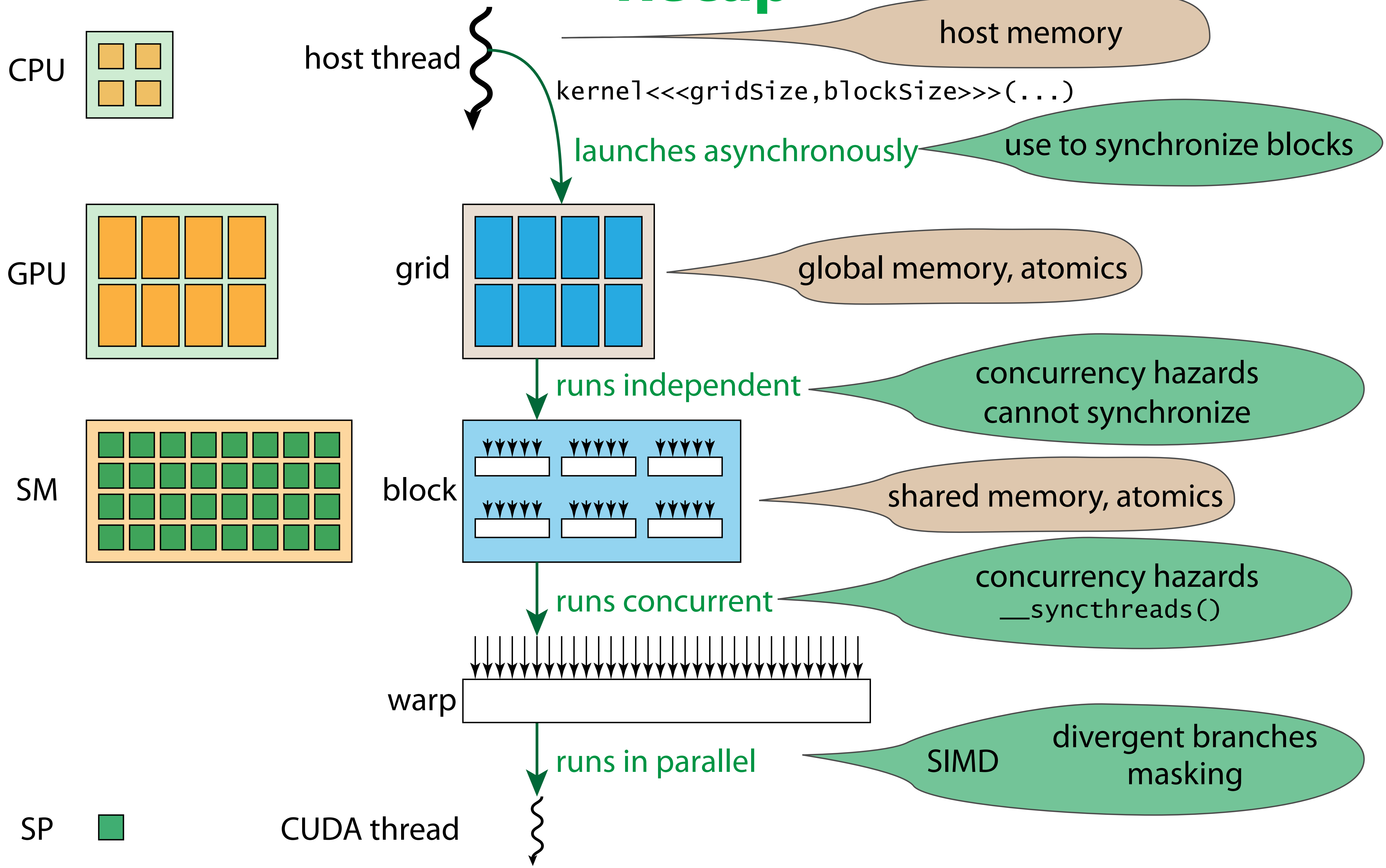
Grid



Terminate kernel, start new one

- Go back to CPU [CC 3.0-]
- Launch sub-kernels [CC 3.5+] new Kepler (2012/2013)

Recap



CPU
host thread
host memory
kernel<<<gridSize,blockSize>>>(...)
launches asynchronously
use to synchronize blocks
GPU
grid
global memory, atomics
runs independent
concurrency hazards
cannot synchronize
SM
block
shared memory, atomics
runs concurrent
concurrency hazards
__syncthreads()
warp
runs in parallel
SIMD
divergent branches
masking
SP
CUDA thread

Thank you

Questions?