并行与分布式计算

Ying Liu, Prof., Ph.D

School of Computer Science and Technology
University of Chinese Academy of Sciences
Data Mining and High Performance Computing Lab

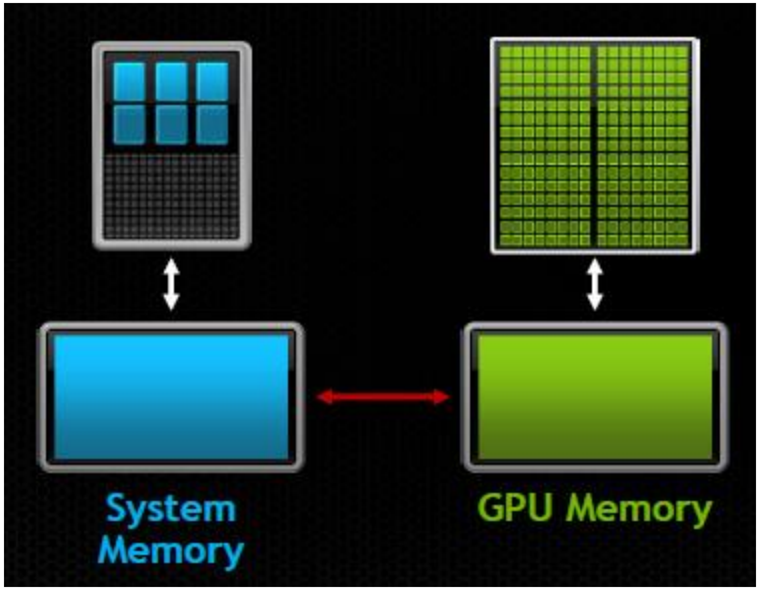
Outline

- Unified memory
- Dynamic parallelism
- Hyper-Q

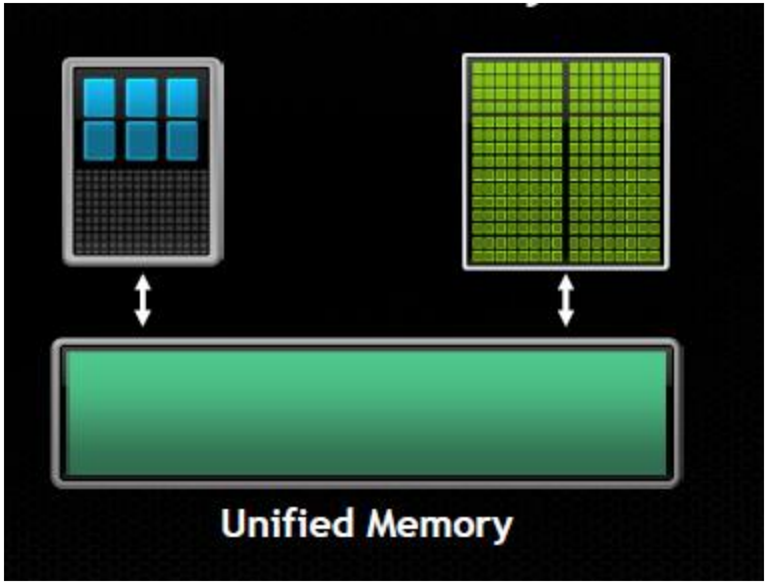
Unified Memory (Capability 2.0+, CUDA 6+)

- A single address space is used for the host and all the devices

Developer view without unified memory



Developer view with unified memory



Unified Memory

```
__global__ void AplusB( int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMalloc(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    int *host_ret = (int *)malloc(1000 * sizeof(int));
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int),
                    cudaMemcpyDeviceToHost);
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, host_ret[i]);
    free(host_ret);
    cudaFree(ret);
    return 0;
}
```

Program written without use of unified memory

Unified Memory

```
__global__ void AplusB(int *ret, int a, int b) {
     ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

Program written with use of unified memory

(use cudaMallocManaged() routine)

Unified Memory

```
__device__ __managed__ int ret[1000];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    AplusB<<< 1, 1000 >>>(10, 100);
    cudaDeviceSynchronize();
    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

Program written with use of unified memory

(use direct reference of a GPU-declared *__managed__* variable)

- *__managed__* qualifier
- A variable that can be directly referenced from host code

Unified Memory

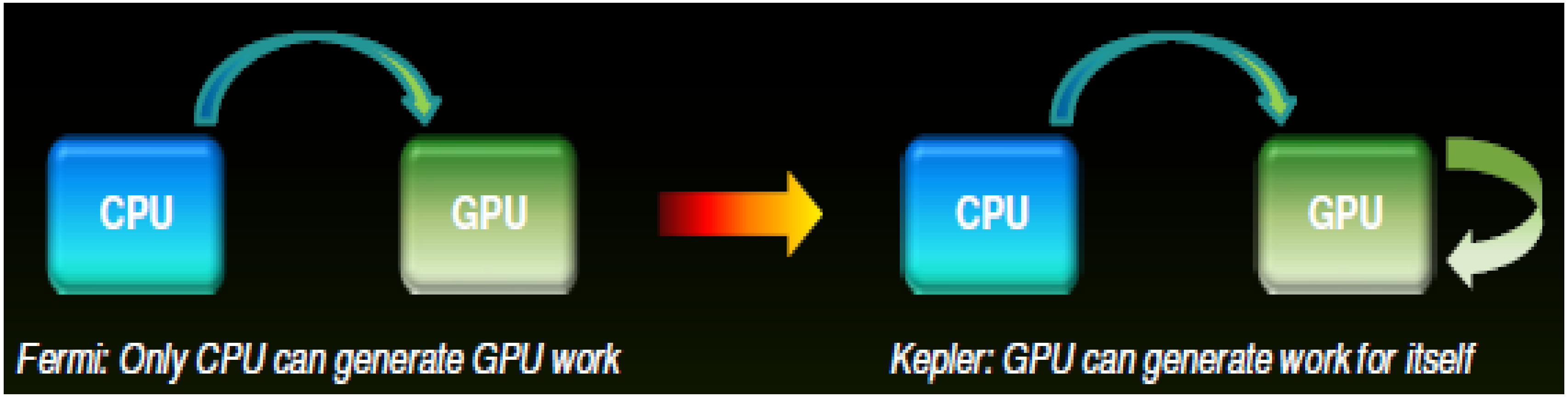
- Simpler programming & memory model
  - Single pointer to data, accessible anywhere
  - Tight language integration
  - Greatly simplifies code porting
- Performance through data locality
  - Migrate data to accessing processor
  - Guarantee global coherency
  - Overlap data transfer with kernel execution

Recent Release

- Unified memory
- Dynamic parallelism
- Hyper-Q

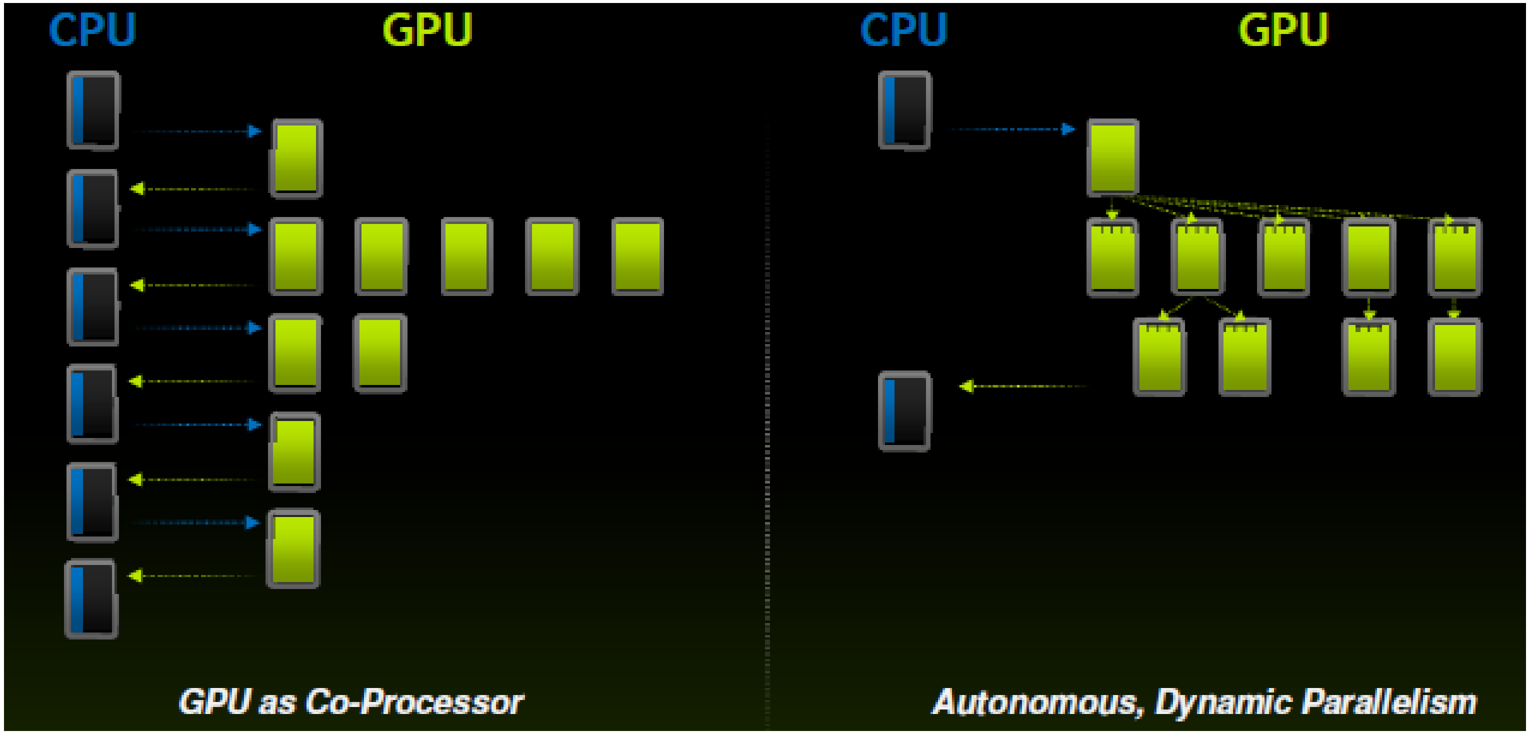
Dynamic Parallelism (Capability 3.5+)

- Launch new grids from the GPU
  - Dynamically
  - Simultaneously
  - Independently



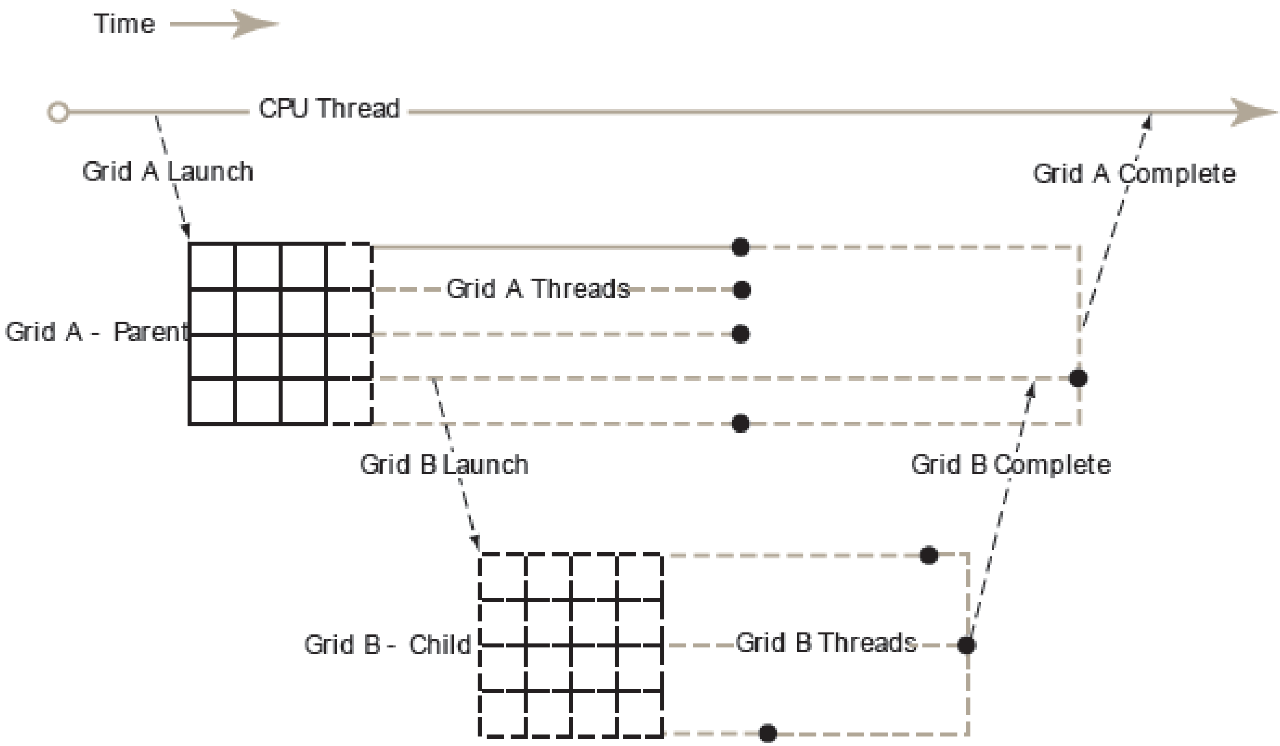
Dynamic Parallelism (Capability 3.5+)

- Reduce the need to transfer execution control and data between host and device



Dynamic Parallelism (Capability 3.5+)

- Parent-Child launch nesting
  - The runtime guarantees an implicit synchronization between the parent and the child



Dynamic Parallelism (Capability 3.5+)

- Parent and child grids share the same global and constant memory storage
- Parent and child grids have distinct local and shared memory
- All the device-side kernel launches are asynchronous with the launching thread
- Restrictions and limitations
  - Memory is reserved by the device runtime system for saving parent-grid state, tracking pending and launches
  - The max nesting depth is 24
  - The device runtime invokes *malloc*() and *free*() to allocate space for device-side launched kernels

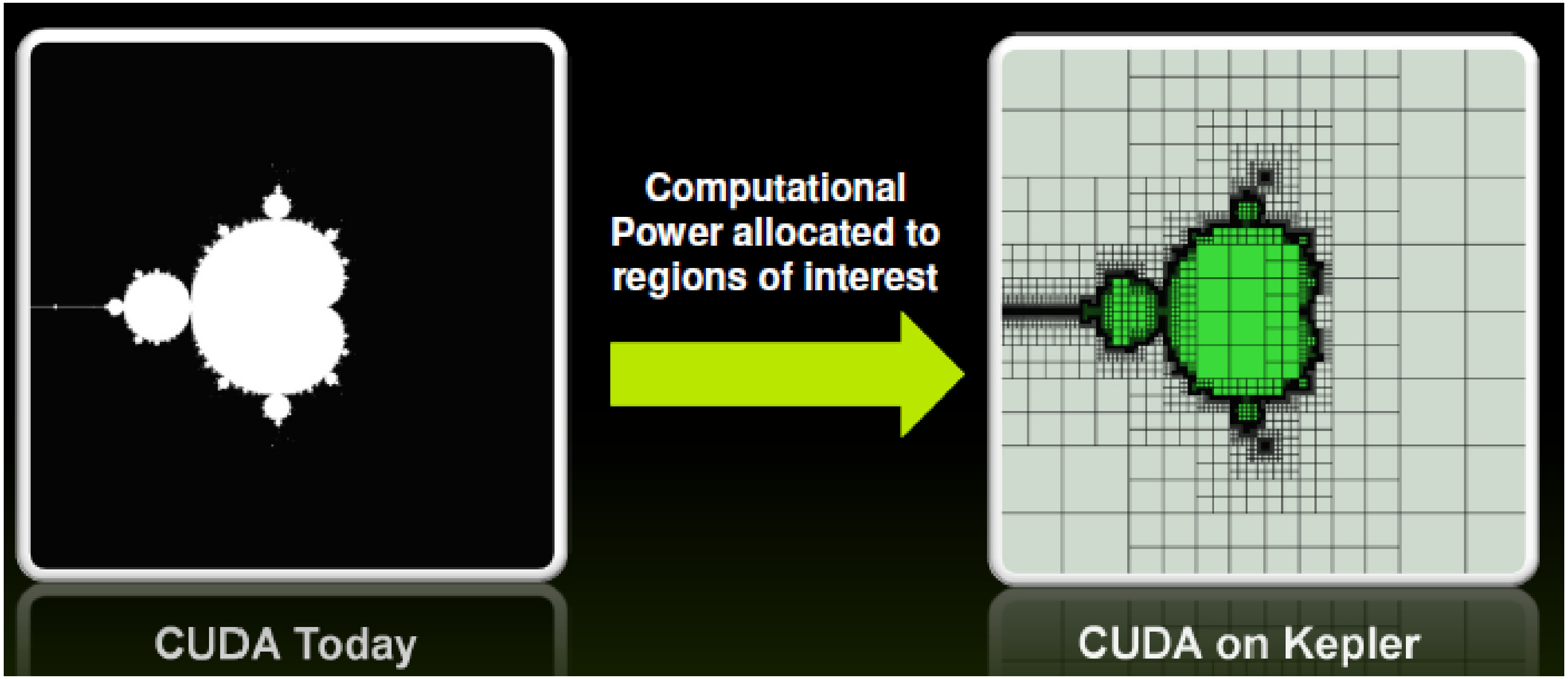
Program with Dynamic Parallelism

```
__global__ void child_launch (int *data) {
    data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch (int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        child_launch<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

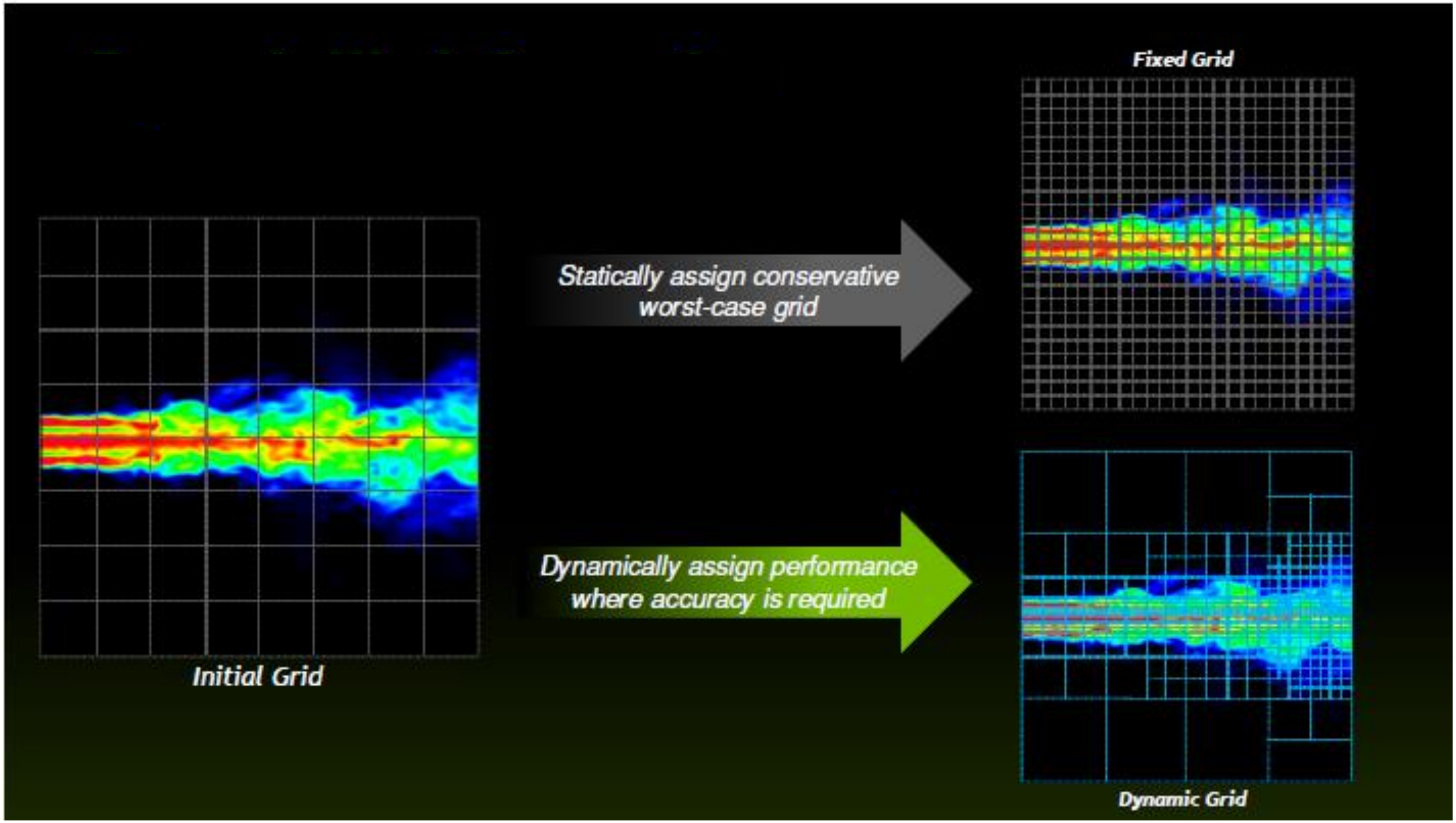
Program with Dynamic Parallelism

```
__global__ void permute (int n, int *data) {
    extern __shared__ int smem[];
    if (n <= 1)
        return;
    smem[threadIdx.x] = data[threadIdx.x];
    __syncthreads();
    permute_data(smem, n);
    __syncthreads();
    data[threadIdx.x] = smem[threadIdx.x];        // Write back to GMEM
                                                  since we can't pass
                                                  SMEM to children.
    __syncthreads();
    if (threadIdx.x == 0) {
        permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data);
        permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data+n/2);
    }
}
void host_launch (int *data) {
    permute<<< 1, 256, 256*sizeof(int) >>>(256, data);
}
```

Data-Dependent Parallelism



Dynamic Work Generation

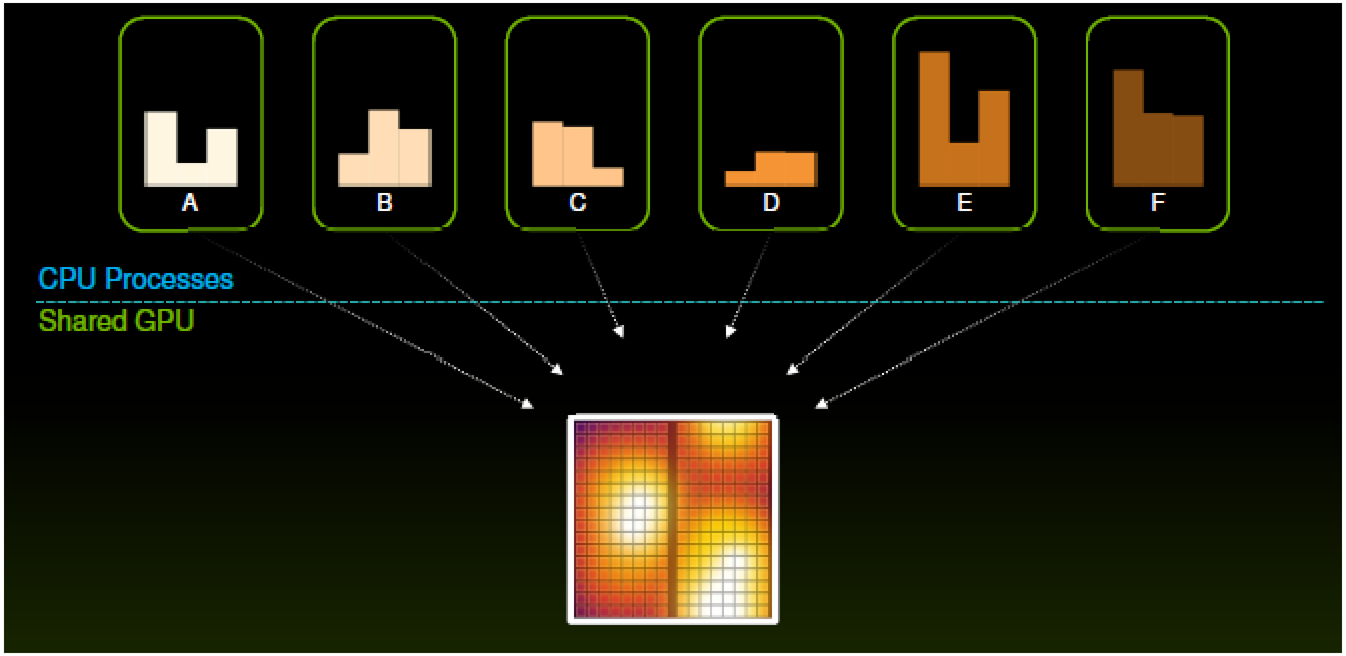


Recent Release

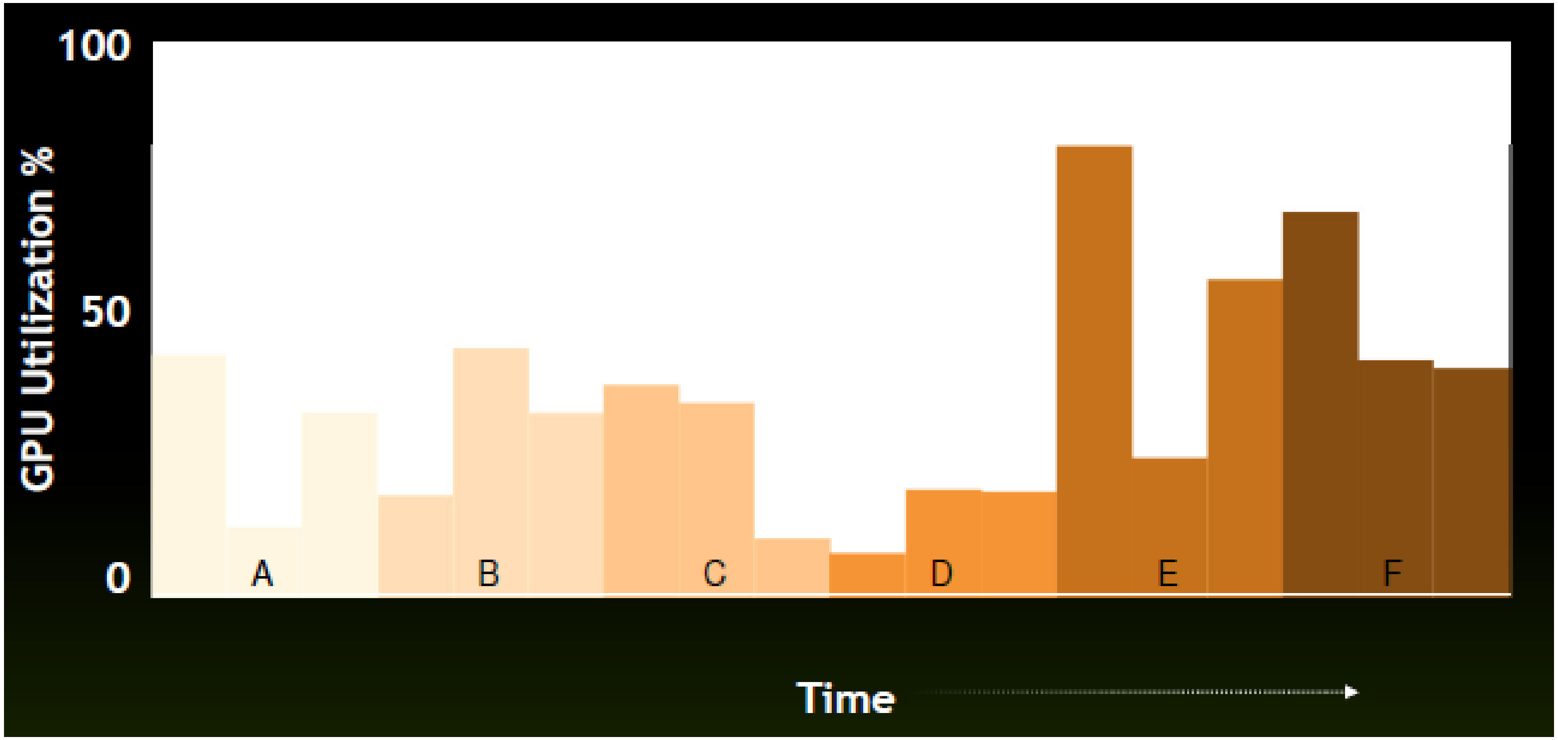
- Unified memory
- Dynamic parallelism
- Hyper-Q

Hyper-Q: Simultaneous Multiprocess

- Enable multiple CPU threads or processes to launch work on a single GPU simultaneously
  - Increase GPU utilization
  - Reduce CPU idle time



Without Hyper-Q



With Hyper-Q