OPTIMIZING KERNELS

OPENCL 2.0 UNIVERSITY TOOLKIT

Zhongliang Chen and Yash Ukidave,
Northeastern University Computer Architecture Research Lab
with
Perhaad Mistry and Dana Schaa, AMD

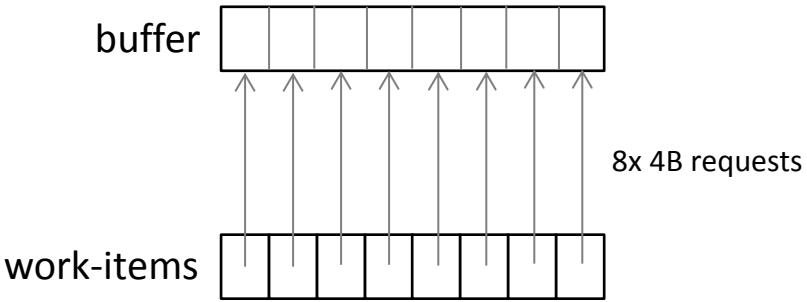
Instructor Notes

AMD

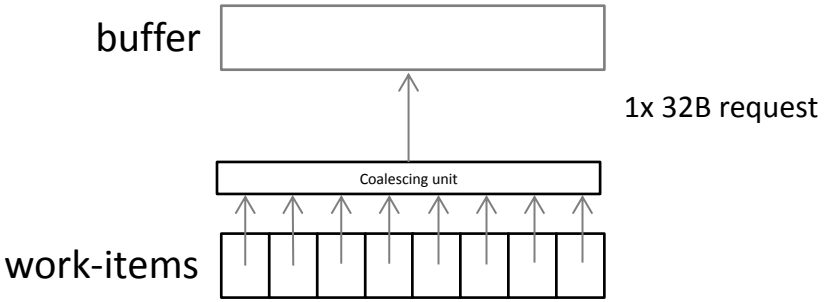
- These slides provide a high level overview of the source code optimization process
  - They cover some of the common optimization steps such coalescing, loop unrolling and vectorization
  - Students should choose some simple applications and try to apply these optimizations to their kernels.
  - They should also run their kernels under tools such as AMD CodeXL to understand the affects of their optimization on kernel performance
- A number of academic papers have covered GPU kernel optimization in detail and should be read alongside this material, some have been listed below
  - Optimization principles and application performance evaluation of a multithreaded GPU using CUDA - Shane Ryoo et.al
  - Exploiting memory access patterns to improve memory performance in data-parallel architectures - B Jang et.al
  - GPU Acceleration of Iterative Digital Breast Tomosynthesis - D Schaa et.al

Coalescing Memory Accesses

- Imagine a scenario where work-items are accessing elements of a buffer
- Naively, one memory request would be generated per work-item
  - With thousands of work-items executing per cycles, this would quickly congest the memory system
- GPU hardware supports *coalescing*, or combining multiple requests into fewer, larger requests



Non-coalesced



Coalesced

Coalescing Memory Accesses

AMD

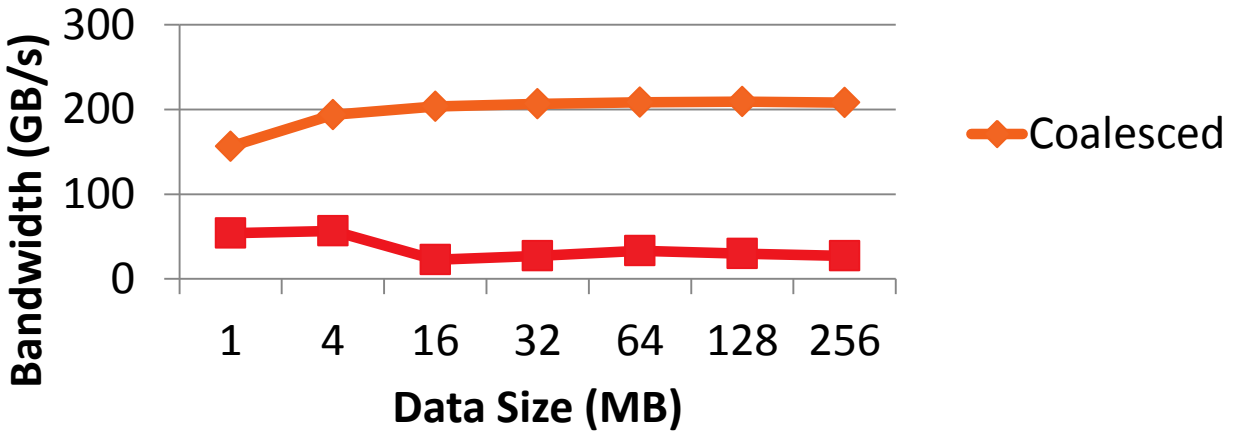
- Recall that for AMD hardware, 64 work-items form a wavefront and must execute the same instruction in a SIMD manner

- For the AMD R9 290X GPU, memory accesses of 16 consecutive work-items are evaluated together and can be coalesced to fully utilize the bus
  - This unit is called a quarter-wavefront and is the important hardware scheduling unit for memory accesses

Coalescing Memory Accesses

AMD

- Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an AMD R9 285 GPU



Vectorization

AMD

- Vectorization allows a single work-item to perform multiple operations at once
- Explicit vectorization is achieved by using vector datatypes (such as float4) in the source program
  - When a number is appended to a datatype, the datatype becomes an array of that length
  - Operations can be performed on vector datatypes just like regular datatypes
    - Each ALU will operate on different element of the float4 data
- CPUs and previous generations of AMD GPUs benefit from explicit vectorization
  - Current generations of AMD and NVIDIA GPUs execute “scalar” operations on SIMD lanes, which do not benefit from explicit vectorization

Vectorization

AMD

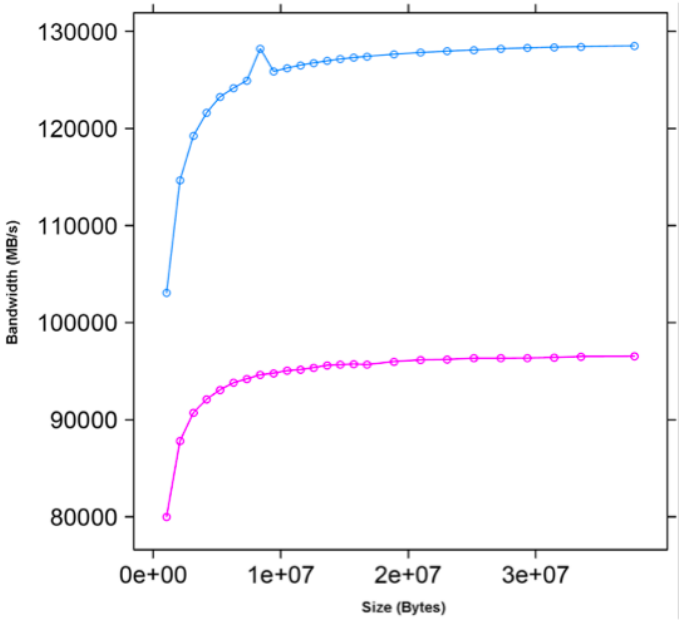
- On AMD Northern Islands and Evergreen GPUs, each processing element executes a multi-way VLIW instruction
  - Northern Islands: 4-way VLIW
    - 4 scalar operations or
    - 2 scalar operations + 1 transcendental operation
  - Evergreen: 5-way VLIW
    - 5 scalar operations or
    - 4 scalar operations + 1 transcendental operation

Vectorization

AMD

- Vectorization improves memory performance on AMD Northern Islands and Evergreen GPUs

```
__kernel void
Copy4(__global const float4 * input,
      __global float4 * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
__kernel void
Copy1(__global const float * input,
      __global float * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
```



Local Memory

AMD

- On GPUs, local memory maps to a high-bandwidth, low-latency memory located on chip
  - Useful for sharing data among work-items within a work-group
  - Accesses to local memory are usually much faster than accesses to global memory (even cached global memory)
  - Accesses to local memory usually do not require coalescing
  - More forgiving than global memory when having non-ideal access patterns
- Additional advantages on some AMD GPUs (e.g., Radeon HD 7970)
  - Local memory is mapped to LDS, 4x larger than L1 cache
  - LDS has a lower latency than L1 cache
- The tradeoff is that the use of local memory will limit the number of in-flight work-groups

Constant Memory

AMD

- Constant memory is a memory space to hold data that is accessed simultaneously by all work-items
  - Usually maps to specialized caching hardware that has a fixed size
  - It should NOT be used for general input data (e.g. an input buffer) that is read-only
- Examples of useful data to place in constant memory
  - Convolution filters, Kmeans cluster centriods, etc.
- Advantages for AMD hardware
  - If all work-items access the same address, then only one access request will be generated per wavefront
  - Constant memory can reduce pressure from L1 cache
  - Constant memory has lower latency than L1 cache

Occupancy

AMD

- Work-items from a work-group are launched together on a compute unit
  - In general, GPU hardware threads have a large amount of state
  - Only the very latest GPUs from AMD support context switching in the traditional sense, though with an extremely high penalty
  - In practice, work-group state is persistent on a compute unit, even during long latency operations
- If there are enough resources available, multiple work groups can be mapped to the same compute unit at the same time
  - Wavefronts from multiple work-group can be swapped in to hide latency
- Resources are fixed per compute unit (number of registers, local memory size, maximum number of wavefronts)
  - Any one of these resource constraints may limit the number of work-groups on a compute unit
- The term *occupancy* is used to describe how well the resources of the compute unit are being utilized

Occupancy: Registers

AMD

- The availability of registers is one of the major limiting factor for large kernels
- On current GPUs, the maximum number of registers required by a kernel must be available for all work-items of a workgroup
  - Example: Consider a GPU with 16384 registers per compute unit running a kernel that requires 35 registers per work-item
  - Each compute unit can execute at most 468 work-items
    - This affects the choice of workgroup size
      - A work-group of 512 is not possible
      - Only 1 work-group of 256 work-items is allowed at a time, even though 212 more work-items could be running
      - 3 work-groups of 128 work-items are allowed, providing 384 work-items to be scheduled, etc.

Occupancy: Registers

AMD

- Consider another example:
  - A GPU has 16384 registers per compute unit
  - The work-group size of a kernel is fixed at 256 work-items
  - The kernel currently requires 17 registers per work-item
- Given the information, each work group requires 4352 registers
  - This allows for 3 active work-groups if registers are the only limiting factor
- If the code can be restructured to only use 16 registers, then 4 active work groups would be possible

Occupancy: Local Memory

AMD

- GPUs have a limited amount of local memory on each compute unit
  - 64 KB local memory on AMD GPUs
- Local memory limits the number of active work-groups per compute unit
- Depending on the kernel, the data per work-group may be fixed regardless of number of work-items (e.g., histograms), or may vary based on the number of work-items (e.g., matrix multiplication, convolution)

Occupancy: Work-items/work-groups

AMD

- GPUs have hardware limitations on the maximum number of work-items per work-group
  - OpenCL limits work-groups to 256 work-items
- AMD GPUs have per-SIMD limits on the number of wavefronts
  - 40 wavefronts (2560 work-items) per compute-unit
  - For a 44 Compute Unit GPU such as the R9 290X there can be upto 40x44 = 1760 wavefronts active on the device

Occupancy: Limiting Factors

AMD

- The minimum of these three factors is what limits the active number of work-items (or occupancy) of a compute unit
- The interactions between the factors are complex
  - The limiting factor may have either work-item or wavefront granularity
  - Changing work-group size may affect register or local memory usage
  - Reducing any factor (such as register usage) slightly may have allow another work group to be active
- AMD CodeXL plots these factors visually allowing the tradeoffs to be visualized

Thread Mapping

AMD

- Thread mapping determines which threads will access which data
  - Proper mappings can align with hardware and provide large performance benefits
  - Improper mappings can be disastrous to performance
- The paper *Static Memory Access Pattern Analysis on a Massively Parallel GPU* by Jang, et. al focuses on the task of effectively mapping threads to the data access patterns of an algorithm

Thread Mapping

- By using different mappings, the same thread can be assigned to access different data elements
  - The examples below show three different possible mappings of threads to data (assuming the thread id is used to access an element)

Mapping				

	`int tid = get_global_id(1) * get_global_size(0) + get_global_id(0);`	`int tid = get_global_id(0) * get_global_size(1) + get_global_id(1);`	`int group_size = get_local_size(0) * get_local_size(1);`
			
			`int tid = get_group_id(1) * get_num_groups(0) * group_size + get_group_id(0) * group_size + get_local_id(1) * get_local_size(0) + get_local_id(0)`

Thread IDs	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15	0 1 4 5 2 3 6 7 8 9 12 13 10 11 14 15

*assuming 2x2 groups

Thread Mapping

AMD

- Consider a serial matrix multiplication algorithm

```
for (i1=0; i1 < M; i1++)
    for (i2=0; i2< N; i2++)
        for (i3=0; i3 < P; i3++)
            C[i1][i2] += A[i1][i3]*B[i3][i2];
```

- This algorithm is suited for output data decomposition
  - We will create *NM* threads
    - Effectively removing the outer two loops
  - Each thread will perform *P* calculations
    - The inner loop will remain as part of the kernel
- Should the index space be MxN or NxM?

Thread Mapping

AMD

- Thread mapping 1: with an MxN index space, the kernel would be:

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

- Thread mapping 2: with an NxM index space, the kernel would be:

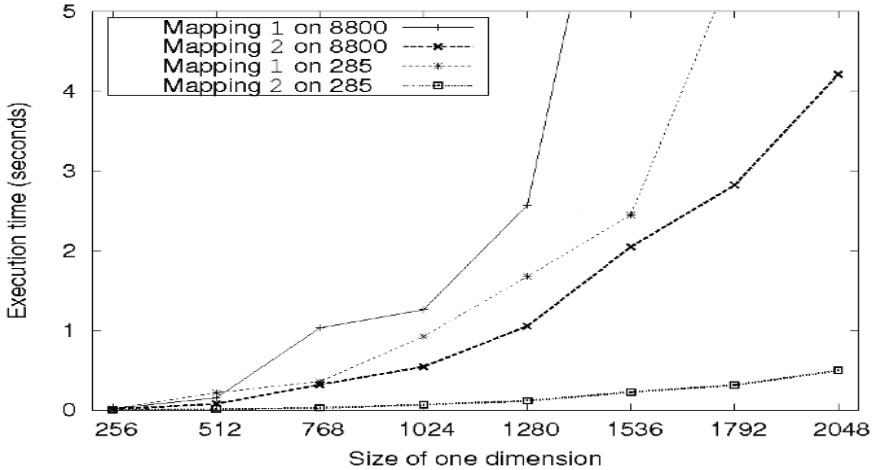
```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

- Both mappings produce functionally equivalent versions of the program

Thread Mapping

AMD

- This figure shows the execution of the two thread mappings on NVIDIA GeForce 285 and 8800 GPUs



- Notice that mapping 2 is far superior in performance for both GPUs

Thread Mapping

AMD

- The discrepancy in execution times between the mappings is due to data accesses on the global memory bus
  - Assuming row-major data, data in a row (i.e., elements in adjacent columns) are stored sequentially in memory
  - To ensure coalesced accesses, consecutive threads in the same wavefront should be mapped to columns (the second dimension) of the matrices
    - This will give coalesced accesses in Matrices B and C
    - For Matrix A, the iterator *i3* determines the access pattern for row-major data, so thread mapping does not affect it

Thread Mapping

AMD

- In mapping 1, consecutive threads (*tx*) are mapped to different rows of Matrix C, and non-consecutive threads (*ty*) are mapped to columns of Matrix B
  - The mapping causes inefficient memory accesses

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

Thread Mapping

AMD

- In mapping 2, consecutive threads (*tx*) are mapped to consecutive elements in Matrices B and C
  - Accesses to both of these matrices will be coalesced
    - Degree of coalescence depends on the workgroup and data sizes

```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

Thread Mapping

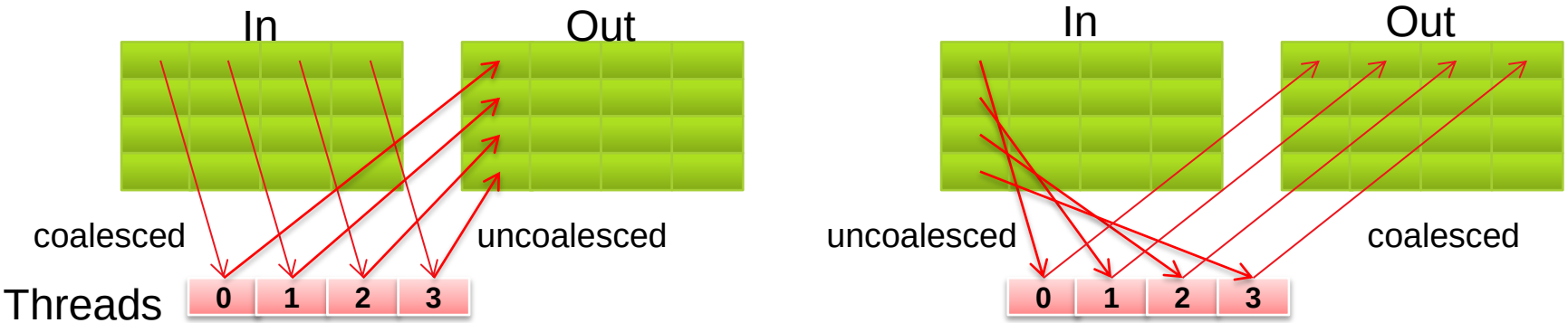
AMD

- In general, threads can be created and mapped to any data element by manipulating the values returned by the thread identifier functions
- The following matrix transpose example will show how thread IDs can be modified to achieve efficient memory accesses

Matrix Transpose

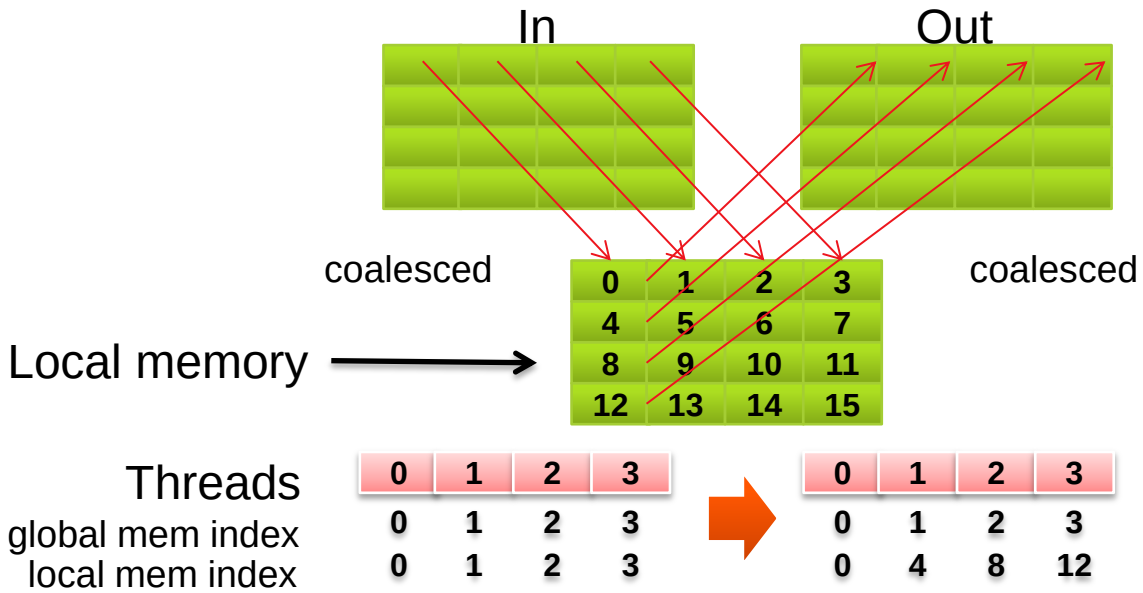
AMD

- A matrix transpose is a straightforward technique
  - Out(x,y) = In(y,x)
- No matter which thread mapping is chosen, one operation (read/write) will produce coalesced accesses while the other (write/read) produces uncoalesced accesses
  - Note that data must be read to a temporary location (such as a register) before being written to a new location



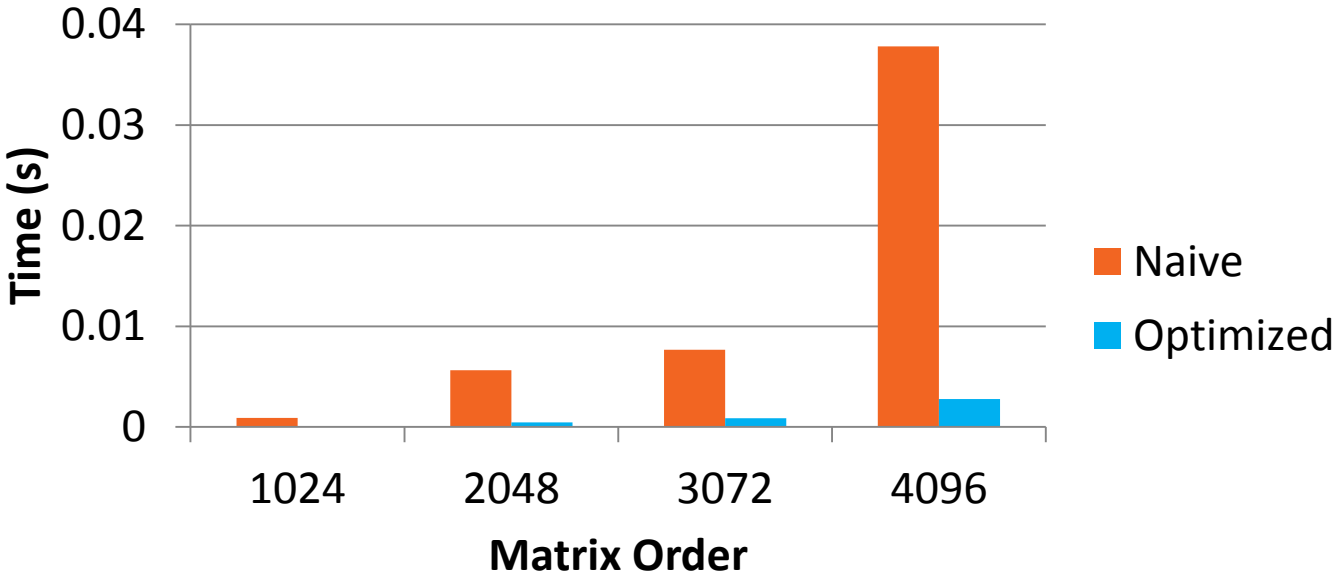
Matrix Transpose

- If local memory is used to buffer the data between reading and writing, we can rearrange the thread mapping to provide coalesced accesses in both directions
  - Note that the work group must be square



Matrix Transpose

- The following figure shows a performance comparison of the two transpose kernels for matrices of size NxM on an AMD 5870 GPU
  - "Optimized" uses local memory and thread remapping



Summary

AMD

- Although writing a simple OpenCL program is relatively easy, optimizing code can be more difficult
  - Coalescing memory access
  - Vectorization
  - Local memory
  - Constant memory
- When creating work groups, hardware limitations (number of registers, size of local memory, etc.) need to be considered
  - Work-groups must be sized appropriately to maximize the number of active work-items and properly hide latencies
- Thread mapping, and its effect on accessing memory, is critical for OpenCL kernel performance