上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

CS4302-01
Parallel and Distributed Computing

Lecture 6 GPU Architecture

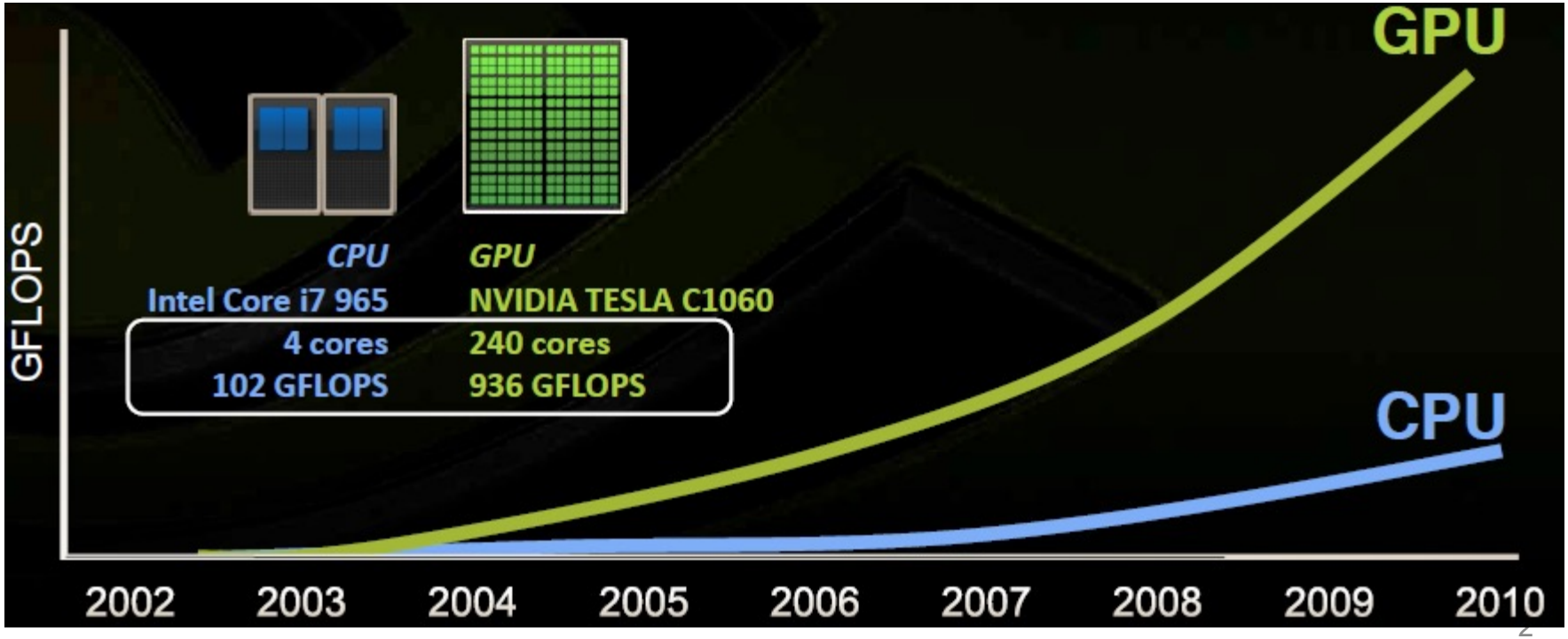
Zhuoran Song

2023/10/10

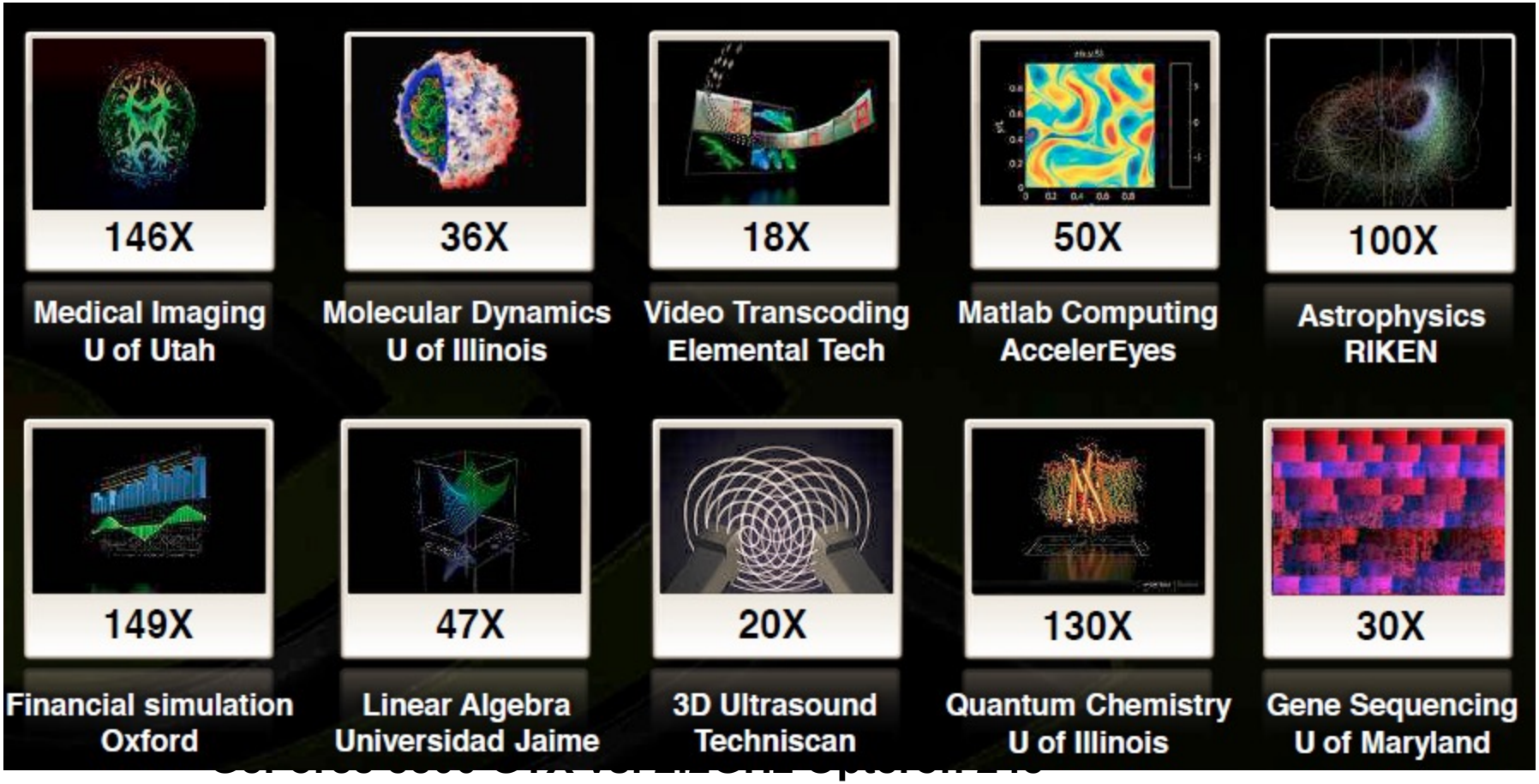
GPU Scaling

A quiet revolution and potential build-up

- Calculation: 20 TFLOPS vs. 100 GFLOPS
- Memory Bandwidth: 1.6TB/s vs. 16 GB/s
- Every PC, phone, pad has GPU now

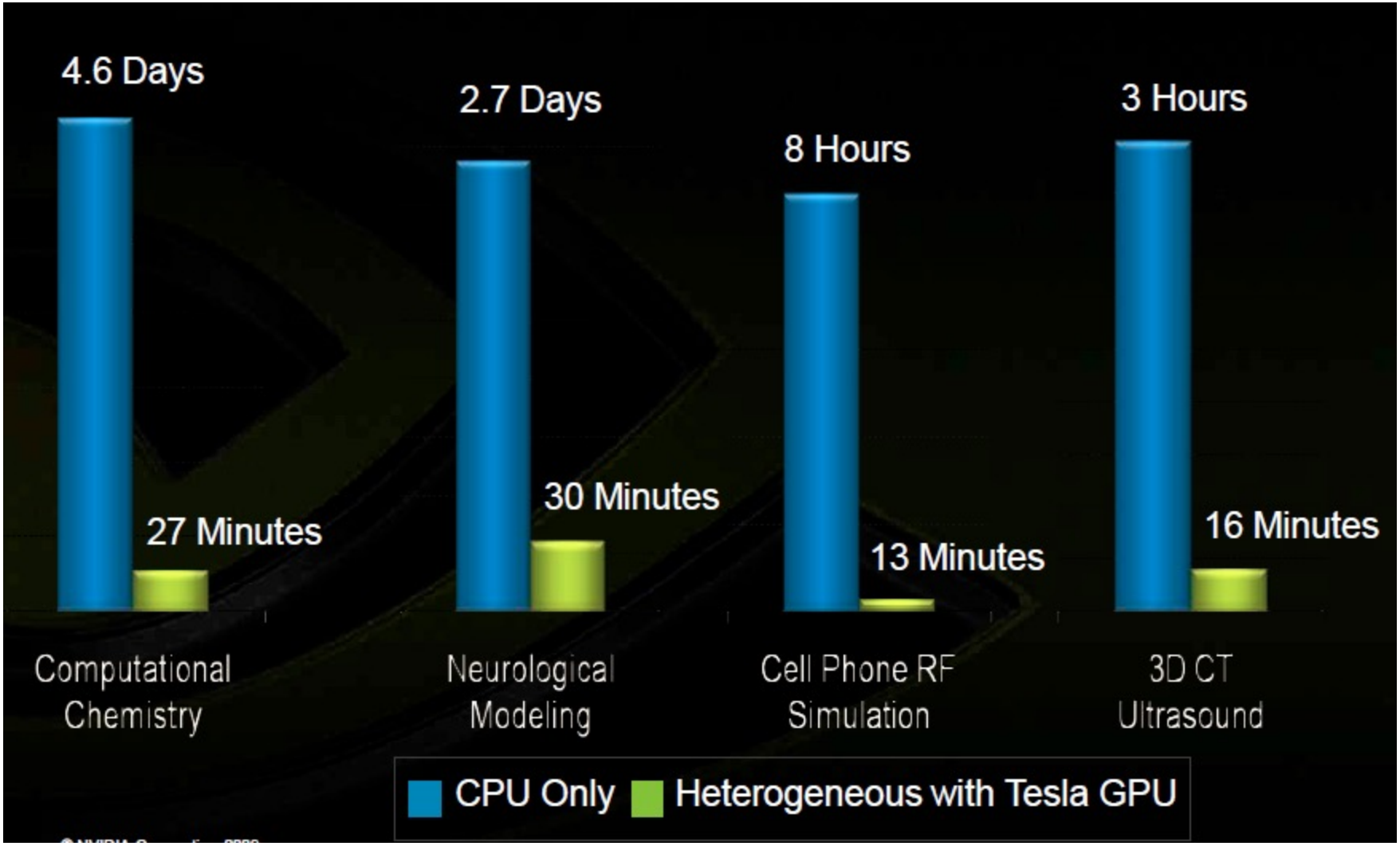


GPU Speedup

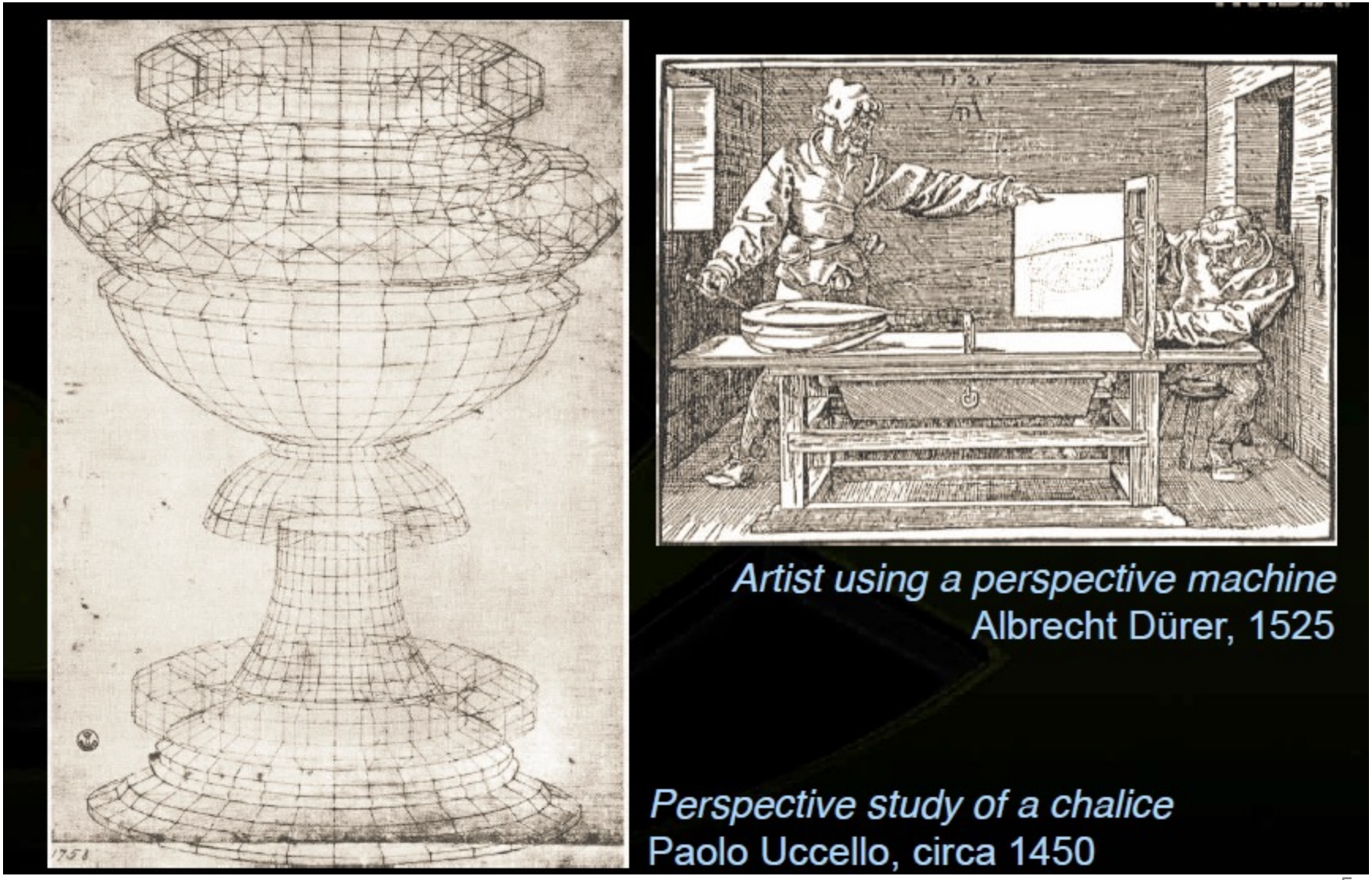


- 10´ speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25´ to 400´ speedup if the function's data requirements and control flow suit the GPU and the application is optimized

GPU Speedup


4.6 Days
27 Minutes
2.7 Days
30 Minutes
8 Hours
13 Minutes
3 Hours
16 Minutes
Computational Chemistry
Neurological Modeling
Cell Phone RF Simulation
3D CT Ultrasound
CPU Only
Heterogeneous with Tesla GPU

Early Graphic Hardware



Artist using a perspective machine
Albrecht Dürer, 1525

Perspective study of a chalice
Paolo Uccello, circa 1450

Early Electronic Machine

Early Graphic Chip

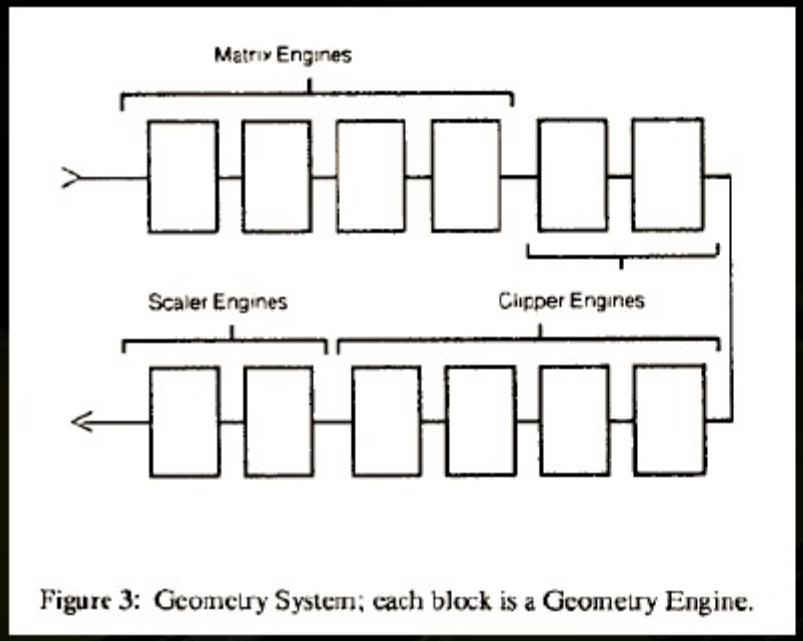
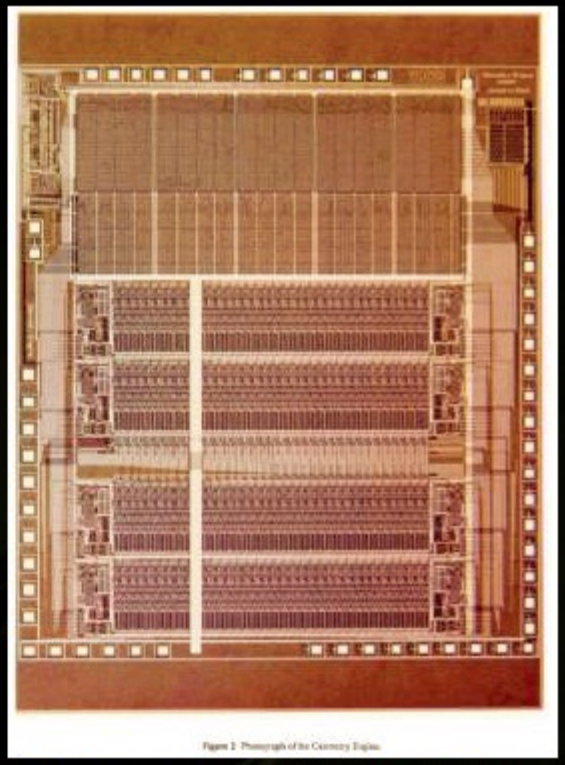


Figure 3: Geometry System; each block is a Geometry Engine.



The Geometry Engine: A VLSI Geometry System for Graphics
Jim Clark, 1982

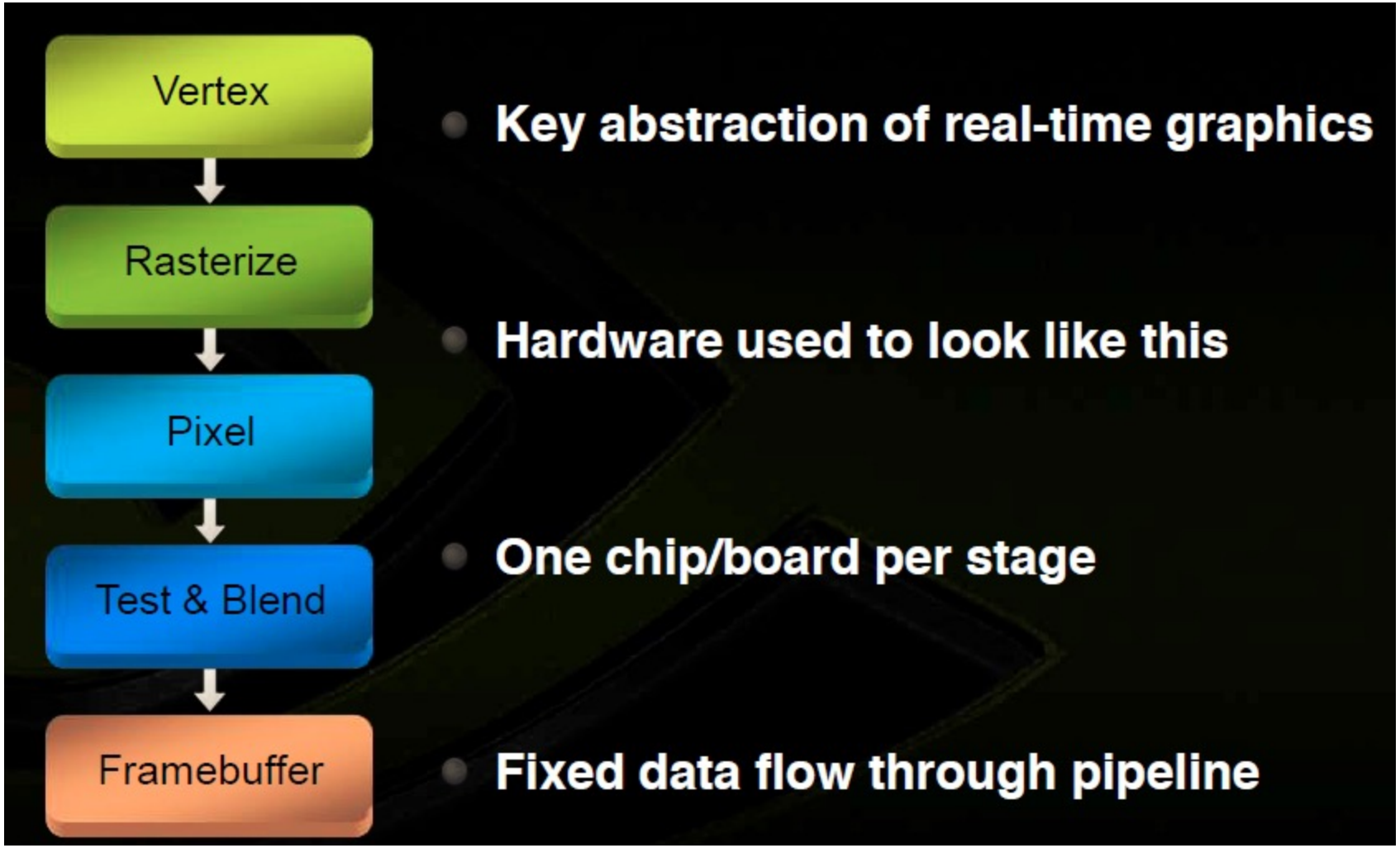
Graphic Pipeline

- Sequence of operations to generate an image using object-order processing
  - *Primitives processed one-at-a-time*
  - *Software pipeline: e.g. Renderman*
- High-quality and efficiency for large scenes
  - *Hardware pipeline: e.g. graphics accelerators*
- Will cover algorithms of modern hardware pipeline
  - *But evolve drastically every few years*
  - *We will only look at triangles*

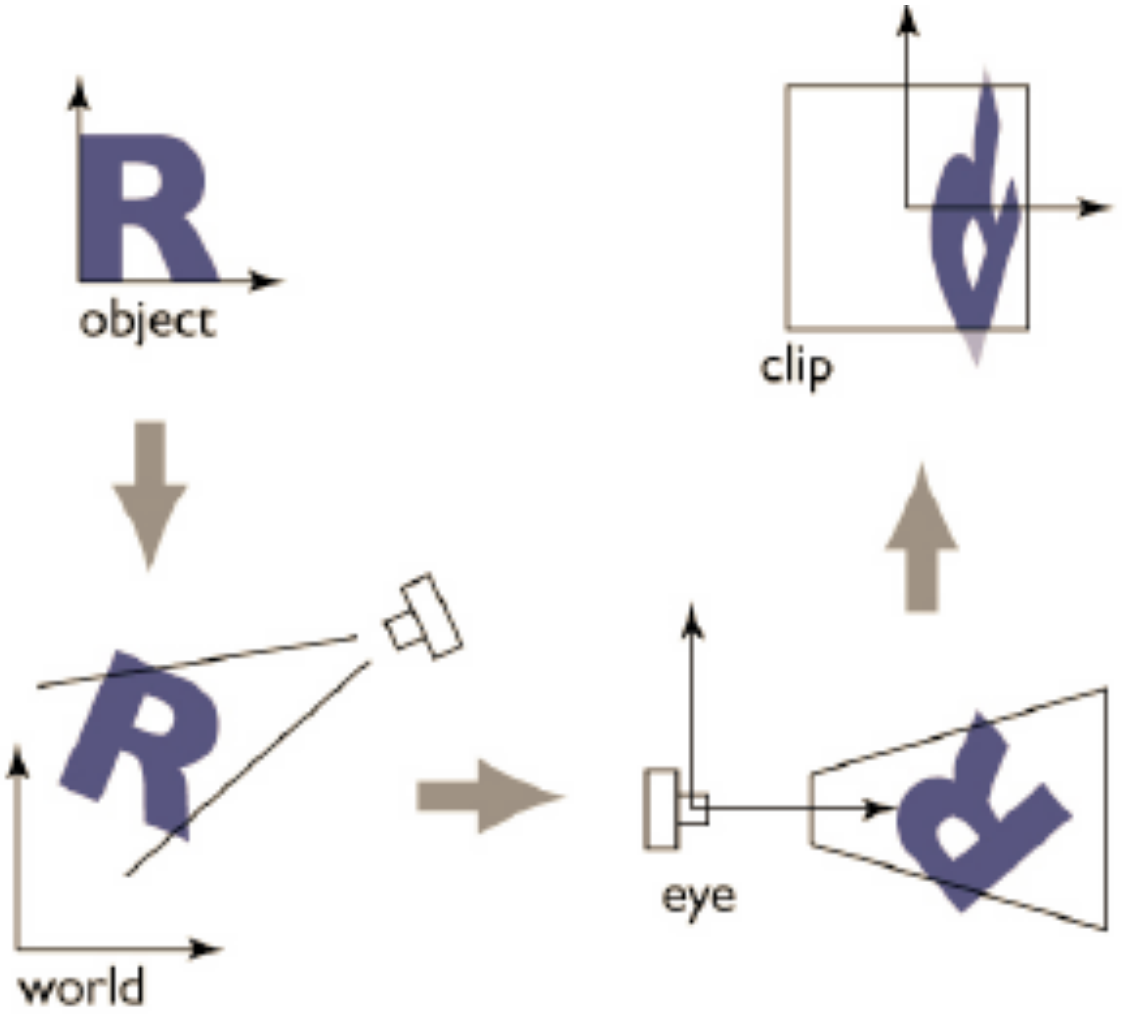
Graphic Pipeline

- Handles only simple primitives by design
  - *Points, lines, triangles, quads (as two triangles)*
  - *Efficient algorithm*
- Complex primitives by tessellation
  - *Complex curves: tessellate into line strips*
  - *Curves surfaces: tessellate into triangle meshes*
- "pipeline" name derives from architecture design
  - *Sequences of stages with defined input/output*
  - *Easy-to-optimize, modular design*

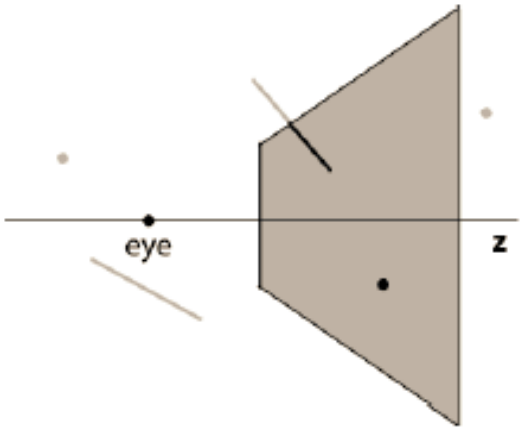
Graphic Pipeline

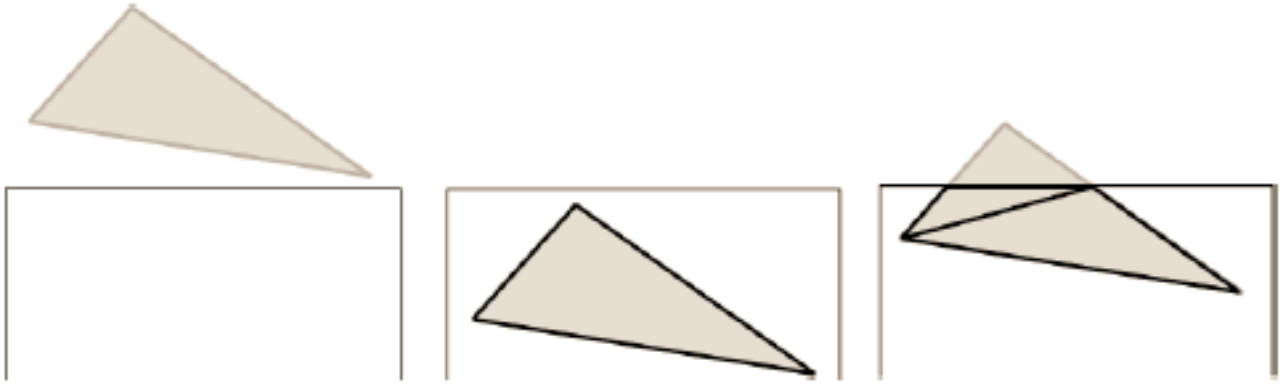


Vertex Processing

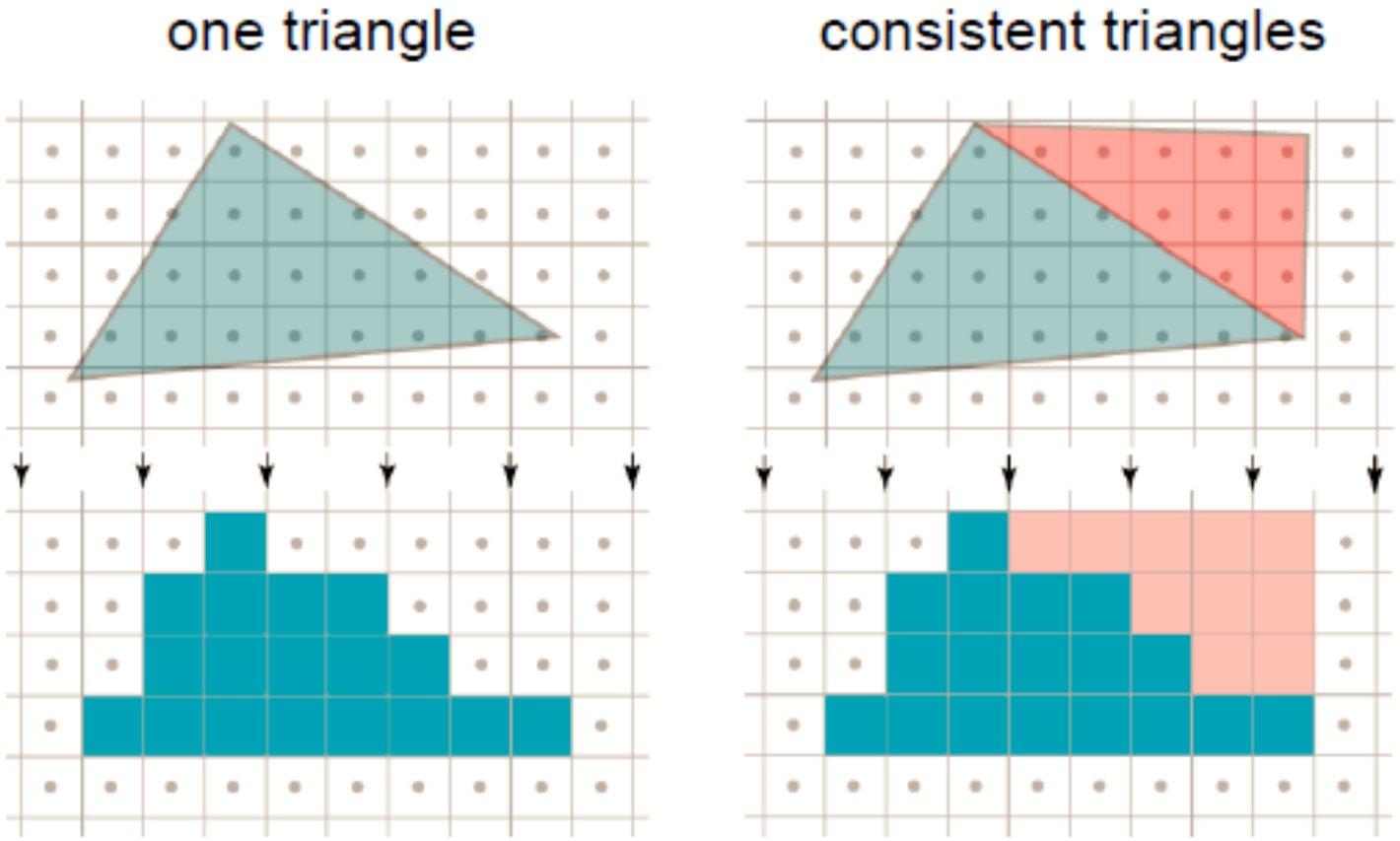


Clipping

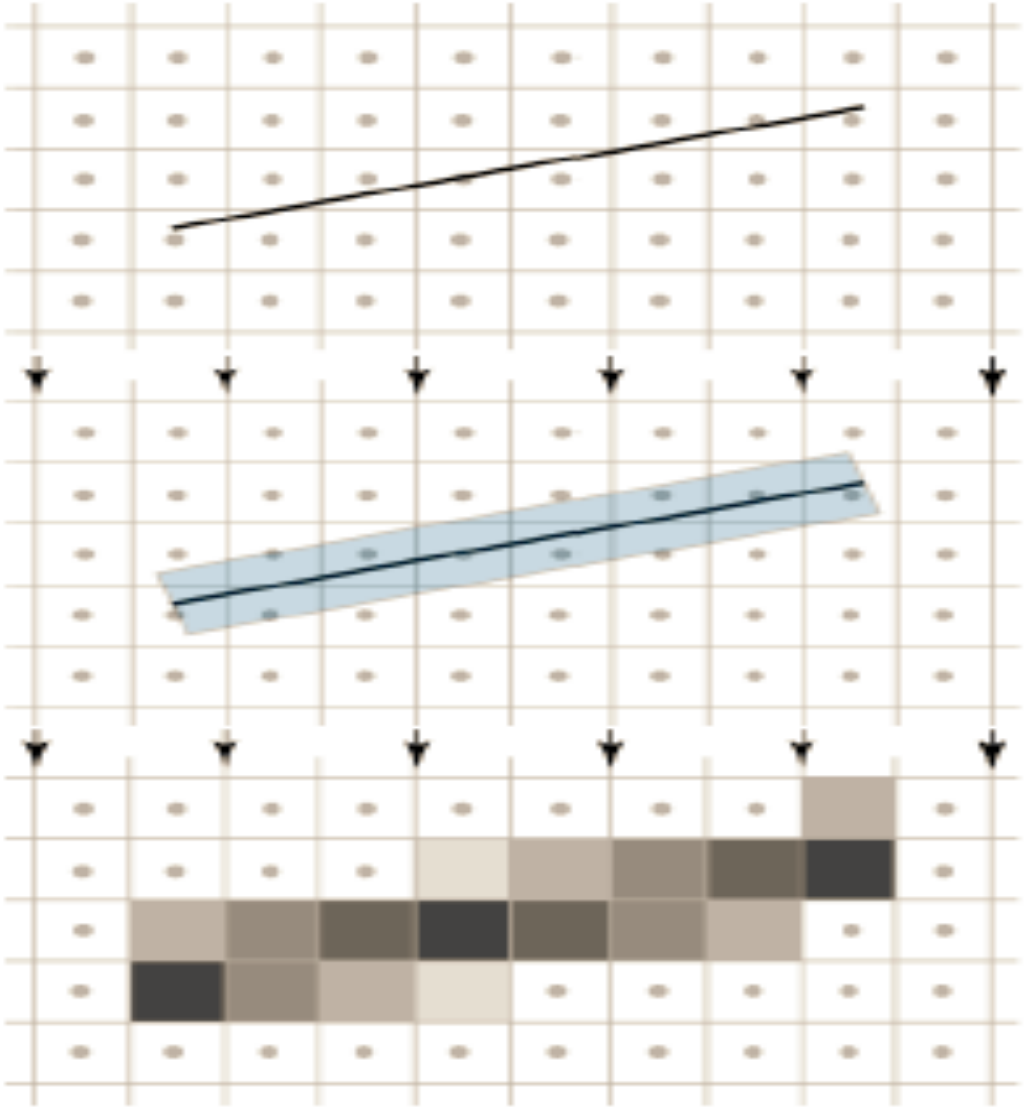




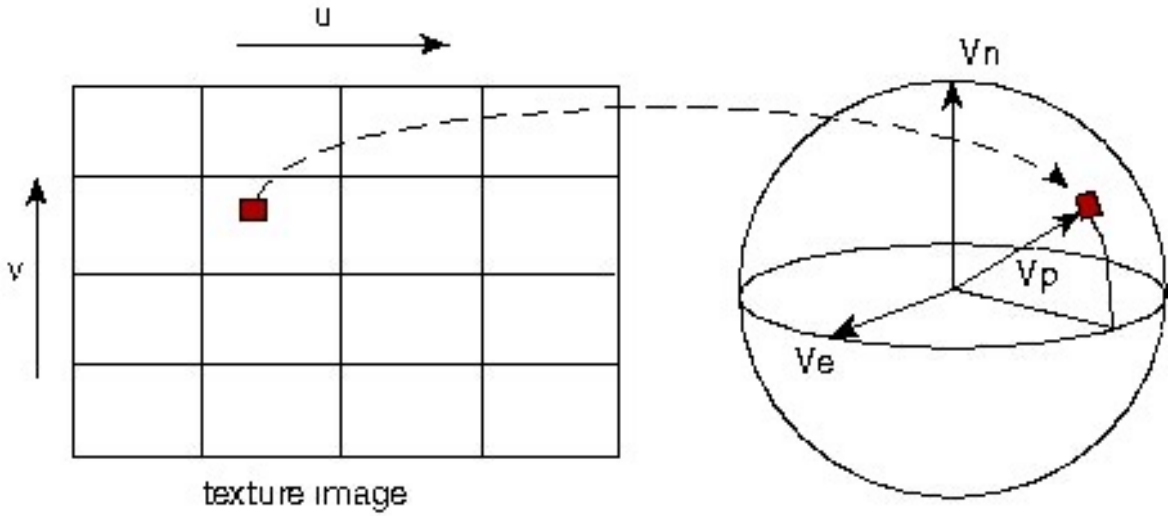
Rasterization



Anti-Aliasing

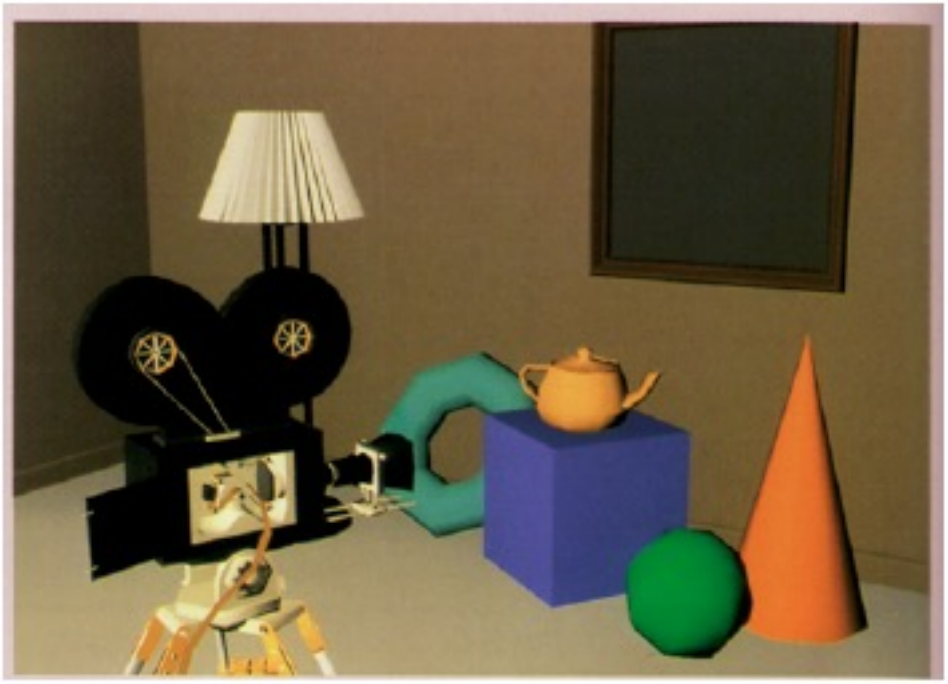


Texture



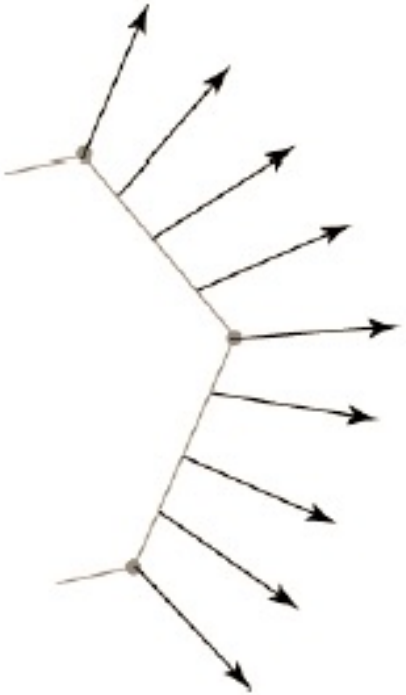
Gouraud Shading

- compute light at vertex position
  - with vertex normals
- interpolate colors linearly over the triangle

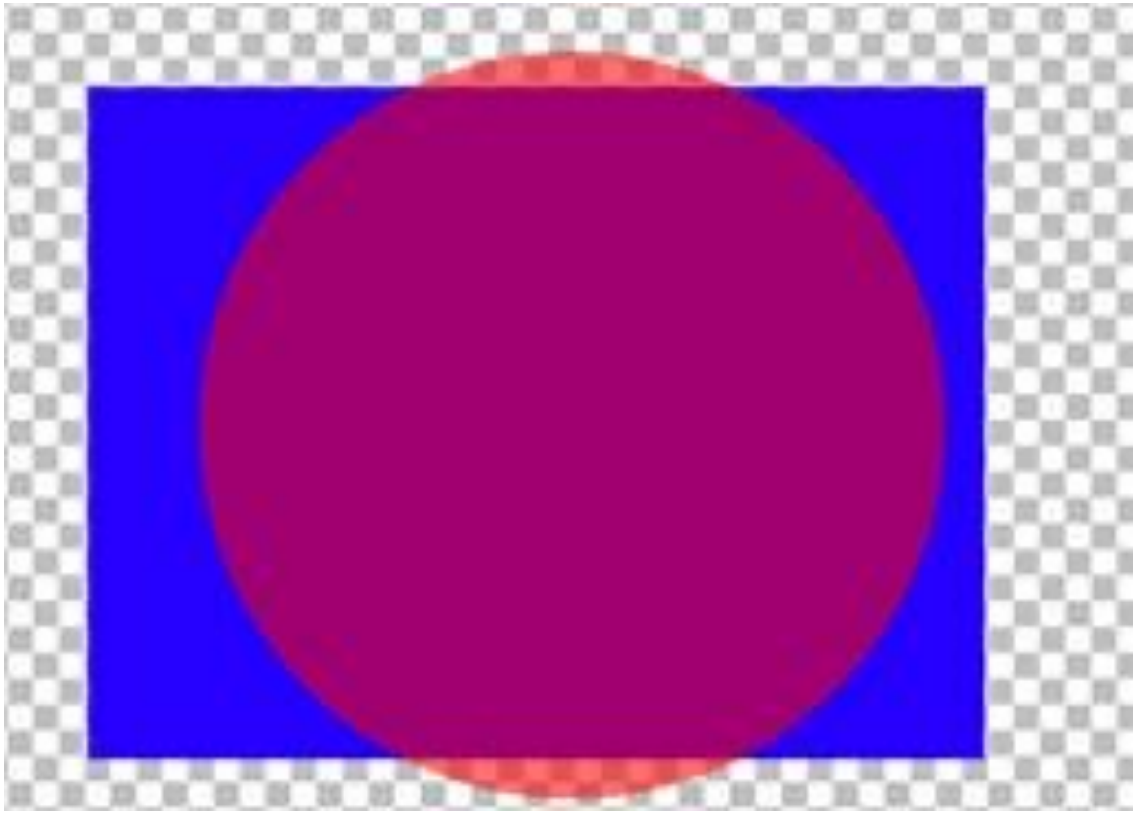


Phong Shading

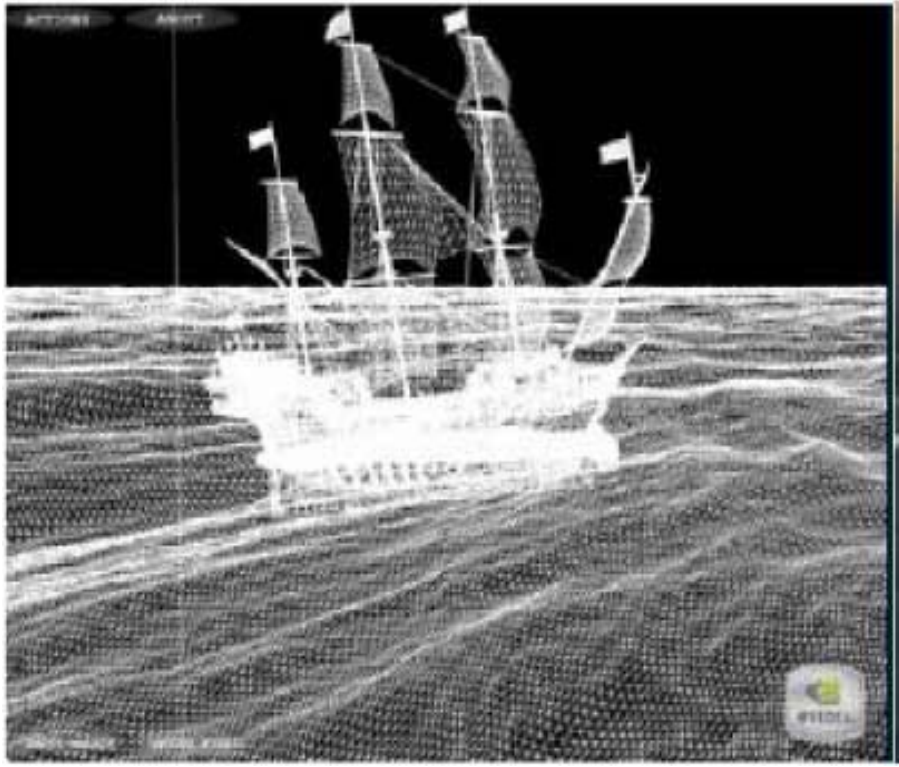
- interpolate normals per-pixels: shading normals
- compute lighting for each pixel
  - lighting depends less on tessellation



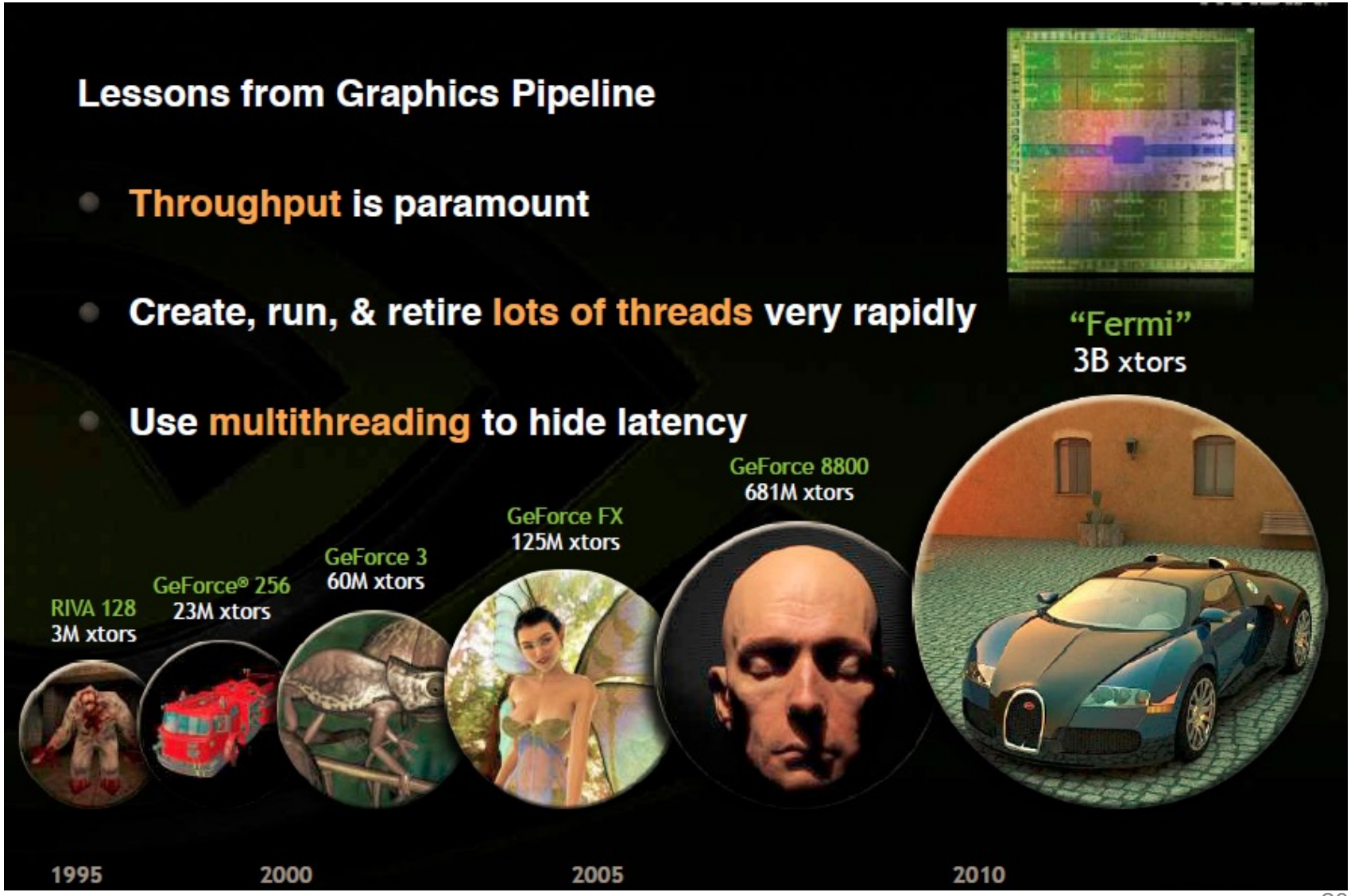
Alpha Blending



Wireframe



GPU Evolution



Moore's Law

- Computers no longer get faster, just wider

- You *must re-think your algorithms to be parallel !*

- Data-parallel computing is most scalable solution

Homework

Problem 1

% 考察对for循环中的数据依赖的理解

Floyd's algorithm is an $\theta(n^3)$ algorithm that solves the all-pairs shortest-path problem. Given an $n \times n$ ajacency matrix A, the following algorithm outputs the shortest path between every pair of vertices. Please use OpenMP to implement a parallel version of Floyd's algorithm. Benchmark your program for different values of n and t(number of threads).Please analyze the speedup and give your reason.

```
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
      }
    }
}
```

Homework

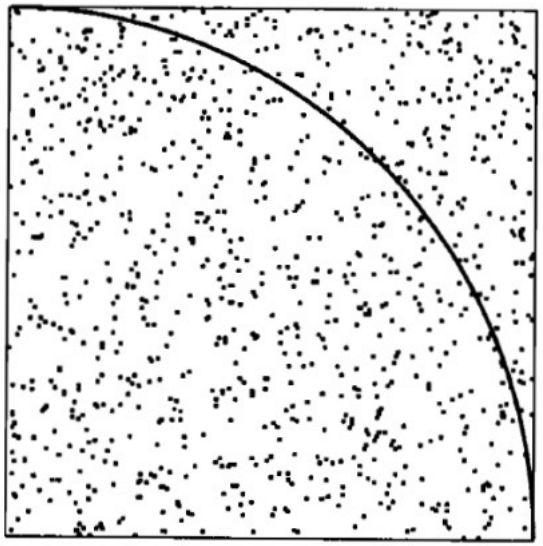
```
void calculate_all_pair_shortest_path(int **nodes_distance)
{
    int possible_short_dist; /* to hold distance between i and j via k */

    for (int k = 0; k < n; ++k)
    {
#pragma omp for schedule(dynamic)
        for (int i = 0; i < n; ++i)
        {
            for (int j = 0; j < n; ++j)
            {
                /* if both nodes (i & j) are not same and path between node_i
                 * and Node_j via node_k exists (i.e. anything except 0)
                 *
                 */
                if ((A[i][k] * A[k][j] != 0) && (i != j))
                {
                    A = A[i][k] + A[k][j];
                    if ((A[i][j] <= possible_short_dist) &&
                        (A[i][j] != 0))
                        continue;

                    A[i][j] = possible_short_dist;
                }
            }
        }
    }
}
```

Homework

We can use Monte Carlo method to compute π. The idea is to generate pairs of points in the unit square (where each coordinate varies between 0 and 1) .We count the fraction of points inside the circle(those points for which $x^2 + y^2 \leq 1$) .The expected value of this fraction is $\pi/4$;so multiplying the fraction by 4 gives an estimate of π.



```
int count ;  //points inside the circle
int i;
int samples;  //total points
unsigned short xi[3];
int t ; // number of threads
int tid;
double x,y;

samples = atoi(argv[1]);
xi[0] = atoi(argv[2]);
xi[1] = atoi(argv[3]);
xi[2] = atoi(argv[4]);
count = 0;

for( i = tid; i < samples ;i+= t) {
    x = erand48(xi);
    y = erand48(xi);
    if(x*x + y*y <= 1.0 )
        count ++;
}


printf("Estimate of pi : %7.5f \n",4.0 * count /samples);
```

Homework

```
#pragma omp parallel private(xi, t, i, x, y, local_count)
  {
    local_count = 0;
    xi[0] = atoi(argv[3]);
    xi[1] = atoi(argv[4]);
    xi[2] = tid = omp_get_thread_num();
    t = omp_get_num_threads();
    for (i = tid; i < samples; i += t) {
      x = erand48(xi);
      y = erand48(xi);
      if (x * x + y * y <= 1.0)
        local_count++;
      printf("%d\n", i);
    }

#pragma omp critical
    count += local_count;
  }
```

Homework

% lock + barrier + reduction

Please simulate a bank system which handles concurrent cash deposits and cash withdrawals of $n = 5$ accounts. Each thread either deposit or withdraw random amount of money from a random account. If the account balance is not enough, the thread withdraw no money. Make sure that at any moment, there is only one thread modifying the same account's balance to ensure data consistency. When all transactions finished, each thread computes the balance sum of all accounts. The balance sum computed by each thread should be eqal.

Hint: Please use the openmp reduction operation to calculate the balance sum.

Hon

```c
#define ACCOUNT_NUM 5

omp_lock_t locks[ACCOUNT_NUM];

int account_balance[ACCOUNT_NUM];

int deposit(int account, int amount) {
  omp_set_lock(&locks[account]);

  account_balance[account] += amount;

  omp_unset_lock(&locks[account]);

  return 1;
}

int withdraw(int account, int amount) {
  omp_set_lock(&locks[account]);

  if (account_balance[account] < amount) {
    omp_unset_lock(&locks[account]);
    return 0;
  } else {
    account_balance[account] -= amount;
    omp_unset_lock(&locks[account]);
  }
```

H

```
#pragma omp parallel num_threads(num_threads)
  {
    int thread_id = omp_get_thread_num();
    srand(thread_id);
    int amount = rand() % 100;

    int dep = rand() % 2;

    int account = rand() % ACCOUNT_NUM;
    int success;

    if (dep) {
      deposit(account, amount);
    } else {
      withdraw(account, amount);
    }

#pragma omp barrier

    int total_sum = 0;

#pragma omp parallel for reduction(+ : total_sum)

    for (int i = 0; i < ACCOUNT_NUM; i++) {
      total_sum += account_balance[i];
    }

    printf("total balance sum of all accounts: %d\n", total_sum);
  }
```