

# AMATH 483/583

## High Performance Scientific Computing

### Lecture 5: Classes, vectors, matrices

Andrew Lumsdaine  
Northwest Institute for Advanced Computing  
Pacific Northwest National Laboratory  
University of Washington  
Seattle, WA

# Overview

- Classes, Vectors, const, overloading
- Tour of computer architecture
- Class Matrix
- Matrix matrix product

# SC'19 Student Cluster Competition Call-Out!

- Teams work with advisor and vendor to design and build a cutting-edge, commercially available cluster constrained by the 3000-watt power limit
- Cluster run a variety of HPC workflows, ranging from being limited by CPU performance to being memory bandwidth limited to I/O intensive
- Teams are comprised of six undergrad or high-school students plus advisor



<https://sc19.supercomputing.org/program/studentssc/student-cluster-competition/>

Team Meetings  
Mondays 5:30PM-8:00PM

# C++ Core Guidelines related to classes

- [C.1: Organize related data into structures \(structs or classes\)](#)
- [C.3: Represent the distinction between an interface and an implementation using a class](#)
- [C.4: Make a function a member only if it needs direct access to the representation of a class](#)
- [C.10: Prefer concrete types over class hierarchies](#)
- [C.11: Make concrete types regular](#)



# Anatomy of a C++ class

Declares  
interface

Hides  
definition

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t          num_rows_;  
    std::vector<double> storage_;  
};
```

Public  
accessors

Private  
data

Maintain  
invariants

# Anatomy of a C++ class

Declares  
interface

Hides

Encapsulation

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
    size_t num_rows() const { return num_rows_; }  
  
private:  
    size_t num_rows_;  
    std::vector<double> storage_;  
};
```

Public  
accessors

Private  
data

Maintain  
invariants

# The Vector class so Far

- Encapsulates vector data
- Member data for dimensions (rows) and for storing elements
- Member function to get number of rows
- Separate interface and implementation via public / private

- Three more things:

- How to bring a Vector into being (“constructors”)
- Function for getting vector data
- Function for setting vector data

Revisit  
operator()

Also called  
function call  
operator

Can create  
*function objects*

# Constructors

- The C++ compiler “knows” about built-in types
- When a variable of a built-in type is declared, the compiler just needs to allocate space for it
- C++ classes are user-defined
- Compiler can do its best (default constructor), but usually we need to do more to create a well-defined object
  
- For example, a well-defined vector should be given its (positive) dimension ***when it is created***. (And the data initialized.)

# Constructors

Built-in type, compiler allocates known amount of space

Default constructor is invoked when variable is declared with no arguments

```
int x = 42;
```

Compiler creates x with **default constructor**

```
Vector x;
```

In this case, the constructor that takes an integer argument

```
Vector x(27);
```

Compiler creates x by making a call to a specific constructor

```
std::cout << "x is " << x.num_rows() << " in length." << std::cout;
```

Create a Vector x with 27 elements

Because that is how we defined the constructor

# Declaring Constructors

```
#include <vector>

class Vector {
public:
    Vector();
    Vector(size_t M);

    size_t num_rows() const { r

private:
    size_t          num_rows_;
    std::vector<double> storage_;
};
```

A constructor is defined using the name of the class

And then the arguments

Can be **overloaded** (different functions distinguished by argument types)

Where have we already seen overloading?

# Defining Constructors

Vector.hpp

```
#include <vector>

class Vector {
public:
    Vector();
    Vector(size_t M);

    size_t num_rows() const { return num_rows_; }

private:
    size_t          num_rows_;
    std::vector<double> storage_;
};
```

Vector.cpp

```
#include "Vector.hpp"

Vector::Vector(size_t M) {
    num_rows_ = M;
    storage_ = std::vector<double>(num_rows);
}

Vector::Vector() {
    num_rows_ = 1;
    storage_ = std::vector<double>(num_rows_);
}
```



# Defining Constructors

Vector.hpp

```
#include <vector>

class Vector {
public:
    Vector() {
        num_rows_ = 1;
        storage_ = std::vector<double>(num_rows);
    }
    Vector(size_t M) {
        num_rows_ = M;
        storage_ = std::vector<double>(num_rows);
    }

    size_t num_rows() const { return num_rows; }

private:
    size_t num_rows_;
    std::vector<double> storage_;
};
```

# Initialization

- We have said that variables should always be initialized
- Different syntaxes

```
int a = 42;
```

```
int b = int(42);
```

```
int c(42);
```

```
int d = { 42 };
```

```
std::vector<double> x = std::vector<double>(27);
```

```
std::vector<double> y(27);
```

c(42)

y(27)

# Defining Constructors

```
#include <vector>
```

```
class Vector {  
public:
```

```
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}
```

```
    size_t num_rows() const { return num_rows_;
```

```
private:
```

```
    size_t num_rows_;
```

```
    std::vector<double> storage_;
```

```
};
```

Note order of  
initialization

Vector.hpp

Initialization syntax  
Introduce with :  
Construct data members

Omit default  
constructor  
(why?)

Note order of  
declaration

# Defining Constructors

Vector.hpp

```
#include <vector>

class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    size_t num_rows() const { return num_rows_; }

private:
    size_t          num_rows_;
    std::vector<double> storage_;
};
```

Initialization

Primordial

Object doesn't yet exist

Object exists

# What Should operator() return?

```
class Vector
public:
    double& operator()(size_t i);

private:
    size_t          num_rows_;
    std::vector<double> storage_;
};
```

Return a *reference* to internal member data

Can assign to internal data through the reference

```
Vector x(5);
```

```
double foo = x(3);
x(2) = 0.0;
```

Can read from internal data through the reference

```
Vector x(5);
```

# All Together

Vector.hpp

```
#include <vector>

class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i) { return storage_[i]; }

    size_t num_rows() const { return num_rows_; }

private:
    size_t          num_rows_;
    std::vector<double> storage_;
};
```

# Reprise operator+()

```
#include <vector>

class Vector {
public:
    Vector operator+(const Vector& y);

private:
    size_t          num_rows_;
    std::vector<double> storage_;
};
```



# Reprise operator+()

C.4: Make a function a member only if it needs direct access to the representation of a class

```
#include <vector>
```

```
class Vector {  
public:
```

```
    Vector operator+(const Vector& y) {  
        Vector z(num_rows_);  
        for (size_t i = 0; i < num_rows_; ++i) {  
            z.storage_[i] = storage_[i] + y.storage[i];  
        }  
    }  
};
```

Data for z

Does this need to be a member?

Data for "x"

Data for y

```
private:  
    size_t          num_rows_;  
    std::vector<double> storage_;  
};
```

# All Together

## Vector.hpp

```
#include <vector>

class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i) { return storage_[i]; }

    size_t num_rows() const { return num_rows_; }

private:
    size_t num_rows_;
    std::vector<double> storage_;
};
```

Can access via  
operator()

Don't need access  
to internals

## Amath583.cpp

```
#include "Vector.hpp"

Vector operator+(const Vector& x, const Vector& y) {
    Vector z(x.num_rows());
    for (size_t i = 0; i < z.num_rows(); ++i) {
        z(i) = x(i) + y(i);
    }
}
```

Return a Vector

Take args by  
const reference

Nicely symmetric

# All Together

## Vector.hpp

```
#include <vector>

class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i) { return storage_[i]; }

    size_t num_rows() const { return num_rows_; }

private:
    size_t num_rows_;
    std::vector<double> storage_;
};
```

## Amath583.hpp

```
#include "Vector.hpp"

Vector operator+(const Vector& x, const Vector& y);
```

## Amath583.cpp

```
#include "Vector.hpp"
#include "amath583.hpp"

Vector operator+(const Vector& x, const Vector& y) {
    Vector z(x.num_rows());
    for (size_t i = 0; i < z.num_rows(); ++i) {
        z(i) = x(i) + y(i);
    }
}
```

# Not quite finished

```
#include "Vector.hpp"

int main() {

    Vector x(100), y(100), z(100), w(100);

    z = x + y;

    return 0;
}
```

```
% c++ constness.cpp
```

```
constness.cpp:20:12: error: no matching function for call to object of type 'const Vector'
```

```
    z(i) = x(i) + y(i);
                ^
```

```
constness.cpp:7:11: note: candidate function not viable: 'this' argument has type
'const Vector', but method is not marked const
```

```
double& operator()(size_t i) { return storage_[i]; }
                ^
```

```
constness.cpp:20:19: error: no matching function for call to object of type 'const Vector'
```

```
    z(i) = x(i) + y(i);
                ^
```

```
constness.cpp:7:11: note: candidate function not viable: 'this' argument has type
'const Vector', but method is not marked const
```

```
double& operator()(size_t i) { return storage_[i]; }
                ^
```

```
|
2 errors generated.
```

# Constness



Vector.hpp

```
#include <vector>

class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i) { return storage_[i]; }

    size_t num_rows() const { return num_rows_; }

private:
    size_t num_rows_;
    std::vector<double> storage_;
};
```

x and y are defined  
to be const

Amath583.hpp

```
#include "Vector.hpp"

Vector operator+(const Vector& x, const Vector& y);
```

“this” is not const

Amath583.cpp

```
#include "Vector.hpp"
#include "amath583.hpp"

Vector operator+(const Vector& x, const Vector& y) {
    Vector z(x.num_rows());
    for (size_t i = 0; i < z.num_rows(); ++i) {
        z(i) = x(i) + y(i);
    }
}
```

# Overloading

```
void foo(size_t i) {  
    std::cout << "foo(size_t i)" << std::endl;  
}
```

Takes a size\_t

```
void foo(double d) {  
    std::cout << "foo(double d)" << std::endl;  
}
```

Takes a double

```
int main() {  
  
    size_t a = 0;  
    double b = 0.0;  
  
    foo(a);  
    foo(b);  
  
    return 0;  
}
```

```
% ./a.out  
foo(size_t i)  
foo(double d)
```

# Overloading

```
void foo(size_t i) {  
    std::cout << "void foo(size_t i)" << std::endl;  
}  
  
size_t foo(size_t i) {  
    std::cout << "size_t foo(size_t i)" << std::endl;  
}
```

Returns void

Returns size\_t

```
% |c++ overload.cpp
```

```
overload.cpp:7:8: error: functions that differ only in their return type cannot be overloaded
```

```
size_t foo(size_t i) {
```

```
~~~~~ ^
```

```
overload.cpp:3:6: note: previous definition is here
```

```
void foo(size_t i) {
```

```
~~~~~ ^
```

```
int main() {  
  
    size_t a = 0;  
    size_t b = 0;  
  
    foo(a);  
    double c = foo(a);  
  
    return 0;  
}
```

Have to pick the function then call it



# No overloading on return values

```
size_t foo(size_t i) {  
    std::cout << "size_t foo(size_t i)" << std::endl;  
  
    return i;  
}
```

```
int main() {
```

```
    size_t a = 0;
```

```
    foo(a);
```

```
    size_t b = foo(a);
```

```
    double c = foo(a);
```

```
    return 0;
```

```
}
```

What happens to the return value is not the concern of the function

Ignore return value

Assign to size\_t

Assign to double

# Constness

```
double parens(double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return y;  
}
```

```
int main() {  
  
    double x = 5.0;  
    double y = parens(x);  
  
    const double z = 5.0;  
    double w = parens(z);  
  
    double a = parens(5.0);  
    double b = parens(x + y);  
  
    const double c = parens(x + y + z + 5.0);  
  
    return 0;  
}
```

x is a ref

```
c++ const3.cpp  
const3.cpp:27:14: error: no matching function for call to 'parens'  
    double w = parens(z, 27);  
                ^~~~~~  
const3.cpp:13:8: note: candidate function not viable: 1st argument ('const double') would lose const  
    qualifier  
double parens(double& x, size_t i) {  
    ^
```

```
const3.cpp:29:14: error: no matching function for call to 'parens'  
    double a = parens(5.0, 27);  
                ^~~~~~
```

```
const3.cpp:13:8: note: candidate function not viable: expects an l-value for 1st argument  
double parens(double& x, size_t i) {  
    ^
```

NOT OKAY

```
const3.cpp:32:20: error: no matching function for call to 'parens'  
    const double c = parens(x + y + 5.0, 27);  
                    ^~~~~~
```

```
const3.cpp:13:8: note: candidate function not viable: expects an l-value for 1st argument  
double parens(double& x, size_t i) {  
    ^
```

Not okay

# Constness

```
double parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return y;  
}
```

x is a const ref

```
./a.out  
called const parens  
called const parens  
called const parens  
called const parens  
called const parens
```

```
int main() {  
  
    double x = 5.0;  
    double y = parens(x);  
  
    const double z = 5.0;  
    double w = parens(z);  
  
    double a = parens(5.0);  
    double b = parens(x + y);  
  
    const double c = parens(x + y + z + 5.0);  
  
    return 0;  
}
```

okay

okay

okay

okay

# Constness

x is a const ref

```
double parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return y;  
}
```

x is a ref

```
double parens(double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return y;  
}
```

```
int main() {
```

```
    double x = 5.0;  
    double y = parens(x);
```

x is lvalue

```
    const double z = 5.0;  
    double w = parens(z);
```

z marked const

```
    double a = parens(5.0);  
    double b = parens(x + y);
```

5.0 is an  
rvalue

```
    const double c = parens(x + y + z + 5.0);
```

x + y is an rvalue

```
    return 0;  
}
```

./a.out

called non const parens

called const parens

called const parens

called const parens

called const parens

# Why not always pass const reference?

```
double parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

Return double

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    parens(z, 27) = p;  
  
    parens(5.0, 27) = p;  
    parens(x + y, 27) = p;  
  
    return 0;  
}
```

c++ const4.cpp

```
const4.cpp:23:17: error: expression is not assignable  
    parens(x, 27) = p;  
    ~~~~~ ^
```

```
const4.cpp:26:17: error: expression is not assignable  
    parens(z, 27) = p;  
    ~~~~~ ^
```

```
const4.cpp:28:19: error: expression is not assignable  
    parens(5.0, 27) = p;  
    ~~~~~ ^
```

```
const4.cpp:29:21: error: expression is not assignable  
    parens(x + y, 27) = p;  
    ~~~~~ ^
```

# Before

```
double parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

# After

```
double& parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```



# Why not always pass const reference?

```
double& parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

But x is const

Return ref to double

Can't return const

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    parens(z, 27) = p;  
  
    parens(5.0, 27) = p;  
    parens(x + y, 27) = p;  
  
    return 0;  
}
```

```
c++ const5.cpp  
const5.cpp:9:10: error: binding value of type 'const double' to reference to type 'double' drops  
    'const' qualifier  
    return x;  
        ^
```

# Before

```
double& parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

## After

```
const double& parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

# Why not always pass const reference?

```
const double& parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    parens(z, 27) = p;  
  
    parens(5.0, 27) = p;  
    parens(x + y, 27) = p;  
  
    return 0;  
}
```

```
c++ const5.cpp  
const5.cpp:26:17: error: cannot assign to return value because function 'parens' returns a const value  
    parens(x, 27) = p;  
    ~~~~~^  
const5.cpp:5:7: note: function 'parens' which returns const-qualified type 'const double &' declared  
    here  
const double& parens(const double& x, size_t i) {  
    ~~~~~  
const5.cpp:29:17: error: cannot assign to return value because function 'parens' returns a const value  
    parens(z, 27) = p;  
    ~~~~~^  
const5.cpp:5:7: note: function 'parens' which returns const-qualified type 'const double &' declared  
    here  
const double& parens(const double& x, size_t i) {  
    ~~~~~  
const5.cpp:31:19: error: cannot assign to return value because function 'parens' returns a const value  
    parens(5.0, 27) = p;  
    ~~~~~^|  
const5.cpp:5:7: note: function 'parens' which returns const-qualified type 'const double &' declared  
    here  
const double& parens(const double& x, size_t i) {  
    ~~~~~  
const5.cpp:32:21: error: cannot assign to return value because function 'parens' returns a const value  
    parens(x + y, 27) = p;  
    ~~~~~^  
const5.cpp:5:7: note: function 'parens' which returns const-qualified type 'const double &' declared  
    here  
const double& parens(const double& x, size_t i) {  
    ~~~~~
```

# Before

```
double& parens(const double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

# After

```
double& parens(double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

# How about no const at all?

```
double& parens(double& x, size_t i) {  
    std::cout << "called const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    parens(z, 27) = p;  
  
    parens(5.0, 27) = p;  
    parens(x + y, 27) = p;  
  
    return 0;  
}
```

```
c++ const5.cpp  
const5.cpp:30:3: error: no matching function for call to 'parens'  
    parens(z, 27) = p;  
    ~~~~~  
const5.cpp:14:9: note: candidate function not viable: 1st argument ('const double') would lose const  
    qualifier  
double& parens(double& x, size_t i) {  
    ^  
const5.cpp:32:3: error: no matching function for call to 'parens'  
    parens(5.0, 27) = p;  
    ~~~~~  
const5.cpp:14:9: note: candidate function not viable: expects an l-value for 1st argument  
double& parens(double& x, size_t i) {  
    ^  
const5.cpp:33:3: error: no matching function for call to 'parens'  
    parens(x + y, 27) = p;  
    ~~~~~  
const5.cpp:14:9: note: candidate function not viable: expects an l-value for 1st argument  
double& parens(double& x, size_t i) {  
    ^
```

# How about no const at all?

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    parens(z, 27) = p;  
  
    parens(5.0, 27) = p;  
    parens(x + y, 27) = p;  
  
    return 0;  
}
```

This makes sense

This *should* be an error

This *should* be an error

This *should* be an error



# More sensible

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    double q = parens(z, 27);  
  
    double r = parens(5.0, 27);  
    double s = parens(x + y, 27);  
  
    return 0;  
}
```

This makes sense

This makes sense

This makes sense

This makes sense

# More sensible

```
double& parens(double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    double q = parens(z, 27);  
  
    double r = parens(5.0, 27);  
    double s = parens(x + y, 27);  
  
    return 0;  
}
```

c++ const6.cpp

const6.cpp:30:14: error: no matching function for call to 'parens'

double q = parens(z, 27);  
                  ^~~~~~

const6.cpp:14:9: note: candidate function not viable: 1st argument ('const double') would lose const  
                  qualifier

double& parens(double& x, size\_t i) {  
                  ^

const6.cpp:32:14: error: no matching function for call to 'parens'

double r = parens(5.0, 27);  
                  ^~~~~~

const6.cpp:14:9: note: candidate function not viable: expects an l-value for 1st argument

double& parens(double& x, size\_t i) {  
                  ^

const6.cpp:33:14: error: no matching function for call to 'parens'

double s = parens(x + y, 27);  
                  ^~~~~~

const6.cpp:14:9: note: candidate function not viable: expects an l-value for 1st argument

double& parens(double& x, size\_t i) {  
                  ^

Oops, need to be const

Going in circles?

# More sensible

```
const double& parens(const double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    double q = parens(z, 27);  
  
    double r = parens(5.0, 27);  
    double s = parens(x + y, 27);  
  
    return 0;  
}
```

```
c++ const6.cpp  
const6.cpp:27:17: error: cannot assign to return value because function 'parens' returns a const value  
    parens(x, 27) = p;  
    ~~~~~^  
const6.cpp:6:7: note: function 'parens' which returns const-qualified type 'const double &' declared  
    here  
const double& parens(const double& x, size_t i) {  
    ^~~~~~
```

Oops, need to be non const

Going in circles?

# Overloading to the rescue

```
const double& parens(const double& x, size_t i) {  
    std::cout << "called non const parens"  
    double y = x;  
    // .. some things  
    return x;  
}
```

const

```
double& parens(double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

Not const

```
int main() {  
    double y = 0.5;  
    double p = 3.14;  
  
    double x = 5.0;  
    parens(x, 27) = p;  
  
    const double z = 5.0;  
    double q = parens(z, 27);  
  
    double r = parens(5.0, 27);  
    double s = parens(x + y, 27);  
  
    return 0;  
}
```

const

Not const

```
./a.out  
called non const parens  
called const parens  
called const parens  
called const parens
```

# What does this have to do with operator()

```
const double& parens(const double& x, size_t i) {  
    std::cout << "called non const parens"  
    double y = x;  
    // .. some things  
    return x;  
}
```

const

const

```
double& parens(double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

Not const

Not const

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
  
private:  
    size_t          num_rows_;  
    std::vector<double> storage_;  
};
```

Where is the const or non-const thing to overload on?

# What does this have to do with operator()

```
const double& parens(const double& x, size_t i) {  
    std::cout << "called non const parens"  
    double y = x;  
    // .. some things  
    return x;  
}
```

const

const

```
double& parens(double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

Not const

Not const

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
    const double& operator()(size_t i) { return storage_[i]; }  
  
private:  
    size_t num_rows_;  
    double* storage_;
```

Only differing by  
return type

Where is the const or non-  
const thing to overload on?

# There is a secret argument

```
const double& parens(const double& x, size_t i) {  
    std::cout << "called non const parens"  
    double y = x;  
    // .. some things  
    return x;  
}
```

const

const

```
double& parens(double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

Not const

Not const

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(size_t i) { return storage_[i]; }  
    const double& operator()(size_t i) { return storage_[i]; }  
};
```

Called "this"

There is a secret argument

There is a secret argument

# There is a secret argument

```
const double& parens(const double& x, size_t i) {  
    std::cout << "called non const parens"  
    double y = x;  
    // .. some things  
    return x;  
}
```

const

const

```
double& parens(double& x, size_t i) {  
    std::cout << "called non const parens" << std::endl;  
    double y = x;  
    // .. some things  
    return x;  
}
```

Not const

Not const

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(Vector *this, size_t i) { return storage_[i]; }  
    const double& operator()(Vector *this, size_t i) { return storage_[i]; }  
  
private:  
    size_t          num_rows_;  
    std::vector<double> storage_;  
};
```

How would we fix our const problem?



# Before

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(Vector *this, size_t i) { return storage_[i]; }  
    const double& operator()(Vector *this, size_t i) { return storage_[i]; }  
  
private:  
    size_t          num_rows_;  
    std::vector<double> storage_;  
};
```

# After

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
    double& operator()(Vector *this, size_t i) { return storage_[i]; }  
    const double& operator()(const Vector *this, size_t i) { return storage_[i]; }  
  
private:  
    size_t          num_rows_;  
    std::vector<double> storage_;  
};
```

const "this"

# After After

```
class Vector {  
public:  
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}  
  
        double& operator()(size_t i)      { return storage_[i]; }  
    const double& operator()(size_t i) const { return storage_[i]; }  
  
private:  
    size_t          num_rows_;  
    std::vector<double> storage_;  
};
```

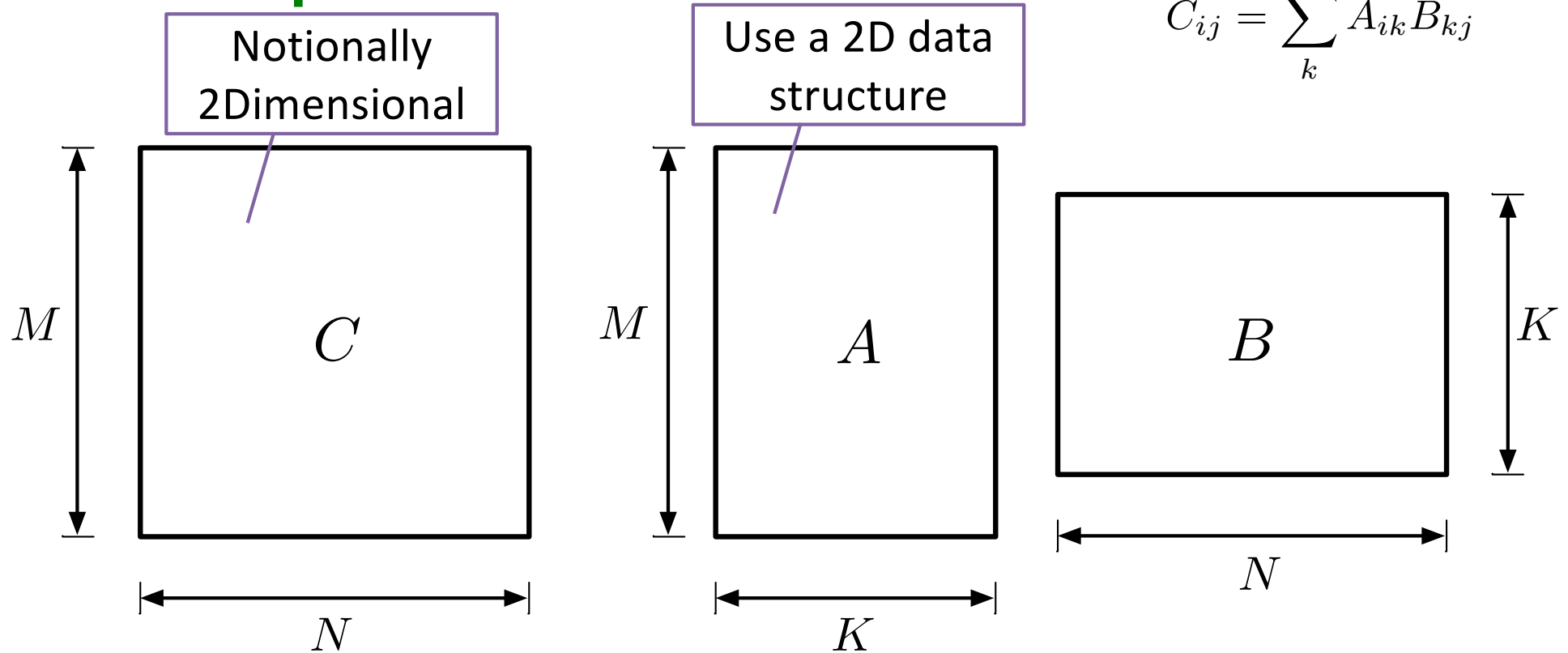
const "this"

# Matrix Representation

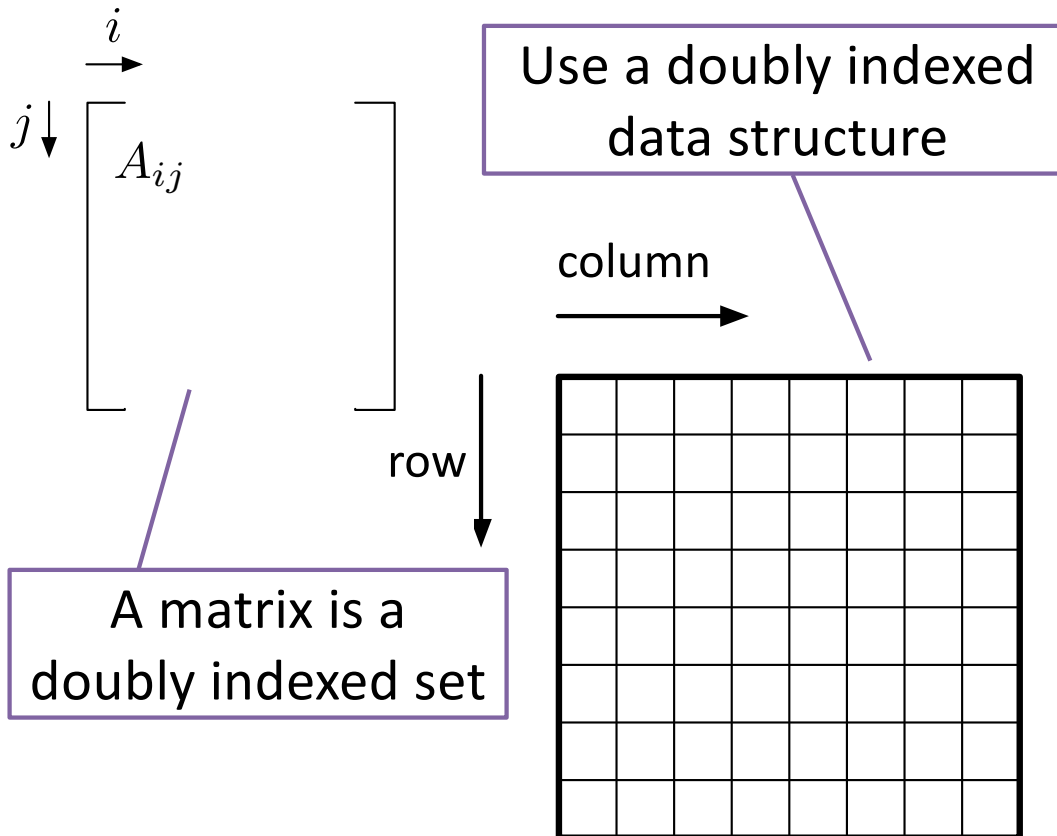
- Two issues
  - Interface (what is the abstraction we want to present?)
  - Implementation (how is the abstraction realized?)
- Sometimes there are tradeoffs
  - Evaluate relative to end user
  - In HPC – performance is most important
  - Elsewhere – safety, ease of use, standards compliance, etc

```
Matrix A(M,K), B(K,N), C(M,N);  
...  
for (int i = 0; i < N; ++i)  
    for (int j = 0; j < N; ++j)  
        for (int k = 0; k < N; ++k)  
            C(i,j) += A(i,k) * B(k,j)
```

# Matrix Representation



# Matrix Representation

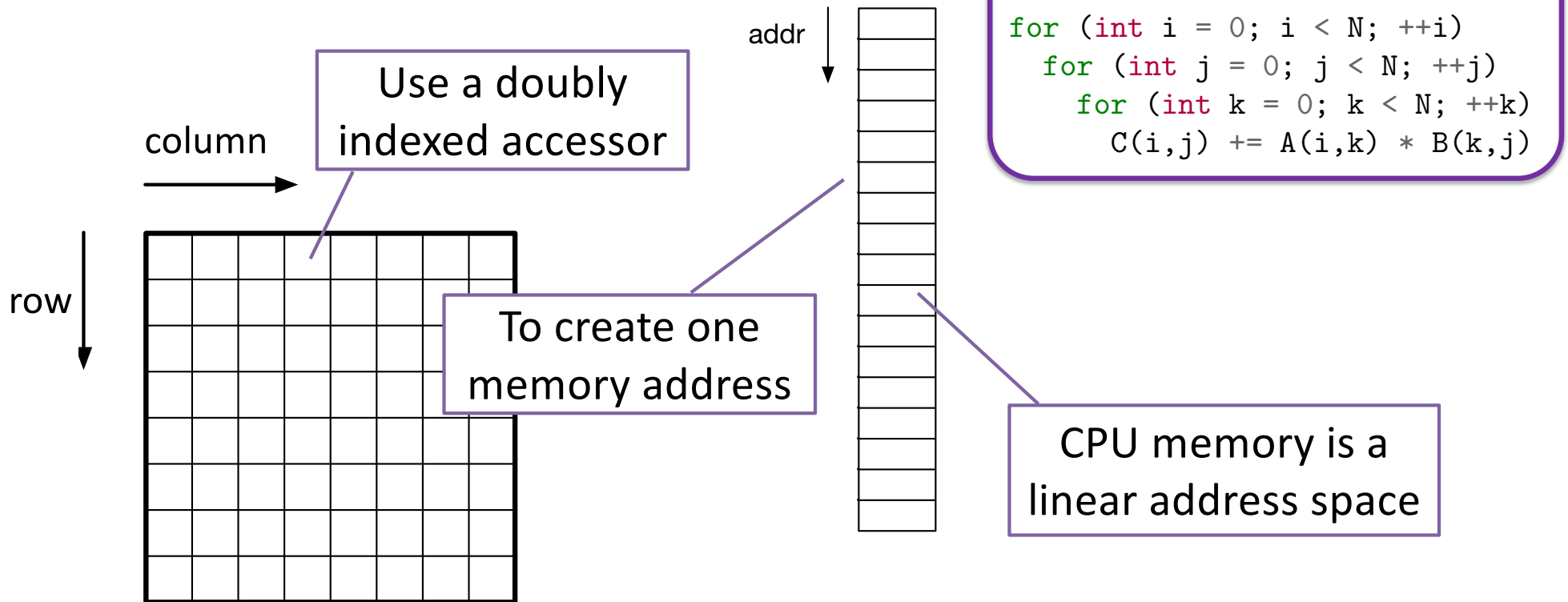


$$C_{ij} = \sum_k A_{ik} B_{kj}$$

```
Matrix A(M,K), B(K,N), C(M,N);  
...  
for (int i = 0; i < N; ++i)  
  for (int j = 0; j < N; ++j)  
    for (int k = 0; k < N; ++k)  
      C(i,j) += A(i,k) * B(k,j)
```

Use a doubly indexed accessor

# Matrix Representation



# Matrix Representation

- To translate double index to single address



```
double **storage_;
```

Array of arrays

Lookup inner pointer  
from outer pointer

```
storage_[i][j]
```

Use inner vector to get  
data element



```
std::vector<std::vector<double> > storage_;
```

Vector of vectors

Lookup inner vector  
from outer vector

```
storage_[i][j]
```

```
std::vector<double> storage_;
```

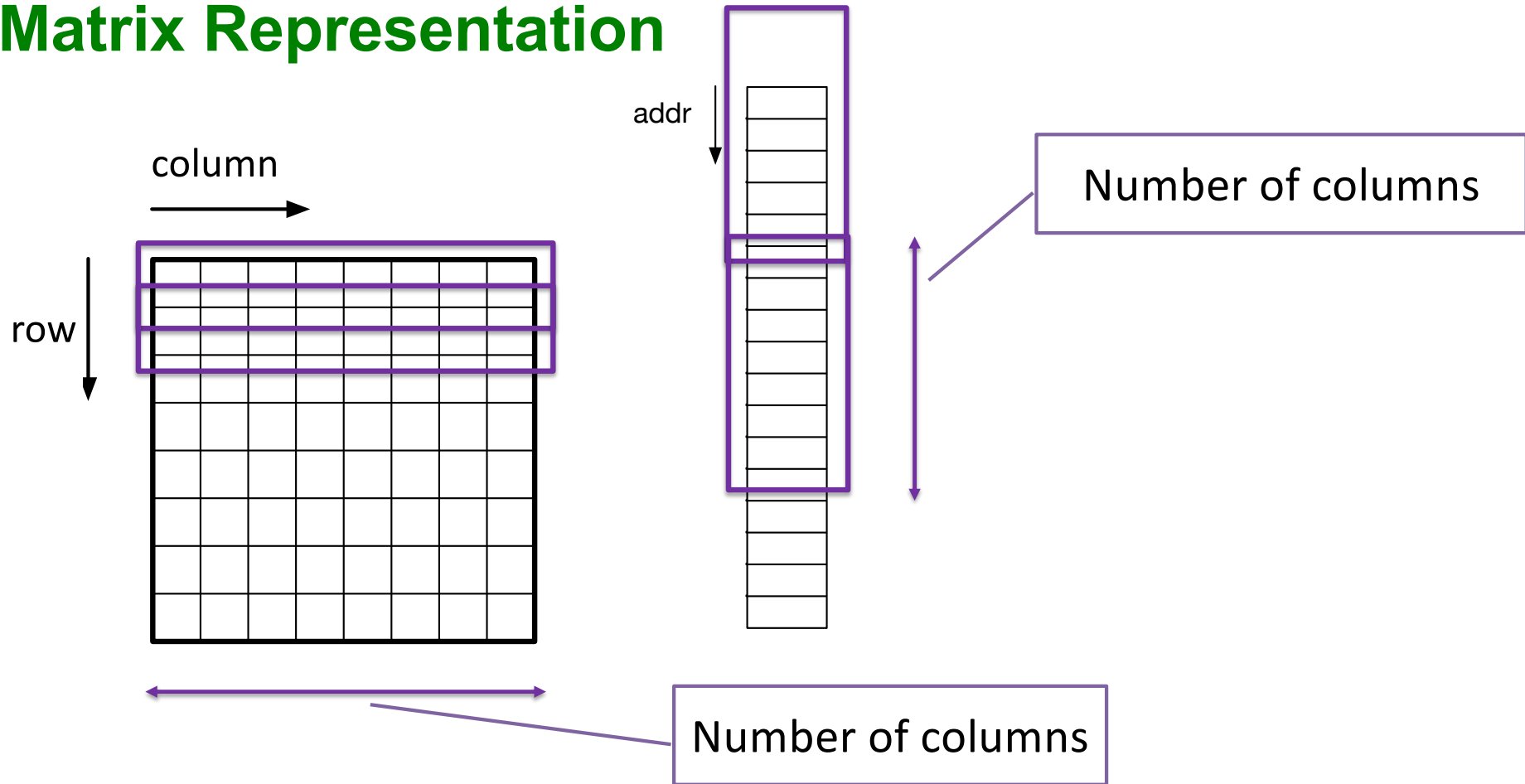
Use single vector to  
get data element

```
storage_[k]
```

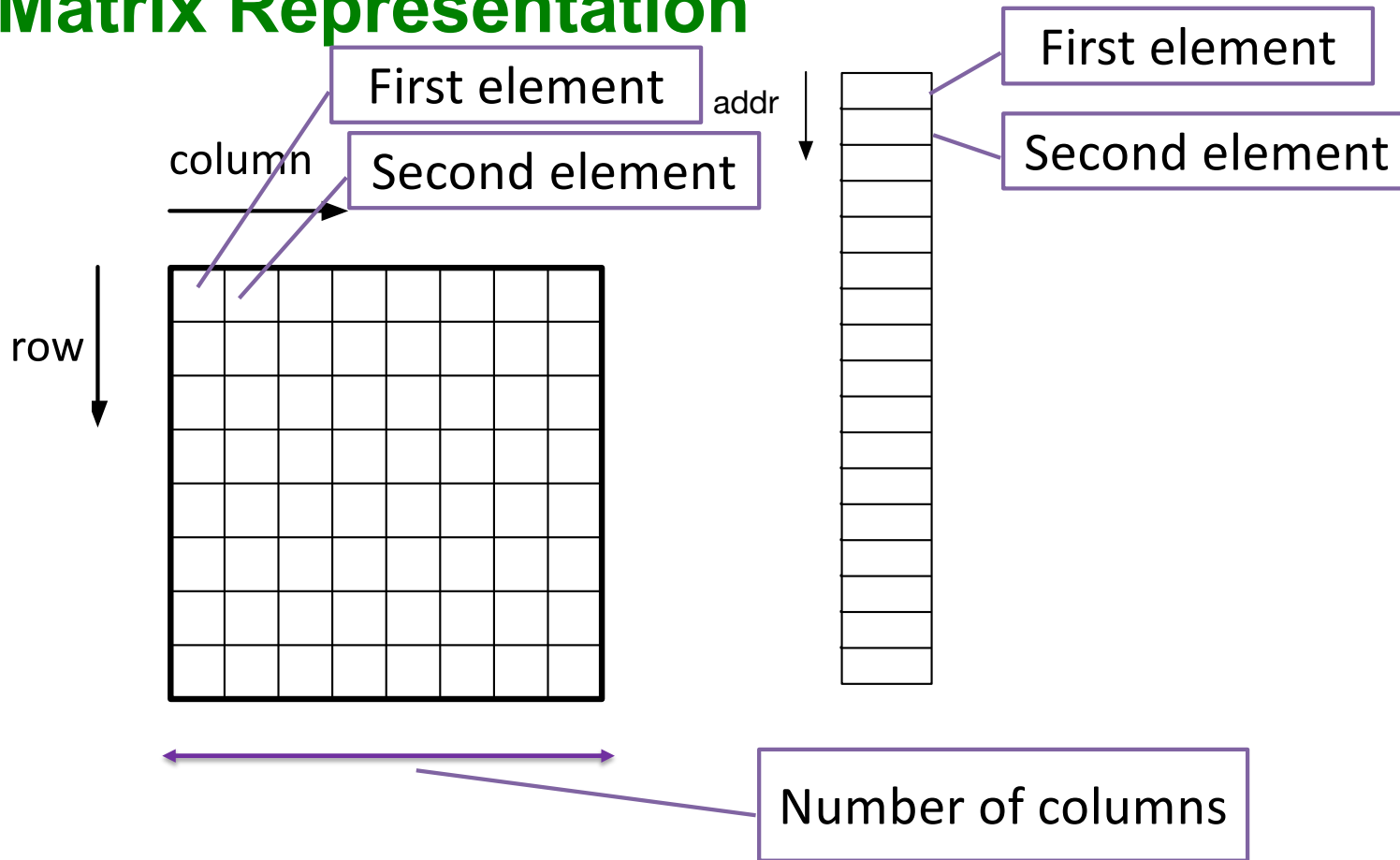
Need to compute this



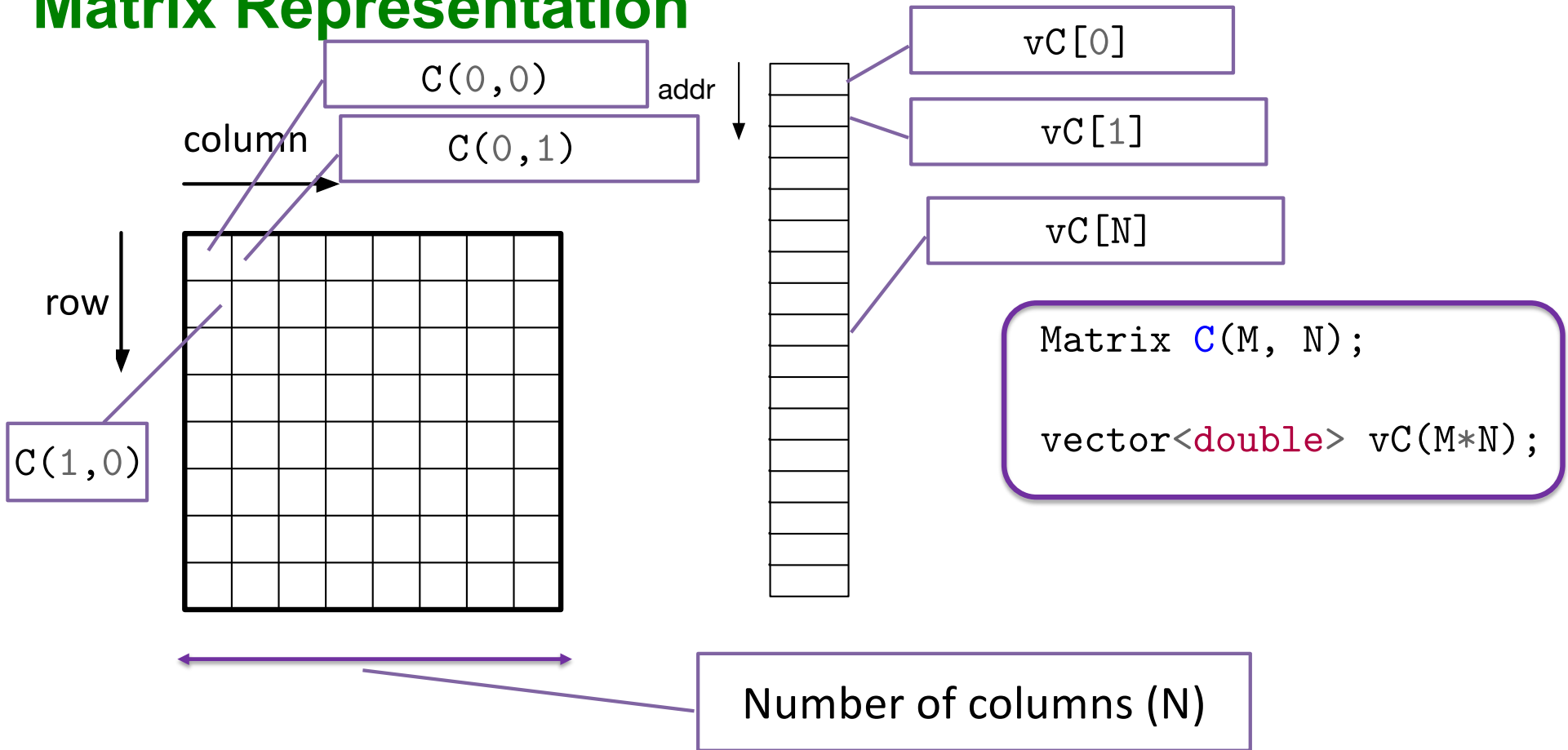
# Matrix Representation



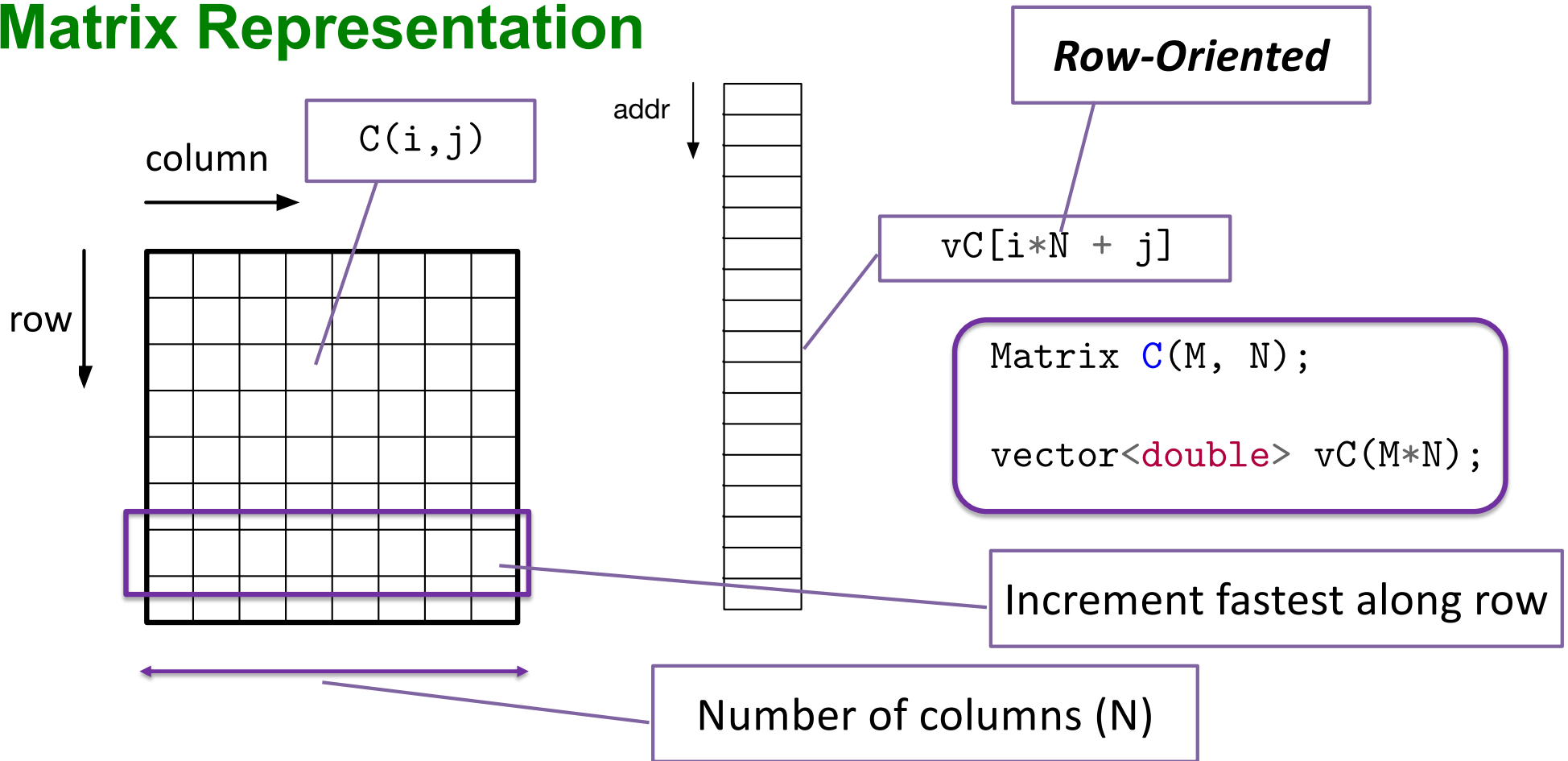
# Matrix Representation



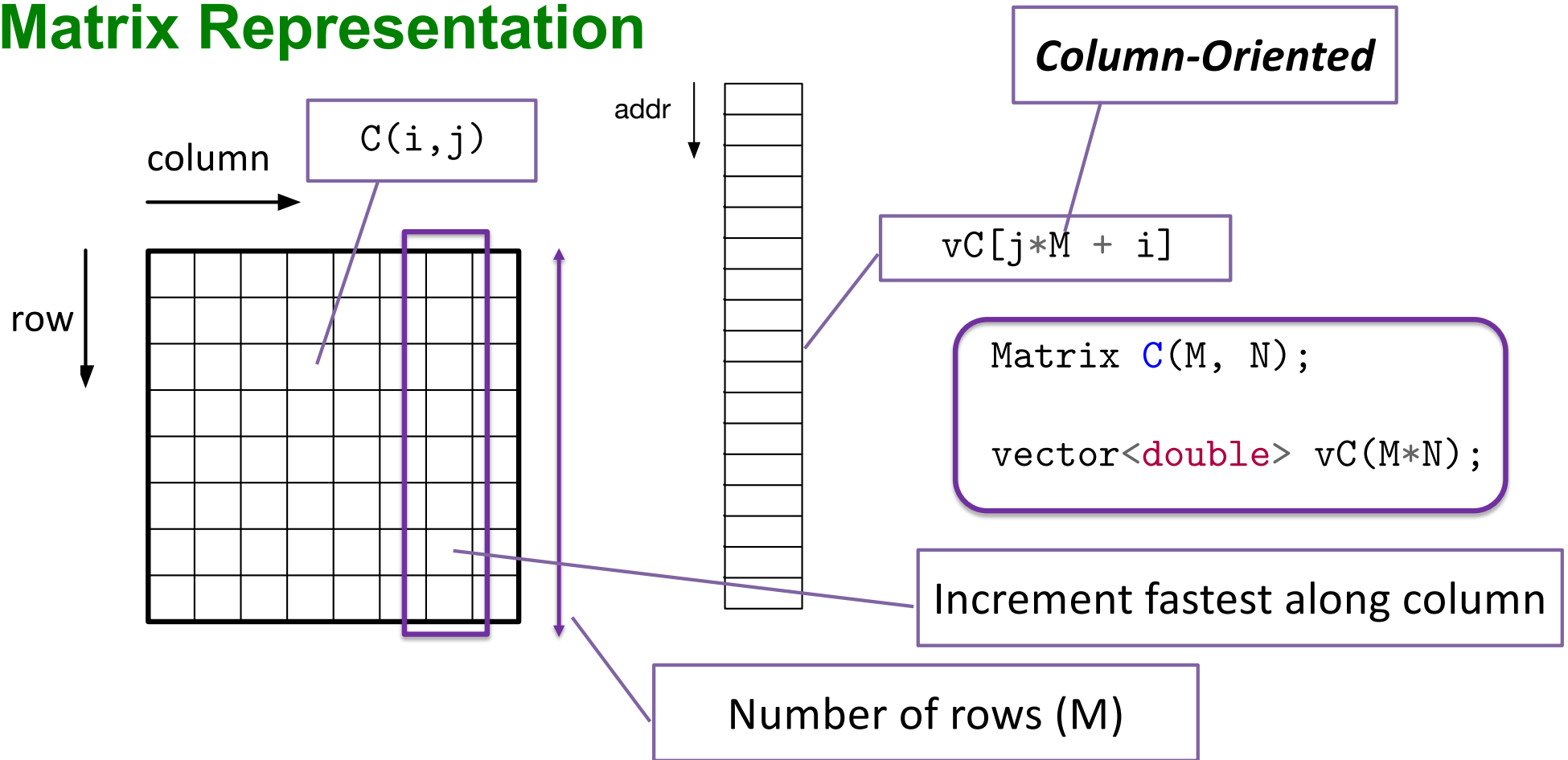
# Matrix Representation



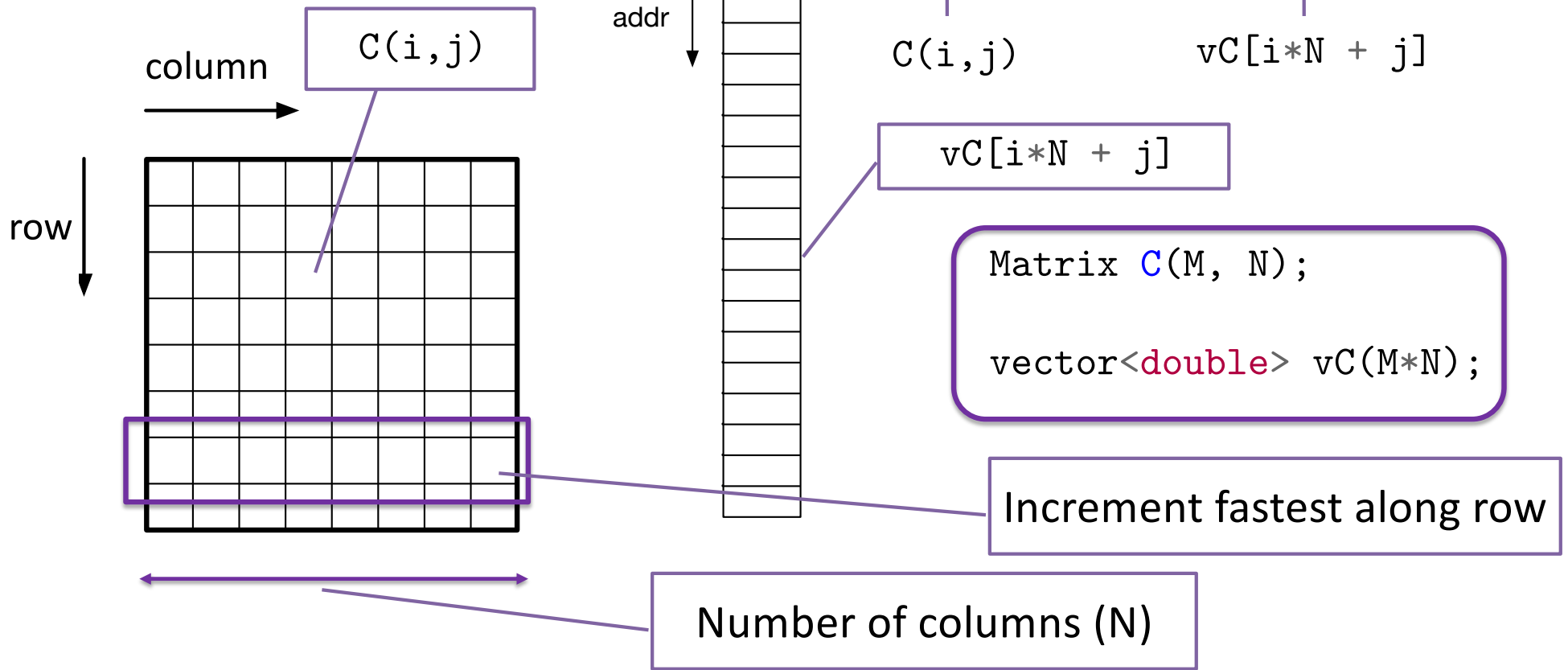
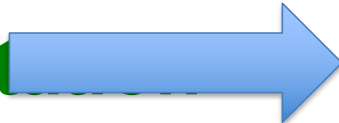
# Matrix Representation



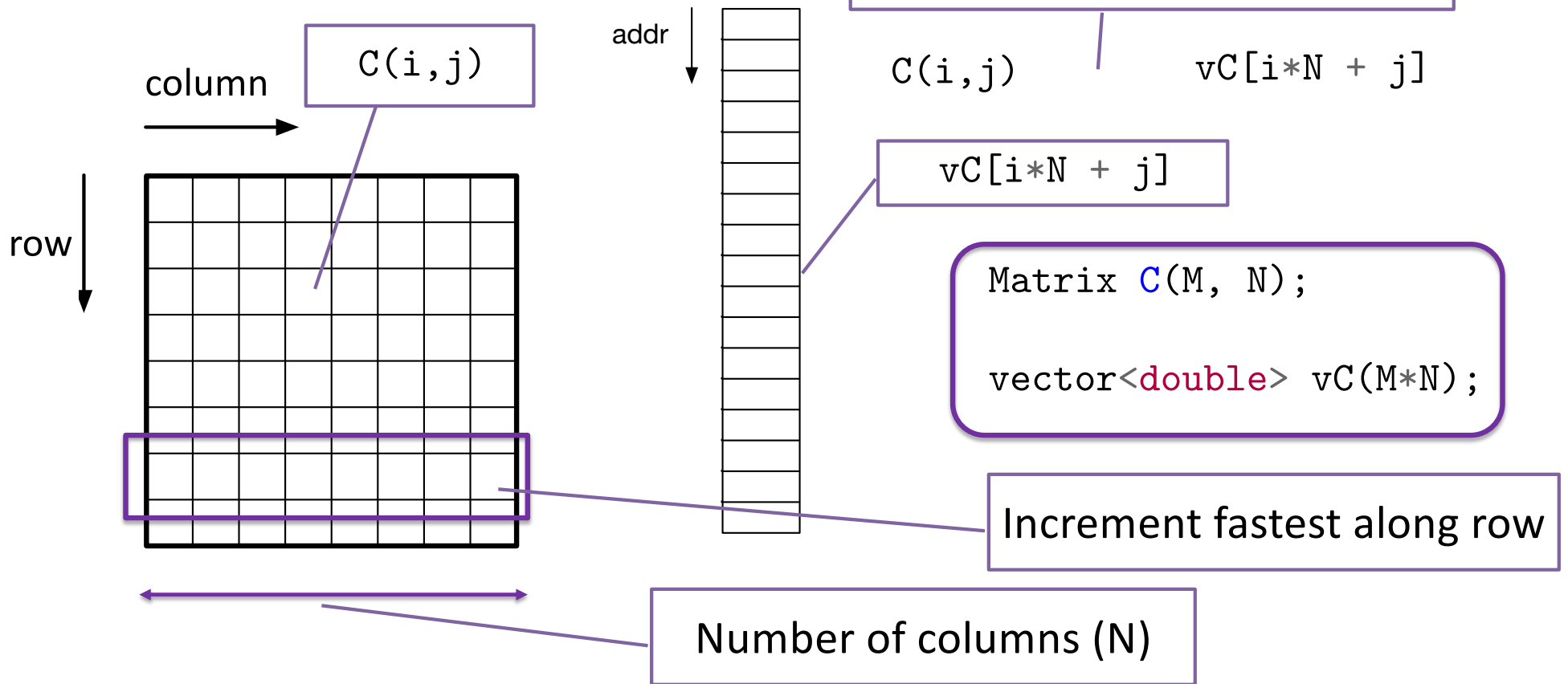
# Matrix Representation



# Matrix Implement



# Matrix Implementation



# Matrix in C++

- Two dimensional accessor `C(i,j)`
- One dimensional access `vC[i*N + j]`
- Simultaneously (and safely) need matrix with
  - (i,j) two dimensional accessor
  - Transparent translation to one dimensional accessor
- Preprocessor?

Only works for C and vC

```
#define C(i,j) vC[i*N+j]
```

Where does N come from



# Matrix in C++

- Two dimensional accessor `C(i,j)`
- One dimensional access `vC[i*N + j]`
  
- A Matrix needs to
  - Have its “own” vector `<double> vC`
  - Have its “own” `N`
  - Have a doubly indexed accessor

# Class Matrix

```
#include <vector>

class Matrix {
public:
    Matrix(size_type M, size_type N)
        : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_) {}
    Matrix(size_type M, size_type N, double init)
        : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_, init) {}

    double &operator()(size_type i, size_type j) {
        return storage_[i * num_cols_ + j];
    }
    const double &operator()(size_type i, size_type j) const {
        return storage_[i * num_cols_ + j];
    }

    size_type num_rows() const { return num_rows_; }
    size_type num_cols() const { return num_cols_; }

private:
    size_type num_rows_, num_cols_;
    std::vector<double> storage_;
};
```

# Class Matrix

```
class Matrix {  
public:  
    Matrix(size_type M, size_type N)  
        : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_) {}  
  
    Matrix(size_type M, size_type N, double init)  
        : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_, init) {}  
  
private:  
    size_type num_rows_, num_cols_;  
    std::vector<double> storage_;  
};
```

# Class Matrix

```
class Matrix {
public:
    double &operator()(size_type i, size_type j) {
        return storage_[i * num_cols_ + j];
    }

    size_type num_rows() const { return num_rows_; }
    size_type num_cols() const { return num_cols_; }

private:
    size_type num_rows_, num_cols_;
    std::vector<double> storage_;
};
```

# Class Matrix

```
class Matrix {
public:
    double &operator()(size_type i, size_type j) {
        return storage_[i * num_cols_ + j];
    }
    const double &operator()(size_type i, size_type j) const {
        return storage_[i * num_cols_ + j];
    }

    size_type num_rows() const { return num_rows_; }
    size_type num_cols() const { return num_cols_; }

private:
    size_type num_rows_, num_cols_;
    std::vector<double> storage_;
};
```

# Class Matrix

```
class Matrix {  
public:  
    double &operator()(size_type i, size_type j) {  
        return storage_[i * num_cols_ + j];  
    }  
    const double &operator()(size_type i, size_type j) const {  
        return storage_[i * num_cols_ + j];  
    }  
  
    size_type num_rows() const { return num_rows_; }  
    size_type num_cols() const { return num_cols_; }  
  
private:  
    size_type num_rows_, num_cols_;  
    std::vector<double> storage_;  
};
```

How would we write the other orientation?

What is the orientation?

Does it matter which?

# Before

```
class Matrix {
public:
    double &operator()(size_type i, size_type j) {
        return storage_[i * num_cols_ + j];
    }
    const double &operator()(size_type i, size_type j) const {
        return storage_[i * num_cols_ + j];
    }

    size_type num_rows() const { return num_rows_; }
    size_type num_Cols() const { return num_cols_; }

private:
    size_type num_rows_, num_cols_;
    std::vector<double> storage_;
};
```

# After

```
class Matrix {
public:
    double &operator()(size_type i, size_type j) {
        return storage_[j * num_rows_ + i];
    }
    const double &operator()(size_type i, size_type j) const {
        return storage_[j * num_rows_ + i];
    }

    size_type num_rows() const { return num_rows_; }
    size_type num_cols() const { return num_cols_; }

private:
    size_type num_rows_, num_cols_;
    std::vector<double> storage_;
};
```



# Finally

```
#include <vector>

class Vector {
public:
    Vector(size_t M) : num_rows_(M), storage_(num_rows_) {}

    double& operator()(size_t i)      { return storage_[i]; }
    const double& operator()(size_t i) const { return storage_[i]; }

    size_t num_rows() { return num_rows_; }

private:
    size_t          num_rows_;
    std::vector<double> storage_;
};
```

# Example: Matrix-Matrix Product

- For matrices  $A \in \mathbb{R}^{M \times K}$  and  $B \in \mathbb{R}^{K \times N}$ , compute  $C \in \mathbb{R}^{M \times N}$

$$C \leftarrow A \times B \quad \text{Defined according to} \quad C_{ij} = \sum_k A_{ik} B_{kj}$$

Locality!  
(Data reuse)

- Workhorse computational kernel (underlying LINPACK, HPL)
- Compute-intensive:  $O(N^3)$  work with  $O(N^2)$  data
- Basic algorithm in C++

Every element is  
accessed N times

Maximize  
locality

```
Matrix A(M,K), B(K,N), C(M,N)
...
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C(i,j) += A(i,k) * B(k,j)
```

# Thank you!

*NORTHWEST INSTITUTE for ADVANCED COMPUTING*

75

AMATH 483/583 High-Performance Scientific Computing Spring 2019  
University of Washington by Andrew Lumsdaine

  
Pacific Northwest  
NATIONAL LABORATORY  
Proudly Operated by Battelle  
for the U.S. Department of Energy

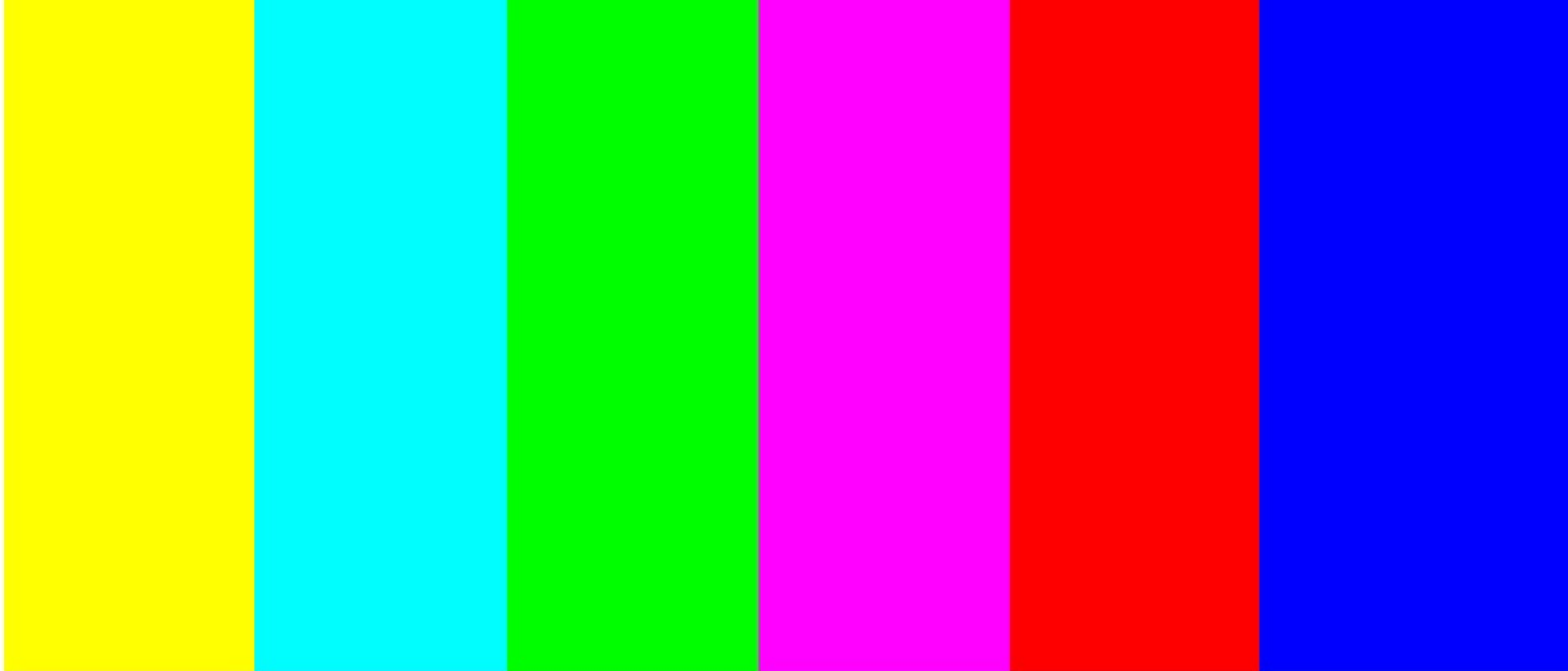
  
UNIVERSITY of  
WASHINGTON



© Andrew Lumsdaine, 2017-2018

Except where otherwise noted, this work is licensed under

<https://creativecommons.org/licenses/by-nc-sa/4.0/>



## Example: Matrix-Matrix Product

- For matrices  $\mathbf{A} \in \mathbb{R}^{M \times K}$  and  $\mathbf{B} \in \mathbb{R}^{K \times N}$ , compute  $\mathbf{C} \in \mathbb{R}^{M \times N}$

$$\mathbf{C} \leftarrow \mathbf{A} \times \mathbf{B} \quad \text{Defined according to} \quad C_{ij} = \sum_k A_{ik} B_{kj}$$

- Workhorse computational kernel (underlying LINPACK, HPL)
- Compute-intensive:  $O(N^3)$  work with  $O(N^2)$  data

Basic algorithm in C

```
double **A, double **B, double **C;
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

## Example: Matrix-Matrix Product

- For matrices  $A \in \mathbb{R}^{M \times K}$  and  $B \in \mathbb{R}^{K \times N}$ , compute  $C \in \mathbb{R}^{M \times N}$   
 $C \leftarrow A \times B$  Defined according to  $C_{ij} = A_{ik} B_{kj}$

- C does not have dynamic multidimensional arrays
- Array of pointers to pointers very bad for performance
- Implementing mapping of 2D to 1D with “leading dimension” arithmetic

```
double *A, double *B, double *C;
int lda, ldb, ldc;
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i*ldc+j] += A[i*lda+k] * B[k*ldb+j];
```

## Example: Matrix-Matrix Product

- For matrices  $A \in \mathbb{R}^{M \times K}$  and  $B \in \mathbb{R}^{K \times N}$ , compute  $C \in \mathbb{R}^{M \times N}$   
 $C \leftarrow A \times B$  Defined according to  $C_{ij} = A_{ik} B_{kj}$

- For notational convenience:

```
#define A(i, j) A[(i)*lda+j]
#define B(i, j) B[(i)*ldb+j]
#define C(i, j) C[(i)*ldc+j]
double *A, double *B, double *C;
int lda, ldb, ldc;
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C(i, j) += A(i, k) * B(k, j);
```

Assume



## Example: Matrix-Matrix Product

- For matrices  $\mathbf{A} \in \mathbb{R}^{M \times K}$  and  $\mathbf{B} \in \mathbb{R}^{K \times N}$ , compute  $\mathbf{C} \in \mathbb{R}^{M \times N}$   
 $\mathbf{C} \leftarrow \mathbf{A} \times \mathbf{B}$  Defined according to  $C_{ij} = A_{ik} B_{kj}$

- For true notational convenience (assuming rest):

```
for (int i = 0; i < M; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < K; ++k)
      C(i, j) += A(i, k) * B(k, j);
```

# Example: Base Case

Matrix Matrix Product Performance



# Improving Locality

- Consider each step of inner loop

```
for (int i = 0; i < M; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < K; ++k)
      C(i, j) += A(i, k) * B(k, j);
```

- Load  $C(i, j)$  into register
- Load  $A(i, k)$  into register
- Load  $B(k, j)$  into register
- Multiply
- Add
- Store

$C(i, j)$

- Four memory operations and two floating point operations per iteration
- 1/3 flop per cycle (if each operation is one cycle)

# Improving Locality

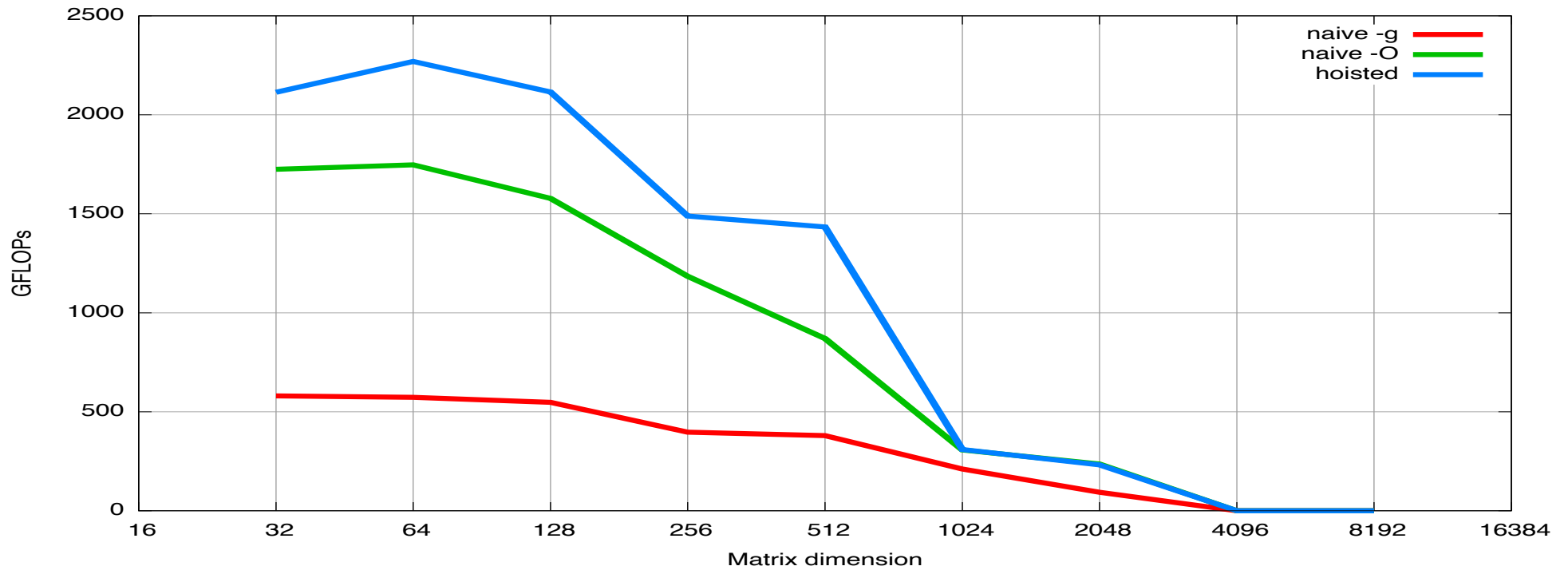
- Consider each step of inner loop

```
for (int i = 0; i < M; ++i)
  for (int j = 0; j < N; ++j) {
    double C00 = C(i, j);
    for (int k = 0; k < K; ++k)
      C00 += A(i, k) * B(k, j);
    C(i, j) = C00;
  }
```

- Load  $A(i, k)$  into register
  - Load  $B(k, j)$  into register
  - Multiply
  - Add
- Two memory operations and two floating point operations per iteration
  - 1/2 flop per cycle (if each operation is one cycle)

# Example: Variable Hoisting

Matrix Matrix Product Performance



# Improving Locality: Unroll and Jam

- Consider each step of inner loop

```

for (int i = 0; i < M; i += 2)
  for (int j = 0; j < N; j += 2) {
    double C00 = C(i, j),    C01 = C(i, j+1);
    double C10 = C(i+1, j), C11 = C(i+1, j+1);
    for (int k = 0; k < K; ++k)
      C00 += A(i, k) * B(k, j); C01 += A(i, k) * B(k, j+1);
      C10 += A(i+1, k) * B(k, j); C11 += A(i+1, k) * B(k, j+1);
    C(i, j) = C00; C(i, j+1) = C01;
    C(i+1, j) = C10; C(i+1, j+1) = C11;
  }

```

- Load  $A(i, k)$  and  $A(i+1, k)$  into registers – note reuse
- Load  $B(k, j)$  and  $B(k, j+1)$  into registers – note reuse
- Multiply (four times)
- Add (four times)
- Four memory operations and eight floating point operations per iteration
- 2/3 flop per cycle (if each operation is one cycle) – 2X the base case

# Example: Register Locality

