Differences between a multidimensional array "[,]" and an array of arrays "[][]" in C#?

Asked 15 years, 9 months ago Modified 8 months ago

Viewed 167k times

520

What are the differences between multidimensional arrays `double[,]` and array of arrays `double[][]` in C#?

If there is a difference?
What is the best use for each one?

c# arrays multidimensional-array jagged-arrays

Share

Improve this question

Follow

edited Feb 9, 2023 at 11:27

Ola Ström
5,271 ● 6 ● 30 ● 49

asked Feb 28, 2009 at 7:55

ecleel
11.9k ● 15 ● 50 ● 48

13 The first `double[,]` is a rectangular array, while `double[][]` is known as a "jagged array". The first will have the same number of "columns" for each row, while the second will (potentially) have a different number of "columns"

for each row. – GreatAndPowerfulOz Aug 19, 2016 at 15:52

I think names are reversed: learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/… – Babu James Aug 7 at 3:31

13 Answers

Sorted by: Highest score (default)

363

Array of arrays (jagged arrays) are faster than multi-dimensional arrays and can be used more effectively. Multidimensional arrays have nicer syntax.

If you write some simple code using jagged and multidimensional arrays and then inspect the compiled assembly with an IL disassembler you will see that the storage and retrieval from jagged (or single dimensional) arrays are simple IL instructions while the same operations for multidimensional arrays are method invocations which are always slower.

Consider the following methods:

```
static void SetElementAt(int[][] array, int i, int j,
{
    array[i][j] = value;
}

static void SetElementAt(int[,] array, int i, int j, i
{
    array[i, j] = value;
}
```

Their IL will be the following:

```
.method private hidebysig static void  SetElementAt(in
                                                    in
                                                    in
                                                    in
{
  // Code size       7 (0x7)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  ldelem.ref
  IL_0003:  ldarg.2
  IL_0004:  ldarg.3
  IL_0005:  stelem.i4
  IL_0006:  ret
} // end of method Program::SetElementAt

.method private hidebysig static void  SetElementAt(in
                                                    in
                                                    in
                                                    in
{
  // Code size       10 (0xa)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  ldarg.3
  IL_0004:  call       instance void int32[0...,0...]:


  IL_0009:  ret
} // end of method Program::SetElementAt
```

When using jagged arrays you can easily perform such operations as row swap and row resize. Maybe in some cases usage of multidimensional arrays will be more safe, but even Microsoft FxCop tells that jagged arrays should

be used instead of multidimensional when you use it to analyse your projects.

Share Improve this answer

Follow

edited Jul 22, 2014 at 6:01

Gabriel Rodriguez

55 ● 10

answered Feb 28, 2009 at 8:07

okutane

14.2k ● 11 ● 60 ● 68

8 @John, measure them yourself, and don't make assumptions. – Hosam Aly Feb 28, 2009 at 14:32

46 Multi-dimensional arrays should logically be more efficent but their implementation by the JIT compiler is not. The above code is not useful since it does not show array access in a loop. – ILoveFortran Feb 28, 2009 at 15:28

3 @Henk Holterman - See my answer below, It might be the case that on windows jagged arrays are fast but one has to realize that this is entirely CLR specific and not the case with e.g. mono... – John Leidegren Mar 1, 2009 at 8:09

17 I know this is an old question, just wondering if the CLR has been optimized for multidimensional arrays since this question was asked. – Anthony Nichols Apr 29, 2016 at 15:11

3 @arthur The C# compiler doesn't do optimizations, the JIT does. Looking at the IL won't tell you how it's optimized. – BlueRaja - Danny Pflughoeft Dec 23, 2017 at 16:42

A multidimensional array creates a nice linear memory layout while a jagged array implies several extra levels of

indirection.

Looking up the value `jagged[3][6]` in a jagged array `var jagged = new int[10][5]` works like this:

- Look up the element at index 3 (which is an array).
- Look up the element at index 6 in that array (which is a value).

For each dimension in this case, there's an additional look up (this is an expensive memory access pattern).

A multidimensional array is laid out linearly in memory, the actual value is found by multiplying together the indexes. However, given the array `var mult = new int[10,30]`, the `Length` property of that multidimensional array returns the total number of elements i.e. 10 * 30 = 300.

The `Rank` property of a jagged array is always 1, but a multidimensional array can have any rank. The `GetLength` method of any array can be used to get the length of each dimension. For the multidimensional array in this example `mult.GetLength(1)` returns 30.

Indexing the multidimensional array is faster. e.g. given the multidimensional array in this example `mult[1,7]` = 30 * 1 + 7 = 37, get the element at that index 37. This is a better memory access pattern because only one memory location is involved, which is the base address of the array.

A multidimensional array therefore allocates a continuous memory block, while a jagged array does not have to be square, e.g. `jagged[1].Length` does not have to equal `jagged[2].Length`, which would be true for any multidimensional array.

Performance

Performance wise, multidimensional arrays should be faster. A lot faster, but due to a really bad CLR implementation they are not.

```
 23.084  16.634  15.215  15.489  14.407  13.691  14.69
 25.782  27.484  25.711  20.844  19.607  20.349  25.86
  5.050   5.085   6.412   5.225   5.100   5.751   6.65
```

The first row are timings of jagged arrays, the second shows multidimensional arrays and the third, well that's how it should be. The program is shown below, FYI this was tested running Mono. (The Windows timings are vastly different, mostly due to the CLR implementation variations).

On Windows, the timings of the jagged arrays are greatly superior, about the same as my own interpretation of what multidimensional array look up should be like, see 'Single()'. Sadly the Windows JIT-compiler is really stupid, and this unfortunately makes these performance discussions difficult, there are too many inconsistencies.

These are the timings I got on Windows, same deal here, the first row are jagged arrays, second multidimensional and third my own implementation of multidimensional, note how much slower this is on Windows compared to Mono.

```
  8.438   2.004   8.439   4.362   4.936   4.533   4.75
  7.414  13.196  11.940  11.832  11.675  11.811  11.81
 11.355  10.788  10.527  10.541  10.745  10.723  10.65
```

Source code:

```
using System;
using System.Diagnostics;
static class ArrayPref
{
    const string Format = "{0,7:0.000} ";
    static void Main()
    {
        Jagged();
        Multi();
        Single();
    }

    static void Jagged()
    {
        const int dim = 100;
        for(var passes = 0; passes < 10; passes++)
        {
            var timer = new Stopwatch();
            timer.Start();
            var jagged = new int[dim][][];
            for(var i = 0; i < dim; i++)
            {
                jagged[i] = new int[dim][];
                for(var j = 0; j < dim; j++)
                {
                    jagged[i][j] = new int[dim];
                    for(var k = 0; k < dim; k++)
```

```
                    {
                        jagged[i][j][k] = i * j * k;
                    }
                }
            }
            timer.Stop();
            Console.Write(Format,
                (double)timer.ElapsedTicks/TimeSpan.Ti
        }
        Console.WriteLine();
    }

    static void Multi()
    {
        const int dim = 100;
        for(var passes = 0; passes < 10; passes++)
        {
            var timer = new Stopwatch();
            timer.Start();
            var multi = new int[dim,dim,dim];
            for(var i = 0; i < dim; i++)
            {
                for(var j = 0; j < dim; j++)
                {
                    for(var k = 0; k < dim; k++)
                    {
                        multi[i,j,k] = i * j * k;
                    }
                }
            }
            timer.Stop();
            Console.Write(Format,
                (double)timer.ElapsedTicks/TimeSpan.Ti
        }
        Console.WriteLine();
    }

    static void Single()
    {
        const int dim = 100;
        for(var passes = 0; passes < 10; passes++)
        {
            var timer = new Stopwatch();
            timer.Start();
```

```
            var single = new int[dim*dim*dim];
            for(var i = 0; i < dim; i++)
            {
                for(var j = 0; j < dim; j++)
                {
                    for(var k = 0; k < dim; k++)
                    {
                        single[i*dim*dim+j*dim+k] = i
                    }
                }
            }
            timer.Stop();
            Console.Write(Format,
                (double)timer.ElapsedTicks/TimeSpan.Ti
        }
        Console.WriteLine();
    }
}
```

Share Improve this answer

Follow

edited Feb 9, 2023 at 11:38

Ola Ström

5,271 ● 6 ● 30 ● 49

answered Feb 28, 2009 at 9:34

John Leidegren

60.9k ● 20 ● 135 ● 157

2 Try timing them yourself, and see how both perform. Jagged arrays are much more optimized in .NET. It may be related to bounds checking, but regardless of the reason, timings and benchmarks clearly show that jagged arrays are faster to access than multi-dimensional ones. – Hosam Aly Feb 28, 2009 at 14:27

12 But your timings appear to be too small (a few milliseconds). At this level you'll have much interference from system services and/or drivers. Make your tests much larger, at least taking a second or two. – Hosam Aly Mar 1, 2009 at 8:12

8 @JohnLeidegren: The fact that multi-dimensional arrays work better when indexing one dimension than another has been understood for half a century, since elements which differ in only one particular dimension will be stored consecutively in memory, and with many types of memory (past and present), accessing consecutive items is faster than accessing distant items. I think in .net one should get optimal results indexing by the last subscript which is what you were doing, but testing the time with the subscripts swapped may be informative in any case. – supercat Aug 27, 2012 at 14:51

19 @supercat: multi-dimensional arrays in C# are stored in row-major order, swapping the order of the subscripts would be slower since you would be accessing the memory non-consecutively. BTW the times reported are no longer accurate, I get almost twice as fast times for multi-dimensional arrays than jagged arrays (tested on latest .NET CLR), which is how it ought to be.. – Amro Nov 5, 2013 at 7:53

11 I know this is a bit pedantic, but I have to mention that this isn't Windows vs Mono, but CLR vs Mono. You sometimes seem to confuse those. The two are not equivalent; Mono works on Windows as well. – Magus Mar 6, 2014 at 17:52

78

Simply put multidimensional arrays are similar to a table in DBMS.
Array of Array (jagged array) lets you have each element hold another array of the same type of variable length.

So, if you are sure that the structure of data looks like a table (fixed rows/columns), you can use a multi-dimensional array. Jagged array are fixed elements & each element can hold an array of variable length

E.g. Psuedocode:

```
int[,] data = new int[2,2];
data[0,0] = 1;
data[0,1] = 2;
data[1,0] = 3;
data[1,1] = 4;
```

Think of the above as a 2x2 table:

> ```
> 1 | 2
> 3 | 4
> ```

```
int[][] jagged = new int[3][];
jagged[0] = new int[4] {  1,  2,  3,  4 };
jagged[1] = new int[2] { 11, 12 };
jagged[2] = new int[3] { 21, 22, 23 };
```

Think of the above as each row having variable number of columns:

> ```
>  1 |  2 |  3 | 4
> 11 | 12
> 21 | 22 | 23
> ```

Share Improve this answer

Follow

edited Nov 25, 2015 at 17:47

Matias Cicero

26.2k ● 20 ● 89 ● 159

answered Feb 28, 2009 at 8:22

shahkalpesh
33.5k • 3 • 65 • 89

4 this is what really matters when deciding what to use.. not this speed thingy.. well speed may become a factor when you have a square array. – Xaser Apr 23, 2016 at 22:00

72

Update .NET 6:

With the release of .NET 6 I decided it was a good time to revisit this topic. I rewrote the test code for new .NET and ran it with the requirement of each part running at least a second. The benchmark was done on AMD Ryzen 5600x.

Results? It's complicated. It seems that Single array is the most performant for smaller and large arrays (< ~25x25x25 & > ~200x200x200) and Jagged arrays being fastest in between. Unfortunately it seems from my testing that multi-dimensional are by far the slowest option. At best performing twice as slow as the fastest option. But! It depends on what you need the arrays for because jagged arrays can take much longer to initialize on 50^3 cube the initialization was roughly 3 times longer than single dimensional. Multi-dimensional was only a little bit slower than single dimensional.

The conclusion? If you need fast code, benchmark it yourself on the machine it's going to run on. CPU architecture can complete change the relative performance of each method.

Numbers!

```
Method name          Ticks/Iteration     Scaled to the
Array size 1x1x1 (10,000,000 iterations):
Jagged:              0.15                4.28
Single:              0.035               1
Multi-dimensional:   0.77                22

Array size 10x10x10 (25,000 iterations):
Jagged:              15                  1.67
Single:              9                   1
Multi-dimensional:   56                  6.2

Array size 25x25x25 (25,000 iterations):
Jagged:              157                 1.3
Single:              120                 1
Multi-dimensional:   667                 5.56

Array size 50x50x50 (10,000 iterations):
Jagged:              1,140               1
Single:              2,440               2.14
Multi-dimensional:   5,210               4.57

Array size 100x100x100 (10,000 iterations):
Jagged:              9,800               1
Single:              19,800              2
Multi-dimensional:   41,700              4.25

Array size 200x200x200 (1,000 iterations):
Jagged:              161,622             1
Single:              175,507             1.086
Multi-dimensional:   351,275             2.17

Array size 500x500x500 (100 iterations):
Jagged:              4,057.413           1.5
Single:              2,709,301           1
Multi-dimensional:   5,359,393           1.98
```

Don't trust me? Run it yourself and verify.

Note: the constant size seems to give jagged arrays an edge, but is not significant enough to change the order in my benchmarks. I have measured in some instance ~7% decrease in performance when using size from user input for jagged arrays, no difference for single arrays and very small difference (~1% or less) for multi-dimensional arrays. It is most prominent in the middle where jagged arrays take the lead.

```
    using System.Diagnostics;

const string Format = "{0,7:0.000} ";
const int TotalPasses = 25000;
const int Size = 50;
Stopwatch timer = new();

var functionList = new List<Action> { Jagged, Single,

Console.WriteLine("{0,5}{1,20}{2,20}{3,20}{4,20}", "Ru
"Ticks/Instance", "ms/Instance");

foreach (var item in functionList)
{
    var warmup = Test(item);
    var run = Test(item);

    Console.WriteLine($"{item.Method.Name}:");
    PrintResult("warmup", warmup);
    PrintResult("run", run);
    Console.WriteLine();
}

static void PrintResult(string name, long ticks)
{
    Console.WriteLine("{0,10}{1,20}{2,20}{3,20}{4,20}"
string.Format(Format, (decimal)ticks / TimeSpan.TicksP
(decimal)ticks / TotalPasses, (decimal)ticks / TotalPa
TimeSpan.TicksPerMillisecond);
}
```

```
long Test(Action func)
{
    timer.Restart();
    func();
    timer.Stop();
    return timer.ElapsedTicks;
}

static void Jagged()
{
    for (var passes = 0; passes < TotalPasses; passes+
    {
        var jagged = new int[Size][][];
        for (var i = 0; i < Size; i++)
        {
            jagged[i] = new int[Size][];
            for (var j = 0; j < Size; j++)
            {
                jagged[i][j] = new int[Size];
                for (var k = 0; k < Size; k++)
                {
                    jagged[i][j][k] = i * j * k;
                }
            }
        }
    }
}

static void Multi()
{
    for (var passes = 0; passes < TotalPasses; passes+
    {
        var multi = new int[Size, Size, Size];
        for (var i = 0; i < Size; i++)
        {
            for (var j = 0; j < Size; j++)
            {
                for (var k = 0; k < Size; k++)
                {
                    multi[i, j, k] = i * j * k;
                }
            }
        }
    }
```

```
}

static void Single()
{
    for (var passes = 0; passes < TotalPasses; passes+
    {
        var single = new int[Size * Size * Size];
        for (var i = 0; i < Size; i++)
        {
            int iOffset = i * Size * Size;
            for (var j = 0; j < Size; j++)
            {
                var jOffset = iOffset + j * Size;
                for (var k = 0; k < Size; k++)
                {
                    single[jOffset + k] = i * j * k;
                }
            }
        }
    }
}

static void SingleStandard()
{
    for (var passes = 0; passes < TotalPasses; passes+
    {
        var single = new int[Size * Size * Size];
        for (var i = 0; i < Size; i++)
        {
            for (var j = 0; j < Size; j++)
            {
                for (var k = 0; k < Size; k++)
                {
                    single[i * Size * Size + j * Size
                }
            }
        }
    }
}
```

Lesson learned: Always include CPU in benchmarks, because it makes a difference. Did it this time? I don't

know but I suspect it might've.

Original answer:

I would like to update on this, because in **.NET Core multi-dimensional arrays are faster than jagged arrays**. I ran the tests from John Leidegren and these are the results on .NET Core 2.0 preview 2. I increased the dimension value to make any possible influences from background apps less visible.

```
Debug (code optimalization disabled)
Running jagged
187.232 200.585 219.927 227.765 225.334 222.745 224.03

Running multi-dimensional
130.732 151.398 131.763 129.740 129.572 159.948 145.46

Running single-dimensional
 91.153 145.657 111.974  96.436 100.015  97.640  94.58


Release (code optimalization enabled)
Running jagged
108.503 95.409 128.187 121.877 119.295 118.201 102.321

Running multi-dimensional
 62.292  60.627  60.611  60.883  61.167  60.923  62.08

Running single-dimensional
 34.974  33.901  34.088  34.659  34.064  34.735  34.91
```

I looked into disassemblies and this is what I found

`jagged[i][j][k] = i * j * k;` needed 34 instructions to execute

`multi[i, j, k] = i * j * k;` needed 11 instructions to execute

`single[i * dim * dim + j * dim + k] = i * j * k;` needed 23 instructions to execute

I wasn't able to identify why single-dimensional arrays were still faster than multi-dimensional but my guess is that it has to do with some optimalization made on the CPU

Share Improve this answer Follow

edited Dec 10, 2021 at 11:34

answered Jul 31, 2017 at 7:46

adsamcik

1,325 • 13 • 14

52

Preface: This comment is intended to address the answer provided by okutane regarding the performance difference between jagged arrays and multidimensional ones.

The assertion that one type is slower than the other because of the method calls isn't correct. One is slower than the other because of more complicated bounds-checking algorithms. You can easily verify this by looking, not at the IL, but at the compiled assembly. For example, on my 4.5 install, accessing an element (via pointer in edx) stored in a two-dimensional array pointed to by ecx with indexes stored in eax and edx looks like so:

```
sub eax,[ecx+10]
cmp eax,[ecx+08]
jae oops //jump to throw out of bounds exception
sub edx,[ecx+14]
cmp edx,[ecx+0C]
jae oops //jump to throw out of bounds exception
imul eax,[ecx+0C]
add eax,edx
lea edx,[ecx+eax*4+18]
```

Here, you can see that there's no overhead from method calls. The bounds checking is just very convoluted thanks to the possibility of non-zero indexes, which is a functionality not on offer with jagged arrays. If we remove the `sub`, `cmp`, and `jmp`s for the non-zero cases, the code pretty much resolves to `(x*y_max+y)*sizeof(ptr)+sizeof(array_header)`. This calculation is about as fast (one multiply could be replaced by a shift, since that's the whole reason we choose bytes to be sized as powers of two bits) as anything else for random access to an element.

Another complication is that there are plenty of cases where a modern compiler will optimize away the nested bounds-checking for element access while iterating over a single-dimension array. The result is code that basically just advances an index pointer over the contiguous memory of the array. Naive iteration over multi-dimensional arrays generally involves an extra layer of nested logic, so a compiler is less likely to optimize the operation. So, even though the bounds-checking overhead of accessing a single element amortizes out to constant runtime with respect to array dimensions and

sizes, a simple test-case to measure the difference may take many times longer to execute.

Share Improve this answer

Follow

edited Jul 6, 2022 at 21:41

zcoop98

3,052 ● 2 ● 22 ● 36

answered Oct 28, 2013 at 16:23

Eglin

592 ● 5 ● 10

3 Thanks for correcting the answer of okutane (not Dmitry). It is annoying that people give wrong answers on Stackoverflow and get 250 up-votes while others give correct answers and get far less. But at the end the IL code is irrelevant. You have to really MEASURE the speed to say anything about performance. Did you do that? I think the difference will be ridiculous. – Elmue May 29, 2017 at 19:13

15

Multi-dimension arrays are (n-1)-dimension matrices.

So `int[,] square = new int[2,2]` is square matrix 2x2, `int[,,] cube = new int [3,3,3]` is a cube - square matrix 3x3. Proportionality is not required.

Jagged arrays are just array of arrays - an array where each cell contains an array.

So MDA are proportional, JD may be not! Each cell can contains an array of arbitrary length!

Share Improve this answer

answered Feb 28, 2009 at 20:23

Follow

abatishchev
100k ● 88 ● 301 ● 442

7

This might have been mentioned in the above answers but not explicitly: with jagged array you can use `array[row]` to refer a whole row of data, but this is not allowed for multi-d arrays.

Share Improve this answer

Follow

answered Sep 2, 2015 at 21:46

Iznt
2,576 ● 2 ● 26 ● 29

4

In addition to the other answers, note that a multidimensional array is allocated as one big chunky object on the heap. This has some implications:

1. Some multidimensional arrays will get allocated on the Large Object Heap (LOH) where their equivalent jagged array counterparts would otherwise not have.
2. The GC will need to find a single contiguous free block of memory to allocate a multidimensional array, whereas a jagged array might be able to fill in gaps caused by heap fragmentation... this isn't usually an issue in .NET because of compaction, but the LOH doesn't get compacted by default (you have to ask for it, and you have to ask every time you want it).
3. You'll want to look into `<gcAllowVeryLargeObjects>` for multidimensional arrays **way** before the issue will ever come up if you only ever use jagged arrays.

Share Improve this answer

Follow

answered Jun 16, 2017 at 17:53

Joe Amenta
5,191 ● 2 ● 32 ● 41

4

I thought I'd chime in here from the future with some performance results from .NET 5, seen as that will be the platform which everyone uses from now on.

These are the same tests that John Leidegren used (in 2009).

My results (.NET 5.0.1):

```
Debug:
(Jagged)
5.616   4.719   4.778   5.524   4.559   4.508   5.91

(Multi)
6.336   7.477   6.124   5.817   6.516   7.098   5.27

(Single)
4.688   3.494   4.425   6.176   4.472   4.347   4.97


Release(code optimizations on):
(Jagged)
2.614   2.108   3.541   3.065   2.172   2.936   1.68

(Multi)
3.371   4.690   4.502   4.153   3.651   3.637   3.58

(Single)
1.934   2.102   2.246   2.061   1.941   1.900   2.17
```

Ran on a a 6 core 3.7GHz AMD Ryzen 1600 machine.

It looks as though the performance ratio is still roughly the same. I'd say unless you're really optimizing hard, just use multi-dimensional arrays as the syntax is slightly easier to use.

Share Improve this answer

Follow

answered Mar 13, 2021 at 17:07

Zintom
234 • 3 • 12

3

Jagged arrays are arrays of arrays or arrays in which each row contains an array of its own.

These arrays can have lengths different than those in the other rows.

Declaration and Allocation an Array of Arrays

The only difference in the declaration of the jagged arrays compared to the regular multidimensional array is that we do not have just one pair of brackets. With the jagged arrays, we have a pair of brackets per dimension. We allocate them this way:

```
int[][] exampleJaggedArray;
jaggedArray = new int[2][];
```

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[3];
```

Run code snippet Expand snippet

The Initializing an array of arrays

```
int[][] exampleJaggedArray = {
new int[] {5, 7, 2},
new int[] {10, 20, 40},
new int[] {3, 25}
};
```

Run code snippet Expand snippet

Memory Allocation

Jagged arrays are an aggregation of references. A jagged array does not directly contain any arrays, but rather has elements pointing to them. The size is unknown and that is why CLR just keeps references to the internal arrays. After we allocate memory for one array-element of the jagged array, then the reference starts pointing to the newly created block in the dynamic memory.

The variable exampleJaggedArray is stored in the execution stack of the program and points to a block in the dynamic memory, which contains a sequence of three

references to other three blocks in memory; each of them contains an array of integer numbers – the elements of the jagged array:

Share Improve this answer Follow

answered Oct 12, 2021 at 14:29

Neil Meyer
581 • 7 • 15

2

I am parsing .il files generated by ildasm to build a database of assemnblies, classes, methods, and stored procedures for use doing a conversion. I came across the following, which broke my parsing.

```
.method private hidebysig instance uint32[0...,0...]
        GenerateWorkingKey(uint8[] key,
                           bool forEncryption) cil man
```

The book Expert .NET 2.0 IL Assembler, by Serge Lidin, Apress, published 2006, Chapter 8, Primitive Types and Signatures, pp. 149-150 explains.

`<type>[]` is termed a Vector of `<type>`,

`<type>[<bounds> [<bounds>**] ]` is termed an array of `<type>`

`**` means may be repeated, `[ ]` means optional.

Examples: Let `<type> = int32`.

1) `int32[...,...]` is a two-dimensional array of undefined lower bounds and sizes

2) `int32[2...5]` is a one-dimensional array of lower bound 2 and size 4.

3) `int32[0...,0...]` is a two-dimensional array of lower bounds 0 and undefined size.

Tom

Share Improve this answer

Follow

edited Feb 12, 2017 at 20:19

Alf Moh

7,377 ● 5 ● 43 ● 50

answered Feb 2, 2017 at 17:56

Thomas C Hutchinson

21 ● 2

1

Using a test based on the one by John Leidegren, I benchmarked the result using .NET 4.7.2, which is the relevant version for my purposes and thought I could share. I originally started with this comment in the dotnet core GitHub repository.

It appears that the performance varies greatly as the array size changes, at least on my setup, 1 processor xeon with 4physical 8logical.

w = initialize an array, and put int i * j in it. wr = do w, then in another loop set int x to [i,j]

As array size grows, multidimensional appears to outperform.

Size	rw	Method	Mean	Error	StdDev	Gen 0
1800*500	w	Jagged	2.445 ms	0.0959 ms	0.1405 ms	5
1800*500	w	Multi	3.079 ms	0.2419 ms	0.3621 ms	2
2000*4000	w	Jagged	50.29 ms	3.262 ms	4.882 ms	59
2000*4000	w	Multi	26.34 ms	1.797 ms	2.690 ms	2
2000*4000	wr	Jagged	55.30 ms	3.066 ms	4.589 ms	59
2000*4000	wr	Multi	32.23 ms	2.798 ms	4.187 ms	2
1000*2000	wr	Jagged	11.18 ms	0.5397 ms	0.8078 ms	14
1000*2000	wr	Multi	6.622 ms	0.3238 ms	0.4847 ms	2

Update: last two tests with double[,] instead of int[,]. The difference appears significant considering the errors. With int, ratio of mean for jagged vs md is between 1.53x and 1.86x, with doubles it is 1.88x and 2.42x.

Size	rw	Method	Mean	Error	StdDev	Gen 0/
1000*2000	wr	Jagged	26.83 ms	1.221 ms	1.790 ms	306

Size	rw	Method	Mean	Error	StdDev	Gen 0/
1000*2000	wr	Multi	12.61 ms	1.018 ms	1.524 ms	156

Share Improve this answer Follow

edited Jun 11, 2021 at 9:51

answered Jun 11, 2021 at 9:36

Kocas

332 ● 1 ● 14

Scalability is so important, thank you for including it here. The comparisons at larger sizes are definitely a significant piece of information, especially considering the use of these types of arrays with AI. – Travis J Aug 11 at 17:41

0

The main difference is in their structure: jagged arrays are arrays of arrays with different lengths, while multi-dimensional arrays have a fixed length for each dimension.

Jagged arrays are more faster compared to multi dimensional arrays. However, multi dimensional arrays are clean in syntax structure.

I have found a very good article on arrays at here

Share Improve this answer

Follow

answered Apr 3 at 14:02

Asif Ghanchi
242 ● 3 ● 11